Network Working Group
Internet-Draft
Obsoletes: 2068, 2616, 2617
(if approved)
Intended status: Standards Track
Expires: May 14, 2008

R. Fielding, Ed.
Day Software
J. Gettys
J. Mogul
HP
H. Frystyk
Microsoft
L. Masinter
Adobe Systems
P. Leach
Microsoft
T. Berners-Lee
W3C/MIT

November 11, 2007

HTTP/1.1, part 1: URIs, Connections, and Message Parsing
draft-fielding-http-p1-messaging-00

Status of this Memo

Copyright Notice

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World Wide Web global information initiative since 1990. This document is Part 1 of the eight-part specification that defines the protocol referred to as "HTTP/1.1" and, taken together, updates RFC 2616 and RFC 2617. Part 1 provides an overview of HTTP and its associated terminology, defines the "http" and "https" Uniform Resource Identifier (URI) schemes, defines the generic message syntax and parsing requirements for HTTP message frames, and describes general security concerns for implementations.

Table of Contents

1. Introduction

This document will define aspects of HTTP related to overall network operation, message framing, interaction with transport protocols, and URI schemes. Right now it only includes the extracted relevant sections of [RFC2616] and [RFC2617].

1.1. Purpose

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0, as defined by RFC 1945 [RFC1945], improved the protocol by allowing messages to be in the format of MIME-like messages, containing metainformation about the data transferred and modifiers on the request/response semantics. However, HTTP/1.0 does not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, or virtual hosts. In addition, the proliferation of incompletely-implemented applications calling themselves "HTTP/1.0" has necessitated a protocol version change in order for two communicating applications to determine each other's true capabilities.

This specification defines the protocol referred to as "HTTP/1.1". This protocol includes more stringent requirements than HTTP/1.0 in order to ensure reliable implementation of its features.

Practical information systems require more functionality than simple retrieval, including search, front-end update, and annotation. HTTP allows an open-ended set of methods and headers that indicate the purpose of a request [RFC2324]. It builds on the discipline of reference provided by the Uniform Resource Identifier (URI) [RFC1630], as a location (URL) [RFC1738] or name (URN) [RFC1737], for indicating the resource to which a method is to be applied. Messages are passed in a format similar to that used by Internet mail [RFC822] as defined by the Multipurpose Internet Mail Extensions (MIME) [RFC2045].

HTTP is also used as a generic protocol for communication between user agents and proxies/gateways to other Internet systems, including those supported by the SMTP [RFC821], NNTP [RFC3977], FTP [RFC959], Gopher [RFC1436], and WAIS [WAIS] protocols. In this way, HTTP allows basic hypermedia access to resources available from diverse applications.

1.2. Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

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

1.3. Terminology

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

connection

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

message

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

request

An HTTP request message, as defined in Section 5.

response

An HTTP response message, as defined in Section 6.

resource

A network data object or service that can be identified by a URI, as defined in Section 3.2. Resources may be available in multiple representations (e.g. multiple languages, data formats, size, and resolutions) or vary in other ways.

entity

The information transferred as the payload of a request or response. An entity consists of metainformation in the form of entity-header fields and content in the form of an entity-body, as described in [Part 3].

representation

An entity included with a response that is subject to content negotiation, as described in [Part 3]. There may exist multiple representations associated with a particular response status.

content negotiation

The mechanism for selecting the appropriate representation when servicing a request, as described in [Part 3]. The representation of entities in any response can be negotiated (including error responses).

variant

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

client

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

user agent

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

server

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

origin server

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

proxy

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

gateway

A server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway.

tunnel

An intermediary program which is acting as a blind relay between two connections. Once active, a tunnel is not considered a party to the HTTP communication, though the tunnel may have been initiated by an HTTP request. The tunnel ceases to exist when both ends of the relayed connections are closed.

cache

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

cacheable

A response is cacheable if a cache is allowed to store a copy of the response message for use in answering subsequent requests. The rules for determining the cacheability of HTTP responses are

defined in [Part 6]. Even if a resource is cacheable, there may be additional constraints on whether a cache can use the cached copy for a particular request.

upstream/downstream

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

inbound/outbound

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

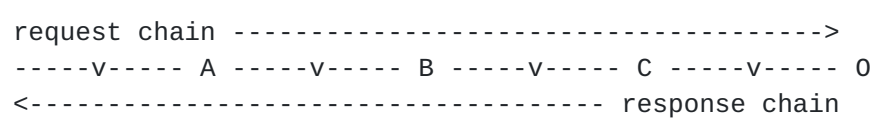
1.4. Overall Operation

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible entity-body content. The relationship between HTTP and MIME is described in [Part 3].

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

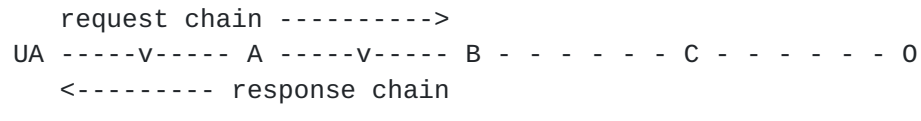
A more complicated situation occurs when one or more intermediaries are present in the request/response chain. There are three common forms of intermediary: proxy, gateway, and tunnel. A proxy is a forwarding agent, receiving requests for a URI in its absolute form, rewriting all or part of the message,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

of the messages.



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

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



Not all responses are usefully cacheable, and some requests may contain modifiers which place special requirements on cache behavior. HTTP requirements for cache behavior and cacheable responses are defined in [Part 6].

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

HTTP communication usually takes place over TCP/IP connections. The default port is TCP 80 [RFC1700], but other ports can be used. This does not preclude HTTP from being implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport; any protocol that provides such guarantees can be used; the mapping of the HTTP/1.1 request and response structures onto the transport data units of the protocol in question is outside the scope of this specification.

In HTTP/1.0, most implementations used a new connection for each request/response exchange. In HTTP/1.1, a connection may be used for one or more request/response exchanges, although connections may be closed for a variety of reasons (see Section 7.1).

2. Notational Conventions and Generic Grammar

2.1. Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [RFC822]. Implementors will need to be familiar with the notation in order to understand this specification. The augmented BNF includes the following constructs:

name = definition

The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal "=" character. White space is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

"literal"

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

rule1 | rule2

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

(rule1 rule2)

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

*rule

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

[rule]

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

N rule

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

#rule

A construct "#" is defined, similar to "*", for defining lists of elements. The full form is "<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by one or more commas (",") and OPTIONAL linear white space (LWS). This makes the usual form of lists very easy; a rule such as

(*LWS element *(*LWS "," *LWS element))

can be shown as

1#element

Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, "(element), , (element) " is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element MUST be present. Default values are 0 and infinity so that "#element" allows any number, including zero; "1#element" requires at least one; and "1#2element" allows one or two.

; comment

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

implied *LWS

The grammar described by this specification is word-based. Except where noted otherwise, linear white space (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent words and separators, without changing the interpretation of a field. At least one delimiter (LWS and/or separators) MUST exist between any two tokens (for the definition of "token" below), since they would otherwise be interpreted as a single token.

2.2. Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs. The US-ASCII coded character set is defined by ANSI X3.4-1986 [USASCII].

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

HTTP/1.1 defines the sequence CR LF as the end-of-line marker for all protocol elements except the entity-body (see Appendix B for tolerant applications). The end-of-line marker within an entity-body is defined by its associated media type, as described in [Part 3].

```
CRLF           = CR LF
```

HTTP/1.1 header field values can be folded onto multiple lines if the continuation line begins with a space or horizontal tab. All linear white space, including folding, has the same semantics as SP. A recipient MAY replace any linear white space with a single SP before interpreting the field value or forwarding the message downstream.

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

The TEXT rule is only used for descriptive field contents and values that are not intended to be interpreted by the message parser. Words of *TEXT MAY contain characters from character sets other than ISO-8859-1 [ISO-8859] only when encoded according to the rules of RFC 2047 [RFC2047].

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

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

Hexadecimal numeric characters are used in several protocol elements.

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

Many HTTP/1.1 header field values consist of words separated by LWS or special characters. These special characters MUST be in a quoted string to be used within a parameter value (as defined in Section 3.4).

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

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

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

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

```
    quoted-string  = ( <"> *(qdtext | quoted-pair ) <"> )
    qdtext         = <any TEXT except <">>
```

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

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

3. Protocol Parameters

3.1. HTTP Version

HTTP uses a "<major>.<minor>" numbering scheme to indicate versions of the protocol. The protocol versioning policy is intended to allow the sender to indicate the format of a message and its capacity for understanding further HTTP communication, rather than the features obtained via that communication. No change is made to the version number for the addition of message components which do not affect communication behavior or which only add to extensible field values. The <minor> number is incremented when the changes made to the protocol add features which do not change the general message parsing algorithm, but which may add to the message semantics and imply additional capabilities of the sender. The <major> number is incremented when the format of a message within the protocol is changed. See RFC 2145 [RFC2145] for a fuller explanation.

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

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

Note that the major and minor numbers MUST be treated as separate integers and that each MAY be incremented higher than a single digit. Thus, HTTP/2.4 is a lower version than HTTP/2.13, which in turn is lower than HTTP/12.3. Leading zeros MUST be ignored by recipients and MUST NOT be sent.

An application that sends a request or response message that includes HTTP-Version of "HTTP/1.1" MUST be at least conditionally compliant with this specification. Applications that are at least conditionally compliant with this specification SHOULD use an HTTP-Version of "HTTP/1.1" in their messages, and MUST do so for any message that is not compatible with HTTP/1.0. For more details on when to send specific HTTP-Version values, see RFC 2145 [RFC2145].

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

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

version request is received, the proxy/gateway MUST either downgrade the request version, or respond with an error, or switch to tunnel behavior.

Due to interoperability problems with HTTP/1.0 proxies discovered since the publication of RFC 2068 [RFC2068], caching proxies MUST, gateways MAY, and tunnels MUST NOT upgrade the request to the highest version they support. The proxy/gateway's response to that request MUST be in the same major version as the request.

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

3.2. Uniform Resource Identifiers

URIs have been known by many names: WWW addresses, Universal Document Identifiers, Universal Resource Identifiers [RFC1630], and finally the combination of Uniform Resource Locators (URL) [RFC1738] and Names (URN) [RFC1737]. As far as HTTP is concerned, Uniform Resource Identifiers are simply formatted strings which identify--via name, location, or any other characteristic--a resource.

3.2.1. General Syntax

URIs in HTTP can be represented in absolute form or relative to some known base URI [RFC1808], depending upon the context of their use. The two forms are differentiated by the fact that absolute URIs always begin with a scheme name followed by a colon. For definitive information on URL syntax and semantics, see "Uniform Resource Identifiers (URI): Generic Syntax and Semantics," RFC 2396 [RFC2396] (which replaces RFCs 1738 [RFC1738] and RFC 1808 [RFC1808]). This specification adopts the definitions of "URI-reference", "absoluteURI", "relativeURI", "port", "host","abs_path", "rel_path", and "authority" from that specification.

The HTTP protocol does not place any a priori limit on the length of a URI. Servers MUST be able to handle the URI of any resource they serve, and SHOULD be able to handle URIs of unbounded length if they provide GET-based forms that could generate such URIs. A server SHOULD return 414 (Request-URI Too Long) status if a URI is longer than the server can handle (see [Part 2]).

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

3.2.2. http URL

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

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

If the port is empty or not given, port 80 is assumed. The semantics are that the identified resource is located at the server listening for TCP connections on that port of that host, and the Request-URI for the resource is abs_path (Section 5.1.2). The use of IP addresses in URLs SHOULD be avoided whenever possible (see RFC 1900 [RFC1900]). If the abs_path is not present in the URL, it MUST be given as "/" when used as a Request-URI for a resource (Section 5.1.2). If a proxy receives a host name which is not a fully qualified domain name, it MAY add its domain to the host name it received. If a proxy receives a fully qualified domain name, the proxy MUST NOT change the host name.

3.2.3. URI Comparison

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

- A port that is empty or not given is equivalent to the default port for that URI-reference;
- Comparisons of host names MUST be case-insensitive;
- Comparisons of scheme names MUST be case-insensitive;
- An empty abs_path is equivalent to an abs_path of "/".

Characters other than those in the "reserved" set (see RFC 2396 [RFC2396]) are equivalent to their ""%" HEX HEX" encoding.

For example, the following three URIs are equivalent:

```
http://abc.com:80/~smith/home.html
http://ABC.com/%7Esmith/home.html
http://ABC.com:/%7esmith/home.html
```

3.3. Date/Time Formats

3.3.1. Full Date

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

```
Sun, 06 Nov 1994 08:49:37 GMT  ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; obsolete RFC 850 format
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 [RFC1123] (an update to RFC 822 [RFC822]). The other formats are described here only for compatibility with obsolete implementations. HTTP/1.1 clients and servers that parse the date value MUST accept all three formats (for compatibility with HTTP/1.0), though they MUST only generate the RFC 1123 format for representing HTTP-date values in header fields. See Appendix B for further information.

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

All HTTP date/time stamps MUST be represented in Greenwich Mean Time (GMT), without exception. For the purposes of HTTP, GMT is exactly equal to UTC (Coordinated Universal Time). This is indicated in the first two formats by the inclusion of "GMT" as the three-letter abbreviation for time zone, and MUST be assumed when reading the asctime format. HTTP-date is case sensitive and MUST NOT include additional LWS beyond that specifically included as SP in the grammar.

```
HTTP-date    = rfc1123-date | rfc850-date | asctime-date
rfc1123-date = wkday "," SP date1 SP time SP "GMT"
rfc850-date  = weekday "," SP date2 SP time SP "GMT"
asctime-date = wkday SP date3 SP time SP 4DIGIT
date1        = 2DIGIT SP month SP 4DIGIT
               ; day month year (e.g., 02 Jun 1982)
date2        = 2DIGIT "-" month "-" 2DIGIT
               ; day-month-year (e.g., 02-Jun-82)
date3        = month SP ( 2DIGIT | ( SP 1DIGIT ))
               ; month day (e.g., Jun  2)
time         = 2DIGIT ":" 2DIGIT ":" 2DIGIT
               ; 00:00:00 - 23:59:59
wkday        = "Mon" | "Tue" | "Wed"
             | "Thu" | "Fri" | "Sat" | "Sun"
weekday      = "Monday" | "Tuesday" | "Wednesday"
             | "Thursday" | "Friday" | "Saturday" | "Sunday"
month        = "Jan" | "Feb" | "Mar" | "Apr"
             | "May" | "Jun" | "Jul" | "Aug"
             | "Sep" | "Oct" | "Nov" | "Dec"
```

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

3.4. Transfer Codings

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

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

Parameters are in the form of attribute/value pairs.

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

All transfer-coding values are case-insensitive. HTTP/1.1 uses transfer-coding values in the TE header field (Section 8.5) and in the Transfer-Encoding header field (Section 8.7).

Whenever a transfer-coding is applied to a message-body, the set of transfer-codings MUST include "chunked", unless the message is

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

Transfer-codings are analogous to the Content-Transfer-Encoding values of MIME [RFC2045], which were designed to enable safe transport of binary data over a 7-bit transport service. However, safe transport has a different focus for an 8bit-clean transfer protocol. In HTTP, the only unsafe characteristic of message-bodies is the difficulty in determining the exact body length (Section 4.4), or the desire to encrypt data over a shared transport.

The Internet Assigned Numbers Authority (IANA) acts as a registry for transfer-coding value tokens. Initially, the registry contains the following tokens: "chunked" (Section 3.4.1), "gzip" ([Part 3]), "compress" ([Part 3]), and "deflate" ([Part 3]).

New transfer-coding value tokens SHOULD be registered in the same way as new content-coding value tokens ([Part 3]).

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

3.4.1. Chunked Transfer Coding

The chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator, followed by an OPTIONAL trailer containing entity-header fields. This allows dynamically produced content to be transferred along with the information necessary for the recipient to verify that it has received the full message.

```
Chunked-Body   = *chunk
                 last-chunk
                 trailer
                 CRLF

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

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

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

The trailer allows the sender to include additional HTTP header fields at the end of the message. The Trailer header field can be used to indicate which header fields are included in a trailer (see Section 8.6).

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

1. the request included a TE header field that indicates "trailers" is acceptable in the transfer-coding of the response, as described in Section 8.5; or,

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

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

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

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

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

4. HTTP Message

4.1. Message Types

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

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

Request (Section 5) and Response (Section 6) messages use the generic message format of RFC 822 [RFC822] for transferring entities (the payload of the message). Both types of message consist of a start-line, zero or more header fields (also known as "headers"), an empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields, and possibly a message-body.

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

In the interest of robustness, servers SHOULD ignore any empty line(s) received where a Request-Line is expected. In other words,

if the server is reading the protocol stream at the beginning of a message and receives a CRLF first, it should ignore the CRLF.

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

4.2. Message Headers

HTTP header fields, which include general-header (Section 4.5), request-header ([Part 2]), response-header ([Part 2]), and entity-header ([Part 3]) fields, follow the same generic format as that given in Section 3.1 of RFC 822 [RFC822]. Each header field consists of a name followed by a colon (":") and the field value. Field names are case-insensitive. The field value MAY be preceded by any amount of LWS, though a single SP is preferred. Header fields can be extended over multiple lines by preceding each extra line with at least one SP or HT. Applications ought to follow "common form", where one is known or indicated, when generating HTTP constructs, since there might exist some implementations that fail to accept anything beyond the common forms.

```
message-header = field-name ":" [ field-value ]
field-name     = token
field-value    = *( field-content | LWS )
field-content  = <the OCTETs making up the field-value
                 and consisting of either *TEXT or combinations
                 of token, separators, and quoted-string>
```

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

The order in which header fields with differing field names are received is not significant. However, it is "good practice" to send general-header fields first, followed by request-header or response-header fields, and ending with the entity-header fields.

Multiple message-header fields with the same field-name MAY be present in a message if and only if the entire field-value for that header field is defined as a comma-separated list [i.e., #(values)]. It MUST be possible to combine the multiple header fields into one

"field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field-value to the first, each separated by a comma. The order in which header fields with the same field-name are received is therefore significant to the interpretation of the combined field value, and thus a proxy MUST NOT change the order of these field values when a message is forwarded.

4.3. Message Body

The message-body (if any) of an HTTP message is used to carry the entity-body associated with the request or response. The message-body differs from the entity-body only when a transfer-coding has been applied, as indicated by the Transfer-Encoding header field (Section 8.7).

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

Transfer-Encoding MUST be used to indicate any transfer-codings applied by an application to ensure safe and proper transfer of the message. Transfer-Encoding is a property of the message, not of the entity, and thus MAY be added or removed by any application along the request/response chain. (However, Section 3.4 places restrictions on when certain transfer-codings may be used.)

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

The presence of a message-body in a request is signaled by the inclusion of a Content-Length or Transfer-Encoding header field in the request's message-headers. A message-body MUST NOT be included in a request if the specification of the request method ([Part 2]) does not allow sending an entity-body in requests. A server SHOULD read and forward a message-body on any request; if the request method does not include defined semantics for an entity-body, then the message-body SHOULD be ignored when handling the request.

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

4.4. Message Length

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

1. Any response message which "MUST NOT" include a message-body (such as the 1xx, 204, and 304 responses and any response to a HEAD request) is always terminated by the first empty line after the header fields, regardless of the entity-header fields present in the message.

2. If a Transfer-Encoding header field (Section 8.7) is present, then the transfer-length is defined by use of the "chunked" transfer-coding (Section 3.4), unless the message is terminated by closing the connection.

3. If a Content-Length header field (Section 8.2) is present, its decimal value in OCTETs represents both the entity-length and the transfer-length. The Content-Length header field MUST NOT be sent if these two lengths are different (i.e., if a Transfer-Encoding header field is present). If a message is received with both a Transfer-Encoding header field and a Content-Length header field, the latter MUST be ignored.

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

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

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

For compatibility with HTTP/1.0 applications, HTTP/1.1 requests containing a message-body MUST include a valid Content-Length header field unless the server is known to be HTTP/1.1 compliant. If a

request contains a message-body and a Content-Length is not given, the server SHOULD respond with 400 (bad request) if it cannot determine the length of the message, or with 411 (length required) if it wishes to insist on receiving a valid Content-Length.

All HTTP/1.1 applications that receive entities MUST accept the "chunked" transfer-coding (Section 3.4), thus allowing this mechanism to be used for messages when the message length cannot be determined in advance.

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

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

4.5. General Header Fields

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

```
    general-header = Cache-Control            ; [Part 6]
                   | Connection               ; Section 8.1
                   | Date                     ; Section 8.3
                   | Pragma                   ; [Part 6]
                   | Trailer                  ; Section 8.6
                   | Transfer-Encoding        ; Section 8.7
                   | Upgrade                  ; Section 8.8
                   | Via                      ; Section 8.9
                   | Warning                  ; [Part 6]
```

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

5. Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource,

the identifier of the resource, and the protocol version in use.

```
Request       = Request-Line              ; Section 5.1
                *(( general-header        ; Section 4.5
                 | request-header         ; [Part 2]
                 | entity-header ) CRLF)  ; [Part 3]
                CRLF
                [ message-body ]          ; Section 4.3
```

5.1. Request-Line

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

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

5.1.1. Method

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

```
Method         = token
```

5.1.2. Request-URI

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

```
Request-URI    = "*"
               | absoluteURI
               | ( abs_path [ "?" query ] )
               | authority
```

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

```
OPTIONS * HTTP/1.1
```

The absoluteURI form is REQUIRED when the request is being made to a proxy. The proxy is requested to forward the request or service it from a valid cache, and return the response. Note that the proxy MAY forward the request on to another proxy or directly to the server specified by the absoluteURI. In order to avoid request loops, a

proxy MUST be able to recognize all of its server names, including any aliases, local variations, and the numeric IP address. An example Request-Line would be:

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

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

The authority form is only used by the CONNECT method ([Part 2]).

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

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

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

The Request-URI is transmitted in the format specified in Section 3.2.1. If the Request-URI is encoded using the "% HEX HEX" encoding [RFC2396], the origin server MUST decode the Request-URI in order to properly interpret the request. Servers SHOULD respond to invalid Request-URIs with an appropriate status code.

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

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

5.2. The Resource Identified by a Request

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

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

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

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

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

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

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

6. Response

After receiving and interpreting a request message, a server responds with an HTTP response message.

```
    Response      = Status-Line               ; Section 6.1
                    *(( general-header        ; Section 4.5
                     | response-header        ; [Part 2]
                     | entity-header ) CRLF)  ; [Part 3]
                    CRLF
                    [ message-body ]          ; Section 4.3
```

6.1. Status-Line

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

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

6.1.1. Status Code and Reason Phrase

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. These codes are fully defined in [Part 2]. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine or display the Reason-Phrase.

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

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

```
Status-Code    = 3DIGIT
Reason-Phrase  = *<TEXT, excluding CR, LF>
```

7. Connections

7.1. Persistent Connections

7.1.1. Purpose

Prior to persistent connections, a separate TCP connection was established to fetch each URL, increasing the load on HTTP servers and causing congestion on the Internet. The use of inline images and other associated data often require a client to make multiple requests of the same server in a short amount of time. Analysis of these performance problems and results from a prototype implementation are available [Pad1995] [Spe]. Implementation experience and measurements of actual HTTP/1.1 (RFC 2068) implementations show good results [Nie1997]. Alternatives have also been explored, for example, T/TCP [Tou1998].

Persistent HTTP connections have a number of advantages:

- By opening and closing fewer TCP connections, CPU time is saved in routers and hosts (clients, servers, proxies, gateways, tunnels, or caches), and memory used for TCP protocol control blocks can be saved in hosts.

- HTTP requests and responses can be pipelined on a connection. Pipelining allows a client to make multiple requests without waiting for each response, allowing a single TCP connection to be used much more efficiently, with much lower elapsed time.

- Network congestion is reduced by reducing the number of packets caused by TCP opens, and by allowing TCP sufficient time to determine the congestion state of the network.

- Latency on subsequent requests is reduced since there is no time spent in TCP's connection opening handshake.

- HTTP can evolve more gracefully, since errors can be reported without the penalty of closing the TCP connection. Clients using future versions of HTTP might optimistically try a new feature, but if communicating with an older server, retry with old semantics after an error is reported.

HTTP implementations SHOULD implement persistent connections.

7.1.2. Overall Operation

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

even after error responses from the server.

Persistent connections provide a mechanism by which a client and a server can signal the close of a TCP connection. This signaling takes place using the Connection header field (Section 8.1). Once a close has been signaled, the client MUST NOT send any more requests on that connection.

7.1.2.1. Negotiation

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

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

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

Clients and servers SHOULD NOT assume that a persistent connection is maintained for HTTP versions less than 1.1 unless it is explicitly signaled. See Appendix D.2 for more information on backward compatibility with HTTP/1.0 clients.

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

7.1.2.2. Pipelining

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

Clients which assume persistent connections and pipeline immediately after connection establishment SHOULD be prepared to retry their connection if the first pipelined attempt fails. If a client does such a retry, it MUST NOT pipeline before it knows the connection is

persistent. Clients MUST also be prepared to resend their requests if the server closes the connection before sending all of the corresponding responses.

Clients SHOULD NOT pipeline requests using non-idempotent methods or non-idempotent sequences of methods (see [Part 2]). Otherwise, a premature termination of the transport connection could lead to indeterminate results. A client wishing to send a non-idempotent request SHOULD wait to send that request until it has received the response status for the previous request.

7.1.3. Proxy Servers

It is especially important that proxies correctly implement the properties of the Connection header field as specified in Section 8.1.

The proxy server MUST signal persistent connections separately with its clients and the origin servers (or other proxy servers) that it connects to. Each persistent connection applies to only one transport link.

A proxy server MUST NOT establish a HTTP/1.1 persistent connection with an HTTP/1.0 client (but see RFC 2068 [RFC2068] for information and discussion of the problems with the Keep-Alive header implemented by many HTTP/1.0 clients).

7.1.4. Practical Considerations

Servers will usually have some time-out value beyond which they will no longer maintain an inactive connection. Proxy servers might make this a higher value since it is likely that the client will be making more connections through the same server. The use of persistent connections places no requirements on the length (or existence) of this time-out for either the client or the server.

When a client or server wishes to time-out it SHOULD issue a graceful close on the transport connection. Clients and servers SHOULD both constantly watch for the other side of the transport close, and respond to it as appropriate. If a client or server does not detect the other side's close promptly it could cause unnecessary resource drain on the network.

A client, server, or proxy MAY close the transport connection at any time. For example, a client might have started to send a new request at the same time that the server has decided to close the "idle" connection. From the server's point of view, the connection is being closed while it was idle, but from the client's point of view, a

request is in progress.

This means that clients, servers, and proxies MUST be able to recover from asynchronous close events. Client software SHOULD reopen the transport connection and retransmit the aborted sequence of requests without user interaction so long as the request sequence is idempotent (see [Part 2]). Non-idempotent methods or sequences MUST NOT be automatically retried, although user agents MAY offer a human operator the choice of retrying the request(s). Confirmation by user-agent software with semantic understanding of the application MAY substitute for user confirmation. The automatic retry SHOULD NOT be repeated if the second sequence of requests fails.

Servers SHOULD always respond to at least one request per connection, if at all possible. Servers SHOULD NOT close a connection in the middle of transmitting a response, unless a network or client failure is suspected.

Clients that use persistent connections SHOULD limit the number of simultaneous connections that they maintain to a given server. A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy. A proxy SHOULD use up to 2*N connections to another server or proxy, where N is the number of simultaneously active users. These guidelines are intended to improve HTTP response times and avoid congestion.

7.2. Message Transmission Requirements

7.2.1. Persistent Connections and Flow Control

HTTP/1.1 servers SHOULD maintain persistent connections and use TCP's flow control mechanisms to resolve temporary overloads, rather than terminating connections with the expectation that clients will retry. The latter technique can exacerbate network congestion.

7.2.2. Monitoring Connections for Error Status Messages

An HTTP/1.1 (or later) client sending a message-body SHOULD monitor the network connection for an error status while it is transmitting the request. If the client sees an error status, it SHOULD immediately cease transmitting the body. If the body is being sent using a "chunked" encoding (Section 3.4), a zero length chunk and empty trailer MAY be used to prematurely mark the end of the message. If the body was preceded by a Content-Length header, the client MUST close the connection.

7.2.3. Use of the 100 (Continue) Status

The purpose of the 100 (Continue) status (see [Part 2]) is to allow a client that is sending a request message with a request body to determine if the origin server is willing to accept the request (based on the request headers) before the client sends the request body. In some cases, it might either be inappropriate or highly inefficient for the client to send the body if the server will reject the message without looking at the body.

Requirements for HTTP/1.1 clients:

- If a client will wait for a 100 (Continue) response before sending the request body, it MUST send an Expect request-header field ([Part 2]) with the "100-continue" expectation.

- A client MUST NOT send an Expect request-header field ([Part 2]) with the "100-continue" expectation if it does not intend to send a request body.

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

Requirements for HTTP/1.1 origin servers:

- Upon receiving a request which includes an Expect request-header field with the "100-continue" expectation, an origin server MUST either respond with 100 (Continue) status and continue to read from the input stream, or respond with a final status code. The origin server MUST NOT wait for the request body before sending the 100 (Continue) response. If it responds with a final status code, it MAY close the transport connection or it MAY continue to read and discard the rest of the request. It MUST NOT perform the requested method if it returns a final status code.

- An origin server SHOULD NOT send a 100 (Continue) response if the request message does not include an Expect request-header field with the "100-continue" expectation, and MUST NOT send a 100 (Continue) response if such a request comes from an HTTP/1.0 (or earlier) client. There is an exception to this rule: for compatibility with RFC 2068, a server MAY send a 100 (Continue) status in response to an HTTP/1.1 PUT or POST request that does not include an Expect request-header field with the "100-continue"

expectation. This exception, the purpose of which is to minimize any client processing delays associated with an undeclared wait for 100 (Continue) status, applies only to HTTP/1.1 requests, and not to requests with any other HTTP-version value.

- An origin server MAY omit a 100 (Continue) response if it has already received some or all of the request body for the corresponding request.

- An origin server that sends a 100 (Continue) response MUST ultimately send a final status code, once the request body is received and processed, unless it terminates the transport connection prematurely.

- If an origin server receives a request that does not include an Expect request-header field with the "100-continue" expectation, the request includes a request body, and the server responds with a final status code before reading the entire request body from the transport connection, then the server SHOULD NOT close the transport connection until it has read the entire request, or until the client closes the connection. Otherwise, the client might not reliably receive the response message. However, this requirement is not be construed as preventing a server from defending itself against denial-of-service attacks, or from badly broken client implementations.

Requirements for HTTP/1.1 proxies:

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

- If the proxy knows that the version of the next-hop server is HTTP/1.0 or lower, it MUST NOT forward the request, and it MUST respond with a 417 (Expectation Failed) status.

- Proxies SHOULD maintain a cache recording the HTTP version numbers received from recently-referenced next-hop servers.

- A proxy MUST NOT forward a 100 (Continue) response if the request message was received from an HTTP/1.0 (or earlier) client and did not include an Expect request-header field with the "100-continue" expectation. This requirement overrides the general rule for forwarding of 1xx responses (see [Part 2]).

7.2.4. Client Behavior if Server Prematurely Closes Connection

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

1. Initiate a new connection to the server

2. Transmit the request-headers

3. Initialize a variable R to the estimated round-trip time to the server (e.g., based on the time it took to establish the connection), or to a constant value of 5 seconds if the round-trip time is not available.

4. Compute T = R * (2**N), where N is the number of previous retries of this request.

5. Wait either for an error response from the server, or for T seconds (whichever comes first)

6. If no error response is received, after T seconds transmit the body of the request.

7. If client sees that the connection is closed prematurely, repeat from step 1 until the request is accepted, an error response is received, or the user becomes impatient and terminates the retry process.

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

- SHOULD NOT continue and

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

8. Header Field Definitions

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

8.1. Connection

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

The Connection header has the following grammar:

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

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

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

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

```
Connection: close
```

in either the request or the response header fields indicates that the connection SHOULD NOT be considered `persistent' (Section 7.1) after the current request/response is complete.

An HTTP/1.1 client that does not support persistent connections MUST include the "close" connection option in every request message.

An HTTP/1.1 server that does not support persistent connections MUST include the "close" connection option in every response message that does not have a 1xx (informational) status code.

A system receiving an HTTP/1.0 (or lower-version) message that includes a Connection header MUST, for each connection-token in this field, remove and ignore any header field(s) from the message with the same name as the connection-token. This protects against mistaken forwarding of such header fields by pre-HTTP/1.1 proxies. See Appendix D.2.

8.2. Content-Length

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

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

An example is

```
Content-Length: 3495
```

Applications SHOULD use this field to indicate the transfer-length of the message-body, unless this is prohibited by the rules in Section 4.4.

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

Note that the meaning of this field is significantly different from the corresponding definition in MIME, where it is an optional field used within the "message/external-body" content-type. In HTTP, it SHOULD be sent whenever the message's length can be determined prior to being transferred, unless this is prohibited by the rules in Section 4.4.

8.3. Date

The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822. The field value is an HTTP-date, as described in Section 3.3.1; it MUST be sent in rfc1123-date format.

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

An example is

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

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

1. If the response status code is 100 (Continue) or 101 (Switching Protocols), the response MAY include a Date header field, at the server's option.

2. If the response status code conveys a server error, e.g. 500 (Internal Server Error) or 503 (Service Unavailable), and it is inconvenient or impossible to generate a valid Date.

3. If the server does not have a clock that can provide a reasonable approximation of the current time, its responses MUST NOT include a Date header field. In this case, the rules in Section 8.3.1 MUST be followed.

A received message that does not have a Date header field MUST be assigned one by the recipient if the message will be cached by that recipient or gatewayed via a protocol which requires a Date. An HTTP implementation without a clock MUST NOT cache responses without revalidating them on every use. An HTTP cache, especially a shared cache, SHOULD use a mechanism, such as NTP [RFC1305], to synchronize its clock with a reliable external standard.

Clients SHOULD only send a Date header field in messages that include an entity-body, as in the case of the PUT and POST requests, and even then it is optional. A client without a clock MUST NOT send a Date header field in a request.

The HTTP-date sent in a Date header SHOULD NOT represent a date and time subsequent to the generation of the message. It SHOULD represent the best available approximation of the date and time of message generation, unless the implementation has no means of generating a reasonably accurate date and time. In theory, the date ought to represent the moment just before the entity is generated. In practice, the date can be generated at any time during the message origination without affecting its semantic value.

8.3.1. Clockless Origin Server Operation

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

8.4. Host

The Host request-header field specifies the Internet host and port number of the resource being requested, as obtained from the original URI given by the user or referring resource (generally an HTTP URL, as described in Section 3.2.2). The Host field value MUST represent

the naming authority of the origin server or gateway given by the original URL. This allows the origin server or gateway to differentiate between internally-ambiguous URLs, such as the root "/" URL of a server for multiple host names on a single IP address.

```
Host = "Host" ":" host [ ":" port ] ; Section 3.2.2
```

A "host" without any trailing port information implies the default port for the service requested (e.g., "80" for an HTTP URL). For example, a request on the origin server for <http://www.w3.org/pub/WWW/> would properly include:

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

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

See sections 5.2 and D.1.1 for other requirements relating to Host.

8.5. TE

The TE request-header field indicates what extension transfer-codings it is willing to accept in the response and whether or not it is willing to accept trailer fields in a chunked transfer-coding. Its value may consist of the keyword "trailers" and/or a comma-separated list of extension transfer-coding names with optional accept parameters (as described in Section 3.4).

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

The presence of the keyword "trailers" indicates that the client is willing to accept trailer fields in a chunked transfer-coding, as defined in Section 3.4.1. This keyword is reserved for use with transfer-coding values even though it does not itself represent a transfer-coding.

Examples of its use are:

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

The TE header field only applies to the immediate connection. Therefore, the keyword MUST be supplied within a Connection header field (Section 8.1) whenever TE is present in an HTTP/1.1 message.

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

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

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

2. If the transfer-coding being tested is one of the transfer-codings listed in the TE field, then it is acceptable unless it is accompanied by a qvalue of 0. (As defined in [Part 3], a qvalue of 0 means "not acceptable.")

3. If multiple transfer-codings are acceptable, then the acceptable transfer-coding with the highest non-zero qvalue is preferred. The "chunked" transfer-coding always has a qvalue of 1.

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

8.6. Trailer

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

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

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

in the trailer.

If no Trailer header field is present, the trailer SHOULD NOT include any header fields. See Section 3.4.1 for restrictions on the use of trailer fields in a "chunked" transfer-coding.

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

- Transfer-Encoding
- Content-Length
- Trailer

8.7. Transfer-Encoding

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

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

Transfer-codings are defined in Section 3.4. An example is:

```
  Transfer-Encoding: chunked
```

If multiple encodings have been applied to an entity, the transfer-codings MUST be listed in the order in which they were applied. Additional information about the encoding parameters MAY be provided by other entity-header fields not defined by this specification.

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

8.8. Upgrade

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

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

For example,

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

The Upgrade header field is intended to provide a simple mechanism for transition from HTTP/1.1 to some other, incompatible protocol. It does so by allowing the client to advertise its desire to use another protocol, such as a later version of HTTP with a higher major version number, even though the current request has been made using HTTP/1.1. This eases the difficult transition between incompatible protocols by allowing the client to initiate a request in the more commonly supported protocol while indicating to the server that it would like to use a "better" protocol if available (where "better" is determined by the server, possibly according to the nature of the method and/or resource being requested).

The Upgrade header field only applies to switching application-layer protocols upon the existing transport-layer connection. Upgrade cannot be used to insist on a protocol change; its acceptance and use by the server is optional. The capabilities and nature of the application-layer communication after the protocol change is entirely dependent upon the new protocol chosen, although the first action after changing the protocol MUST be a response to the initial HTTP request containing the Upgrade header field.

The Upgrade header field only applies to the immediate connection. Therefore, the upgrade keyword MUST be supplied within a Connection header field (Section 8.1) whenever Upgrade is present in an HTTP/1.1 message.

The Upgrade header field cannot be used to indicate a switch to a protocol on a different connection. For that purpose, it is more appropriate to use a 301, 302, 303, or 305 redirection response.

This specification only defines the protocol name "HTTP" for use by the family of Hypertext Transfer Protocols, as defined by the HTTP version rules of Section 3.1 and future updates to this specification. Any token can be used as a protocol name; however, it will only be useful if both the client and server associate the name with the same protocol.

8.9. Via

The Via general-header field MUST be used by gateways and proxies to indicate the intermediate protocols and recipients between the user agent and the server on requests, and between the origin server and the client on responses. It is analogous to the "Received" field of RFC 822 [RFC822] and is intended to be used for tracking message forwards, avoiding request loops, and identifying the protocol capabilities of all senders along the request/response chain.

```
Via =  "Via" ":" 1#( received-protocol received-by [ comment ] )
received-protocol = [ protocol-name "/" ] protocol-version
protocol-name     = token
protocol-version  = token
received-by       = ( host [ ":" port ] ) | pseudonym
pseudonym         = token
```

The received-protocol indicates the protocol version of the message received by the server or client along each segment of the request/response chain. The received-protocol version is appended to the Via field value when the message is forwarded so that information about the protocol capabilities of upstream applications remains visible to all recipients.

The protocol-name is optional if and only if it would be "HTTP". The received-by field is normally the host and optional port number of a recipient server or client that subsequently forwarded the message. However, if the real host is considered to be sensitive information, it MAY be replaced by a pseudonym. If the port is not given, it MAY be assumed to be the default port of the received-protocol.

Multiple Via field values represents each proxy or gateway that has forwarded the message. Each recipient MUST append its information such that the end result is ordered according to the sequence of forwarding applications.

Comments MAY be used in the Via header field to identify the software of the recipient proxy or gateway, analogous to the User-Agent and Server header fields. However, all comments in the Via field are optional and MAY be removed by any recipient prior to forwarding the message.

For example, a request message could be sent from an HTTP/1.0 user agent to an internal proxy code-named "fred", which uses HTTP/1.1 to forward the request to a public proxy at nowhere.com, which completes the request by forwarding it to the origin server at www.ics.uci.edu. The request received by www.ics.uci.edu would then have the following Via header field:

```
Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)
```

Proxies and gateways used as a portal through a network firewall SHOULD NOT, by default, forward the names and ports of hosts within the firewall region. This information SHOULD only be propagated if explicitly enabled. If not enabled, the received-by host of any host behind the firewall SHOULD be replaced by an appropriate pseudonym for that host.

For organizations that have strong privacy requirements for hiding internal structures, a proxy MAY combine an ordered subsequence of Via header field entries with identical received-protocol values into a single such entry. For example,

```
Via: 1.0 ricky, 1.1 ethel, 1.1 fred, 1.0 lucy
```

could be collapsed to

```
Via: 1.0 ricky, 1.1 mertz, 1.0 lucy
```

Applications SHOULD NOT combine multiple entries unless they are all under the same organizational control and the hosts have already been replaced by pseudonyms. Applications MUST NOT combine entries which have different received-protocol values.

9. IANA Considerations

TBD.

10. Security Considerations

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

10.1. Personal Information

HTTP clients are often privy to large amounts of personal information (e.g. the user's name, location, mail address, passwords, encryption keys, etc.), and SHOULD be very careful to prevent unintentional leakage of this information via the HTTP protocol to other sources. We very strongly recommend that a convenient interface be provided for the user to control dissemination of such information, and that designers and implementors be particularly careful in this area. History shows that errors in this area often create serious security and/or privacy problems and generate highly adverse publicity for the implementor's company.

10.2. Abuse of Server Log Information

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

handling can be constrained by law in certain countries. People using the HTTP protocol to provide data are responsible for ensuring that such material is not distributed without the permission of any individuals that are identifiable by the published results.

10.3. Attacks Based On File and Path Names

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

10.4. DNS Spoofing

Clients using HTTP rely heavily on the Domain Name Service, and are thus generally prone to security attacks based on the deliberate mis-association of IP addresses and DNS names. Clients need to be cautious in assuming the continuing validity of an IP number/DNS name association.

In particular, HTTP clients SHOULD rely on their name resolver for confirmation of an IP number/DNS name association, rather than caching the result of previous host name lookups. Many platforms already can cache host name lookups locally when appropriate, and they SHOULD be configured to do so. It is proper for these lookups to be cached, however, only when the TTL (Time To Live) information reported by the name server makes it likely that the cached information will remain useful.

If HTTP clients cache the results of host name lookups in order to achieve a performance improvement, they MUST observe the TTL information reported by DNS.

If HTTP clients do not observe this rule, they could be spoofed when a previously-accessed server's IP address changes. As network renumbering is expected to become increasingly common [RFC1900], the

possibility of this form of attack will grow. Observing this requirement thus reduces this potential security vulnerability.

This requirement also improves the load-balancing behavior of clients for replicated servers using the same DNS name and reduces the likelihood of a user's experiencing failure in accessing sites which use that strategy.

10.5. Proxies and Caching

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

Proxy operators should protect the systems on which proxies run as they would protect any system that contains or transports sensitive information. In particular, log information gathered at proxies often contains highly sensitive personal information, and/or information about organizations. Log information should be carefully guarded, and appropriate guidelines for use developed and followed. (Section 10.2).

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

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

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

10.6. Denial of Service Attacks on Proxies

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

11. Acknowledgments

This specification makes heavy use of the augmented BNF and generic constructs defined by David H. Crocker for RFC 822 [RFC822]. Similarly, it reuses many of the definitions provided by Nathaniel Borenstein and Ned Freed for MIME [RFC2045]. We hope that their inclusion in this specification will help reduce past confusion over the relationship between HTTP and Internet mail message formats.

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

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

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

Based on an XML translation of RFC 2616 by Julian Reschke.

12. References

[ISO-8859] International Organization for Standardization, "Information technology - 8-bit single byte coded graphic - character sets", 1987-1990.

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.

[Nie1997] Nielsen, H., Gettys, J., Prud'hommeaux, E., Lie, H., and C. Lilley, "Network Performance Effects of HTTP/1.1, CSS1,

and PNG", Proceedings of ACM SIGCOMM '97, Cannes France , Sep 1997.

[Pad1995] Padmanabhan, V. and J. Mogul, "Improving HTTP Latency", Computer Networks and ISDN Systems v. 28, pp. 25-35, Dec 1995.

Slightly revised version of paper in Proc. 2nd International WWW Conference '94: Mosaic and the Web, Oct. 1994, which is available at <http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html>.

[RFC1123] Braden, R., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, October 1989.

[RFC1305] Mills, D., "Network Time Protocol (Version 3) Specification, Implementation", RFC 1305, March 1992.

[RFC1436] Anklesaria, F., McCahill, M., Lindner, P., Johnson, D., Torrey, D., and B. Alberti, "The Internet Gopher Protocol (a distributed document search and retrieval protocol)", RFC 1436, March 1993.

[RFC1630] Berners-Lee, T., "Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web", RFC 1630, June 1994.

[RFC1700] Reynolds, J. and J. Postel, "Assigned Numbers", STD 2, RFC 1700, October 1994.

[RFC1737] Masinter, L. and K. Sollins, "Functional Requirements for Uniform Resource Names", RFC 1737, December 1994.

[RFC1738] Berners-Lee, T., Masinter, L., and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, December 1994.

[RFC1808] Fielding, R., "Relative Uniform Resource Locators", RFC 1808, June 1995.

[RFC1900] Carpenter, B. and Y. Rekhter, "Renumbering Needs Work", RFC 1900, February 1996.

[RFC1945] Berners-Lee, T., Fielding, R., and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message

Bodies", RFC 2045, November 1996.

[RFC2047] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, November 1996.

[RFC2068] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, January 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2145] Mogul, J., Fielding, R., Gettys, J., and H. Nielsen, "Use and Interpretation of HTTP Version Numbers", RFC 2145, May 1997.

[RFC2324] Masinter, L., "Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)", RFC 2324, April 1998.

[RFC2396] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[RFC3977] Feather, C., "Network News Transfer Protocol (NNTP)", October 2006.

[RFC4288] Freed, N. and J. Klensin, "Media Type Specifications and Registration Procedures", BCP 13, RFC 4288, December 2005.

[RFC821] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC 821, August 1982.

[RFC822] Crocker, D., "Standard for the format of ARPA Internet text messages", STD 11, RFC 822, August 1982.

[RFC959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, October 1985.

[Spe] Spero, S., "Analysis of HTTP Performance Problems", <http://sunsite.unc.edu/mdma-release/http-prob.html>.

[Tou1998] Touch, J., Heidemann, J., and K. Obraczka, "Analysis of HTTP Performance", ISI Research Report ISI/RR-98-463 (original report dated Aug.1996), Aug 1998, <http://www.isi.edu/touch/pubs/http-perf96/>.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

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

Appendix A. Internet Media Type message/http and application/http

In addition to defining the HTTP/1.1 protocol, this document serves as the specification for the Internet media type "message/http" and "application/http". The message/http type can be used to enclose a single HTTP request or response message, provided that it obeys the MIME restrictions for all "message" types regarding line length and encodings. The application/http type can be used to enclose a pipeline of one or more HTTP request or response messages (not intermixed). The following is to be registered with IANA [RFC4288].

Media Type name: message

Media subtype name: http

Required parameters: none

Optional parameters: version, msgtype

version: The HTTP-Version number of the enclosed message (e.g., "1.1"). If not present, the version can be determined from the first line of the body.

msgtype: The message type -- "request" or "response". If not present, the type can be determined from the first line of the body.

Encoding considerations: only "7bit", "8bit", or "binary" are permitted

Security considerations: none

Media Type name: application

Media subtype name: http

Required parameters: none

Optional parameters: version, msgtype

version: The HTTP-Version number of the enclosed messages (e.g., "1.1"). If not present, the version can be determined from the first line of the body.

msgtype: The message type -- "request" or "response". If not present, the type can be determined from the first line of the body.

Encoding considerations: HTTP messages enclosed by this type are in "binary" format; use of an appropriate Content-Transfer-Encoding is required when transmitted via E-mail.

Security considerations: none

Appendix B. Tolerant Applications

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

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

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

The character set of an entity-body SHOULD be labeled as the lowest common denominator of the character codes used within that body, with the exception that not labeling the entity is preferred over labeling

the entity with the labels US-ASCII or ISO-8859-1. See [Part 3].

Additional rules for requirements on parsing and encoding of dates and other potential problems with date encodings include:

- HTTP/1.1 clients and caches SHOULD assume that an RFC-850 date which appears to be more than 50 years in the future is in fact in the past (this helps solve the "year 2000" problem).

- An HTTP/1.1 implementation MAY internally represent a parsed Expires date as earlier than the proper value, but MUST NOT internally represent a parsed Expires date as later than the proper value.

- All expiration-related calculations MUST be done in GMT. The local time zone MUST NOT influence the calculation or comparison of an age or expiration time.

- If an HTTP header incorrectly carries a date value with a time zone other than GMT, it MUST be converted into GMT using the most conservative possible conversion.

Appendix C. Conversion of Date Formats

HTTP/1.1 uses a restricted set of date formats (Section 3.3.1) to simplify the process of date comparison. Proxies and gateways from other protocols SHOULD ensure that any Date header field present in a message conforms to one of the HTTP/1.1 formats and rewrite the date if necessary.

Appendix D. Compatibility with Previous Versions

It is beyond the scope of a protocol specification to mandate compliance with previous versions. HTTP/1.1 was deliberately designed, however, to make supporting previous versions easy. It is worth noting that, at the time of composing this specification (1996), we would expect commercial HTTP/1.1 servers to:

- recognize the format of the Request-Line for HTTP/0.9, 1.0, and 1.1 requests;

- understand any valid request in the format of HTTP/0.9, 1.0, or 1.1;

- respond appropriately with a message in the same major version used by the client.

And we would expect HTTP/1.1 clients to:

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

For most implementations of HTTP/1.0, each connection is established by the client prior to the request and closed by the server after sending the response. Some implementations implement the Keep-Alive version of persistent connections described in Section 19.7.1 of RFC 2068 [RFC2068].

D.1. Changes from HTTP/1.0

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

D.1.1. Changes to Simplify Multi-homed Web Servers and Conserve IP Addresses

The requirements that clients and servers support the Host request-header, report an error if the Host request-header (Section 8.4) is missing from an HTTP/1.1 request, and accept absolute URIs (Section 5.1.2) are among the most important changes defined by this specification.

Older HTTP/1.0 clients assumed a one-to-one relationship of IP addresses and servers; there was no other established mechanism for distinguishing the intended server of a request than the IP address to which that request was directed. The changes outlined above will allow the Internet, once older HTTP clients are no longer common, to support multiple Web sites from a single IP address, greatly simplifying large operational Web servers, where allocation of many IP addresses to a single host has created serious problems. The Internet will also be able to recover the IP addresses that have been allocated for the sole purpose of allowing special-purpose domain names to be used in root-level HTTP URLs. Given the rate of growth of the Web, and the number of servers already deployed, it is extremely important that all implementations of HTTP (including updates to existing HTTP/1.0 applications) correctly implement these requirements:

- Both clients and servers MUST support the Host request-header.
- A client that sends an HTTP/1.1 request MUST send a Host header.

- Servers MUST report a 400 (Bad Request) error if an HTTP/1.1 request does not include a Host request-header.

- Servers MUST accept absolute URIs.

D.2. Compatibility with HTTP/1.0 Persistent Connections

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

However, talking to proxies is the most important use of persistent connections, so that prohibition is clearly unacceptable. Therefore, we need some other mechanism for indicating a persistent connection is desired, which is safe to use even when talking to an old proxy that ignores Connection. Persistent connections are the default for HTTP/1.1 messages; we introduce a new keyword (Connection: close) for declaring non-persistence. See Section 8.1.

The original HTTP/1.0 form of persistent connections (the Connection: Keep-Alive and Keep-Alive header) is documented in RFC 2068. [RFC2068]

D.3. Changes from RFC 2068

This specification has been carefully audited to correct and disambiguate key word usage; RFC 2068 had many problems in respect to the conventions laid out in RFC 2119 [RFC2119].

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

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

Proxies should be able to add Content-Length when appropriate.

Transfer-coding had significant problems, particularly with interactions with chunked encoding. The solution is that transfer-codings become as full fledged as content-codings. This involves adding an IANA registry for transfer-codings (separate from content codings), a new header field (TE) and enabling trailer headers in the future. Transfer encoding is a major performance benefit, so it was worth fixing [Nie1997]. TE also solves another, obscure, downward interoperability problem that could have occurred due to interactions between authentication trailers, chunked encoding and HTTP/1.0 clients.(Section 3.4, 3.4.1, and 8.5)

Index

Authors' Addresses

Roy T. Fielding (editor)
Day Software
23 Corporate Plaza DR, Suite 280
Newport Beach, CA 92660
USA

Phone: +1-949-706-5300
Fax: +1-949-706-5305
Email: fielding@gbiv.com
URI: http://roy.gbiv.com/

James Gettys
Hewlett-Packard Company
HP Labs, Cambridge Research Laboratory
One Cambridge Center
Cambridge, MA 02138
USA

Email: Jim.Gettys@hp.com

Jeffrey C. Mogul
Hewlett-Packard Company
HP Labs, Large Scale Systems Group
1501 Page Mill Road, MS 1177
Palo Alto, CA 94304
USA

Email: JeffMogul@acm.org

Henrik Frystyk Nielsen
Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052
USA

Email: henrikn@microsoft.com

Larry Masinter
Adobe Systems, Incorporated
345 Park Ave
San Jose, CA 95110
USA

Email: LMM@acm.org
URI: http://larry.masinter.net/

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

Email: paulle@microsoft.com

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

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

Full Copyright Statement

Intellectual Property

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).