Network Working Group
INTERNET-DRAFT
<draft-fielding-http-spec-01.ps>
Expires June 19, 1995

T. Berners-Lee
R. T. Fielding
H. Frystyk Nielsen
December 19, 1994

Hypertext Transfer Protocol — HTTP/1.0

Status of this Memo

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

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

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

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

Abstract

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

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification reflects preferred usage of the protocol referred to as "HTTP/1.0", and is compatible with the most commonly used HTTP server and client programs implemented prior to November 1994.

Table of Contents

1. Introduction

1.1 Purpose

The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification reflects preferred usage of the protocol referred to as "HTTP/1.0". This specification does not necessarily reflect the "current practice" of any single HTTP server or client implementation. It does, however, seek to remain compatible with existing implementations wherever possible, and should be considered the reference for future implementations of HTTP/1.0.

Practical information systems require more functionality than simple retrieval, including search, front-end update, and annotation. HTTP/1.0 allows an open-ended set of methods to be used to indicate the purpose of a request. It builds on the discipline of reference provided by the Universal Resource Identifier (URI) [2], as a location (URL) [3] or name (URN), for indicating the resource on which a method is to be applied. Messages are passed in a format similar to that used by Internet Mail [7] and the Multipurpose Internet Mail Extensions (MIME) [4].

HTTP/1.0 is also used for communication between user agents and various gateways, allowing hypermedia access to existing Internet protocols like SMTP [12], NNTP [11], FTP [14], Gopher [1], and WAIS [8]. HTTP/1.0 is designed to allow such gateways, via proxy servers, without any loss of the data conveyed by those earlier protocols.

1.2 Overall Operation

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

On the Internet, the communication generally takes place over a TCP/IP connection. The default port is TCP 80 [15], but other ports can be used. This does not preclude the HTTP/1.0 protocol from being implemented on top of any other protocol on the Internet, or on other networks. The mapping of the HTTP/1.0 request and response structures onto the transport data units of the protocol in question is outside the scope of this specification.

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

1.3 Terminology

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

connection
A virtual circuit connecting two parties for the purpose of communication.

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

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

resource
A network data object or service which can be identified by a URI.

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

user agent
The client program which is closest to the user and which initiates requests at their behest.

server
A program that accepts connections in order to service requests by sending back responses.

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

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

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

2. Notational Conventions and Generic Grammar

2.1 Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [7]. Implementors will need to be familiar with the notation in order to understand this specification. The augmented BNF includes the following constructs:

name = definition

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

"literal"

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

rule1 | rule2

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

(rule1 rule2)

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

*rule

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

[rule]

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

N rule

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

#rule

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

; comment

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

2.2 Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs. The US-ASCII character set is defined in [17].

```
OCTET          = <any 8-bit character>
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)>
HTAB           = <US-ASCII HT, horizontal-tab (9)>
<">            = <US-ASCII double-quote mark>
```

HTTP/1.0 defines the character sequence CR LF as the end-of-line marker for all protocol elements except the Object-Body (see Appendix A for tolerant applications). The end-of-line marker for an Object-Body is defined by its associated media type, as described in Section 4.2.2.

```
CRLF           = CR LF
```

HTTP/1.0 headers can be folded onto multiple lines if the continuation lines begin with linear whitespace characters. All linear whitespace (including folding) has the same semantics as SP.

```
LWS            = [CRLF] ( SP | HTAB )
```

Many HTTP/1.0 header field values consist of words separated by LWS or special characters. These special characters must be in a quoted string to be used within a parameter value.

```
word           = token | quoted-string

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

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

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

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

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

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

The text rule is only used for descriptive field contents. Words of *text may contain characters from character sets other than US-ASCII only when encoded according to the rules of RFC 1522 [5].

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

3. HTTP Message

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

```
HTTP-message   = Simple-Request            ; HTTP/0.9 messages
               | Simple-Response
               | Full-Request              ; HTTP/1.0 messages
               | Full-Response
```

Full-Request and Full-Response use the generic message format of RFC 822 [7] for transferring objects. Both messages may include optional header fields (a.k.a. "headers") and an object body. The object body is separated from the headers by a null line (i.e., a line with nothing preceding the CRLF).

```
Full-Request   = Request-Line              ; see Section 5.1
                 *General-Header           ; see Section 4.3
                 *Request-Header           ; see Section 5.5
                 *Object-Header            ; see Section 7
                 CRLF
                 [ Object-Body ]

Full-Response  = Status-Line               ; see Section 6.1
                 *General-Header           ; see Section 4.3
                 *Response-Header          ; see Section 6.4
                 *Object-Header            ; see Section 7
                 CRLF
                 [ Object-Body ]
```

Simple-Request and Simple-Response, as defined in Section 5 and Section 6 respectively, do not allow the use of any header information and are limited to a single request method (GET). This prevents the client from using content negotiation and the server from identifying the media type of the returned object. Therefore, clients are discouraged from using the Simple-Request format except for the simplest of applications.

3.1 Header Fields

HTTP header fields, which include Request-Header, Response-Header, General-Header, Object-Header, and extension fields, follow the same generic format as that given in Section 3.1 of RFC 822 [7]. Each header field consists of a name followed by a colon (":") and the field value. The field value may be preceded by any amount of linear-white-space, though a single SP is preferred. Header fields can be extended over multiple lines by preceding each extra line with one or more linear white-space characters.

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

field-name     = 1*<any CHAR, excluding CTLs, SP, and ":">
field-value    = *( field-content | comment | LWS )

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

The order in which header fields are received is not significant. However, it is considered "good practice" to send General-Header fields first, followed by Request-Header or Response-Header fields prior to the Object-Header fields. Comments can be included in HTTP header fields by surrounding the comment text with parentheses.

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

> ***Note:*** *Use of comments within HTTP headers is generally discouraged, since they are rarely seen by human eyes and hence only increase network traffic. However, they may be useful for messages posted or retrieved via NNTP and SMTP gateways.*

3.2 Object Body

The object body (if any) sent with an HTTP/1.0 request or response is in a format and encoding defined by the Object-Header fields.

```
Object-Body    = *OCTET
```

The actual length, encoding, and data type of the Object-Body is determined via the header fields Content-Length, Content-Encoding, Content-Transfer-Encoding, and Content-Type, similar to those defined by MIME [4]. If the Content-Length header field is present, its value in bytes (number of octets) represents the length of the Object-Body. Otherwise, the body length is determined either by a heuristic function of the Content-Type and Content-Encoding, or by the closing of the connection by the server.

> ***Note:*** *Closing the connection cannot be used to indicate the value of the* Content-Length *header when the* Object-Body *is part of a request message, as it leaves no possibility for the server to send back a response.*

4. Usage of RFC 822 and MIME Constructs

HTTP/1.0 reuses many of the constructs defined for Internet Mail (RFC 822, [7]) and the Multipurpose Internet Mail Extensions (MIME, [4]) to allow Object's to be transmitted in an open variety of representations. However, because it is not limited by the restrictions of existing mail protocols and gateways, HTTP does not obey some of the constraints imposed by RFC 822 and MIME for mail transport. This section describes how these common constructs are defined within HTTP.

4.1 Date/Time Format

For historical reasons, HTTP/1.0 allows three different formats for the representation of date/time stamps:

```
Sun, 06 Nov 1994 08:49:37 GMT    ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT   ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994         ; ANSI C's asctime() format
```

The first format is preferred as an Internet standard and represents a fixed-length subset of that defined by RFC 1123 [6] (an update to RFC 822 [7]). The second format is in common use today, but is based on the obsolete RFC 850 [10] date format and lacks a four-digit year. HTTP/1.0 clients and servers must accept all three formats, though they should never generate the third (asctime) format. It is strongly recommended that future clients and servers only generate the RFC 1123 format for representing date/time stamps in HTTP/1.0 requests and responses.

All HTTP/1.0 date/time stamps must be represented in Universal Time (UT), also known as Greenwich Mean Time (GMT), without exception. This is indicated in the first two formats by the inclusion of "GMT" as the three-letter abbreviation for time zone, and should be assumed when reading the asctime format.

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

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

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

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

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

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

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

Note: *HTTP/1.0 requirements for the date/time stamp format apply only to their usage within the protocol stream. Clients and servers are not required to use these formats for user presentation, request logging, etc.*

4.2 Media Types

HTTP uses Internet Media Types [13], formerly referred to as MIME Content-Types [4], in order to provide open and extensible data typing and type negotiation. For mail applications, where there is no type negotiation between sender and receiver, it is reasonable to put strict limits on the set of allowed media types. With HTTP, however, user agents can identify acceptable media types as part of the connection, and thus are allowed more freedom in the use of non-registered types. The following grammar for media types is a superset of that for MIME.

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

parameter      = attribute "=" value
attribute      = token                   ; case-insensitive
value          = token | quoted-string   ; sometimes
                                         ; case-sensitive
```

4.2.1 Multipart Types

HTTP provides for a number of "multipart" types — encapsulations of several object body parts within a single message's Object-Body. Multipart responses should only be used in responses when the user agent has indicated acceptability of the multipart type in addition to the media types of each constituent body part. User agents may use multipart types when submitting objects as part of a POST or PUT request.

As in MIME [4], all multipart types share a common syntax and must include a boundary parameter as part of the media-type. Unlike in MIME, multipart body parts may contain HTTP header fields which are significant to the meaning of that part.

```
boundary         = 0*69( bchar | SP ) bchar
bchar            = DIGIT | ALPHA | "'" | "(" | ")" | "+"
                 |  "_" | "," | "-" | "." | "/" | ":" | "=" | "?"
```

The Object-Body of a multipart message is specified as follows:

```
multipart-body   = discard-text 1*encapsulation
                   close-delimiter discard-text

encapsulation    = delimiter body-part CRLF

delimiter        = "--" boundary CRLF        ; boundary is obtained
                                             ; from Content-Type.

close-delimiter  = "--" boundary "--" CRLF

discard-text     = *(*text CRLF)             ; ignored

body-part        = *Object-Header
                   CRLF
                   [ Object-Body ]           ; May be recursive
                                             ; if boundary differs
```

A URI-header field (Section 7.9) should be included in the body-part for each enclosed object that can be identified by a URI.

4.2.1.1 Multipart/mixed

The "multipart/mixed" media type is used when the first body-part contains references to other parts which the sender wishes to send at the same time. For example, the first body-part could be an HTML document and the following body-parts could be annotations upon that document. However, the use of "multipart/mixed" is strongly discouraged in cases where the related objects are likely to have already been retrieved and cached by a user agent or caching proxy.

4.2.1.2 Multipart/parallel

The "multipart/parallel" media type is identical to "multipart/mixed", but with the additional semantics that the parts should be presented simultaneously by the user agent. This media type would be appropriate for situations where simultaneous presentation is an important aspect of the information, such as for audio-annotated slides and movies.

> ***Note:*** *This document does not define what is meant by "simultaneous presentation." That is, HTTP/1.0 does not provide any means of synchronization between the parts in messages of type "multipart/parallel".*

4.2.1.3 Other Multipart Types

The other multipart types registered by IANA [15] do not have any special meaning for HTTP/1.0, though user agents may need to understand each type in order to correctly interpret the purpose of each body-part.

4.2.2 Conversion to Canonical Form

Regardless of the media type, HTTP does not require any conversion to canonical form of line terminators or other constructs in the Object-Body while it is being transferred between an HTTP client and server. However, gateway applications must understand that, before an HTTP Object-Body can passed to a MIME-conforming protocol, conversion to canonical form may be necessary [4]. Furthermore, additional processing may be required in order to convert an HTTP message to something that can be transferred via other protocols, particularly when a Content-Encoding has been applied to the enclosed object.

In contrast, no conversion should be necessary for a MIME-conforming message to be tranferred using HTTP.

4.3 General Message Header Fields

There are a few header fields which have general applicability for both request and response messages, but which do not apply to the communicating parties or the object being transferred. Although none of the General-Header fields are required, they are all strongly recommended where their use is appropriate, and should be understood by all future HTTP/1.0 clients and servers. These headers apply only to the message being transmitted.

```
General-Header  = Connection
                | Date
                | Forwarded
                | Mandatory
                | Message-ID
                | MIME-Version
```

4.3.1 Connection

The Connection header is used to specify the parameters (desired or actual) of the current connection. Clients can use this header to indicate their desire to use a set of connection options. Servers can use this header to indicate what options are actually being applied. This field applies only to the current connection — receivers should not cache or otherwise save the connection information after the connection is closed. Proxies must not forward this header, though they may generate a separate Connection header for their own connections.

```
Connection      = "Connection" ":" 1#connect-option

connect-option  = token [ "=" word ]
```

Although current HTTP/1.0 clients and servers do not make use of the Connection header outside of experiments, this field will be necessary to enable future extensibility of connection-specific behavior. Most importantly, HTTP/1.0 proxies need to know that they must not forward this header even when they do not understand or make use of its contents. For example, an experimental client may send:

```
Connection: keep-alive
```

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

```
Connection: keep-alive, timeout=10, maxreq=5
```

to indicate that the connection will be kept open for a maximum of 5 requests, but will timeout if the next request is not received within 10 seconds. Note that the semantics of these options are not defined for HTTP/1.0, though similar options may be defined by future versions of HTTP.

4.3.2 Date

The Date header represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822. If a message is received via direct connection with the user agent (in the case of requests) or the origin server (in the case of responses), then the default date can be assumed to be the current date at the receiving end. However, since the date—as it is believed by the origin—is important for evaluating cached responses, origin servers should always include a Date header. A received message which does not have a Date header field should be assigned one by the receiver if and only if the message will be cached by that receiver or gatewayed via a protocol which requires a Date. The field value is an HTTP-date, as described in Section 4.1.

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

An example is

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

Only one Date header field is allowed per message.

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

4.3.3 Forwarded

The Forwarded header is to be used by proxies to indicate the intermediate steps between the user agent and the server (on requests) and between the origin server and the client (on responses). It is analogous to the “Received” field of RFC 822 and is intended to be used for tracing transport problems and avoiding request loops.

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

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

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

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

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

4.3.4 Mandatory

The Mandatory header is used to indicate a list of other header field names which must be understood by the receiver before the contents of the message can be stored, cached, or presented to a user. This header is used to alert the receiver that, unlike the default behavior, it cannot safely ignore the semantics of the listed field-names if they are not understood. It is anticipated that the field names correspond to conditions that must be met by the receiver in order to abide by the sender's licensing, copyright, payment, or other fair-usage constraints.

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

Recommended usage of the Mandatory header is not defined for HTTP/1.0. However, caching proxies must not cache responses which contain mandatory header information which is not understood by the proxy.

4.3.5 Message-ID

The Message-ID field in HTTP is identical to that used by Internet Mail and USENET messages, as defined in [10]. That is, it gives the message a single, unique identifier which can be used for identifying the message (not its contents) for "much longer" than the expected lifetime of that message.

```
Message-ID     = "Message-ID" ":" "<" addr-spec ">"
addr-spec      = unique-string "@" FQDN
unique-string  = <1*CHAR, not including whitespace, "@", or ">">
```

where unique-string must be unique within the host specified by FQDN. An example is

```
Message-ID: <9411151630.4256@info.cern.ch>
```

which is composed using the time, date and process-ID on the host *info.cern.ch*.

4.3.6 MIME-Version

HTTP is not a MIME-conformant protocol. However, HTTP/1.0 messages may include a single MIME-Version header field to indicate what version of the MIME protocol was used to construct the message. Use of the MIME-Version header field should indicate that the message is in full compliance with the MIME protocol (as defined in [4]). Unfortunately, current versions of HTTP/1.0 clients and servers use this field indiscriminantly, and thus receivers must not take it for granted that the message is indeed in full compliance with MIME. Gateways are responsible for ensuring this compliance (where possible) when exporting HTTP messages to strict MIME environments. Future HTTP/1.0 applications must only use MIME-Version when the message is intended to be MIME-conformant.

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

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

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 object requested, the identifier of the object, and the protocol version in use. For backwards compatibility with the more limited HTTP/0.9 protocol, there are two valid formats for an HTTP request:

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

Simple-Request  = "GET" SP URI CRLF          ; HTTP/0.9 request

Full-Request    = Request-Line               ; see Section 5.1
                  *General-Header            ; see Section 4.3
                  *Request-Header            ; see Section 5.5
                  *Object-Header             ; see Section 7
                  CRLF
                  [ Object-Body ]            ; see Section 3.2
```

If an HTTP/1.0 server receives a Simple-Request, it must respond with an HTTP/0.9 Simple-Response. Similarly, if a client receives a response that does not begin with a Status-Line, it should assume that the response is a Simple-Response and parse it accordingly.

5.1 Request-Line

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

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

5.2 Method

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

```
Method          = "GET" | "HEAD" | "PUT" | "POST"
                | "DELETE" | "LINK" | "UNLINK"
                | extension-method

extension-method= token
```

The methods GET and HEAD must be supported by all conforming HTTP/1.0 servers. The list of methods acceptable by a specific resource can be specified in an “Allow” Object-Header (Section 7.1). However, the client is always notified through the return code of the response whether a method is currently allowed on a specific resource, as this can change dynamically. The set of common methods for HTTP/1.0 is described below. Although this set can be easily expanded, additional methods cannot be assumed to share the same semantics for separately extended clients and servers. In order to maintain compatibility, the semantic definition for extension methods should be registered with the HTTP registration authority (Section 10). Servers should return the Status-Code “501 Not Implemented” if the method is unknown.

5.2.1 GET

The GET method means retrieve whatever object is identified by the URI. In the case where the URI refers to a data-producing process, or a script which can be run by such a process, it is the produced data which shall be returned as the Object-Body in the response and not the source text of the script or process (unless that happens to be the output of the data-producing process).

The semantics of the GET method is changed to a "conditional GET" if the request message includes an If-Modified-Since header field. A conditional GET method requests that the identified Object-Body be transferred only if it has been modified since the date given by the If-Modified-Since header. If it has been modified (or the passed If-Modified-Since date is invalid), the response is exactly the same as for a normal GET. If the object has not been modified since the If-Modified-Since date, the server shall return a "304 Not Modified" response. The "conditional GET" method is intended to reduce network usage by allowing cached objects to be refreshed without requiring multiple requests and transferring unnecessary data.

Data originating from HTML forms [16] can be passed to the server using the GET method by appending a "?" and a set of attribute/value pairs. Section 11.2 describes when to use GET and POST respectively when passing form data in a request.

5.2.2 HEAD

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

5.2.3 POST

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

- Annotation of existing documents;
- Posting a message to a bulletin board topic, newsgroup, mailing list, or similar group of articles;
- Providing a block of data (usually a form) to a data-handling process, or a script which can be run by such a process;
- Extending a document during authorship.

The posted object is considered to be subordinate to the specified URI, in the way that a file is subordinate to a directory containing it, or a news article is subordinate to a newsgroup to which it is posted.

The client can suggest a URI for identifying the new resource by including a URI-header field in the request. However, the server should treat that URI as advisory only and may store the object under a different URI. The origin server must inform the user agent of the allocated URI via a URI-header in the response.

The client apply relationships between the new resource and other existing resources by including Link header fields, as described in Section 7.14. The server may use the Link information to perform other operations as a result of the new object being added. For example, lists and indexes might be updated. However, no mandatory operation is imposed on the origin server. The origin server may also generate its own or additional links to other resources.

If a resource has been created on the origin server, the response should contain the allocated URI and all applicable Link header fields, along with an Object-Body (preferably of type "text/html") which describes the status of the request and refers to the new resource.

A successful POST does not require that the object be created as a resource on the origin server or made accessible for future reference. That is, the action performed by the POST method might not result in a resource that can be identified by a URI. In this case, a “200 OK” is the appropriate Status-Code returned in the response. If a resource has been created, “201 Created” should be the response.

> ***Note:*** *The user agent may not assume any postconditions of the method in terms of web topology. For example, if a POST is accepted, the effect may be delayed or overruled by human moderation, batch processing, etc. The user agent should not rely on the resource being immediately (or ever) created.*

5.2.4 PUT

The PUT method specifies that the enclosed Object in the request is to be stored under the supplied URI. If the URI points to an already existing resource, the enclosed Object should be considered a modified version of the one residing on the origin server. If the URI does not point to an existing resource, and that URI is capable of being defined as a new resource from the requesting user agent, the origin server can create the resource at that URI.

The client can create or modify relationships between the enclosed Object and other existing resources by including Link header fields, as described in Section 7.14. As with POST, the server may use the Link information to perform other operations as a result of the request. However, no mandatory operation is imposed on the origin server. The origin server can generate its own or additional links to other resources.

The actual method for determining how the resource is placed, and what happens to its predecessor, is defined entirely by the origin server. If version control is implemented by the origin server, the Version and Derived-From header fields should be used to help identify and control revisions to a resource.

5.2.5 DELETE

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

5.2.6 LINK

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

5.2.7 UNLINK

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

5.3 HTTP-Version

The HTTP-Version element defines the version of the HTTP protocol being used for the request. If the protocol version is not specified, the server shall assume that the client uses HTTP version 0.9 and the response should be formatted as a Simple-Response.

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

Full-Request messages which apply the protocol defined by this document should use an HTTP-Version of "`HTTP/1.0`".

5.4 Universal Resource Identifier

The URI is a Universal Resource Identifier, as defined in RFC 1630 [2], and identifies the resource upon which to apply the request.

```
URI             = <As defined in RFC 1630>
```

Unless the server is being used as a proxy, a partial URI shall be given with the assumptions of the protocol (http) and host name (the server's address) being obvious. That is, if the full URI looks like

http://info.cern.ch/hypertext/WWW/TheProject.html

then the corresponding partial URI in the Simple-Request or Full-Request is

/hypertext/WWW/TheProject.html

If the client is sending the request through a proxy, the protocol and host name must be explicitly declared.

The URI must be encoded using the escaping scheme described in [2].

5.5 Request Header Fields

The request header fields allow the client to pass additional information about the request (and about the client itself) to the server. All header fields are optional and conform to the generic HTTP-header syntax.

```
Request-Header  = User-Agent
                | If-Modified-Since
                | Pragma
                | Authorization
                | Proxy-Authorization
                | Referer
                | From
                | Accept
                | Accept-Encoding
                | Accept-Language
```

Unknown header fields should be considered Object-Header fields.

5.5.1 User-Agent

The User-Agent field contains information about the user agent originating the request. This is for statistical purposes, the tracing of protocol violations, and automated recognition of user agents for the sake of tailoring responses to avoid particular user agent limitations or features. Although it is not required, user agents should always include this field with requests. The field can contain multiple tokens specifying the product name, with an optional slash and version designator, and other products which form a significant part of the user agent. By convention, the products are listed in order of their significance for identifying the application.

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

product         = token ["/" product-version]
product-version = 1*DIGIT "." 1*DIGIT
```

Example:

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

Product tokens should be short and to the point — use of this field for advertizing or other non-essential information is explicitly deprecated and will be considered as non-conformance to the protocol.

> ***Note:*** *Some current proxy applications append their product information to the list in the User-Agent field. This is no longer recommended, since it makes machine interpretation of these fields ambiguous. Instead, proxies should use the Forwarded header described in Section 4.3.3.*

5.5.2 If-Modified-Since

The If-Modified-Since header field is used with the GET method to make it conditional: if the requested document has not been modified since the time specified in this field, the document will not be returned from the server; instead, a “304 Not Modified” response will be returned without any Object-Body.

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

An example of the field is:

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

The purpose of this feature is to allow efficient updates of local cache information with a minimum amount of transaction overhead. The same functionality can be obtained, though with much greater overhead, by issuing a HEAD request and following it with a GET request if the server indicates that the object has been modified.

5.5.3 Pragma

The Pragma header field is used to specify directives are to be be applied by servers to which they are relevant (e.g. proxies). They allow the client to request a certain behavior by all servers along the length of a request chain. Although multiple pragma directives can be listed as part of the request, HTTP/1.0 currently only defines the semantics for the "`no-cache`" directive.

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

pragma-directive  = "no-cache" | extension-pragma
extension-pragma  = token
```

When the “`no-cache`” directive is present, a caching proxy should not return a document from the cache even though it has not expired, but it should always request the document from the actual server.

Pragmas must be passed through by proxies even though they might have significance to the proxy itself. This is necessary in cases when the request has to go through many proxies, and the pragma may affect all of them. It is not possible to specify a pragma for a specific proxy; however, any pragma-directive not relevant to a gateway or proxy should be ignored.

5.5.4 Authorization

This version of the HTTP protocol contains a simple access authentication scheme which is explained in Section 9. The Authorization header field identifies the user that originated the request.

```
Authorization          = "Authorization" ":" (
                           ( "Basic" encoded-cookie )
                         | ( extension-scheme [ extension-encrypted ] ) )

encoded-cookie         = <base64 encoding of userid-password>
userid-password        = [ token ] ":" *text

extension-scheme       = token
extension-encrypted    = word
```

This field is extensible so that it can be used to identify more advanced encryption schemes. The first word is a specification of the authorization system in use, followed by an encoded version of the User-ID and the password (separated by a ":" in decoded form). The encoding used for the Basic scheme is identical to the base64 encoding of MIME [4] without line breaks. The Basic scheme provides only a low level of authentication similar to the methods used by unmodified FTP and Telnet. It cannot be considered a mechanism for secure transfer.

> ***Note:*** *Since both the User-ID and the password are optional, the* userid-password *could be just the single colon character* (":")*. However, that is not recommended practice.*

5.5.5 Proxy-Authorization

The Proxy-Authorization header field allows the client to identify itself (or its user) to a proxy which requires authentication. The format is the same as for Authorization.

```
Proxy-Authorization    = "Proxy-Authorization" ":" (
                           ( "Basic" encoded-cookie )
                         | ( extension-scheme [ extension-encrypted ] ) )
```

Unlike Authorization, the Proxy-Authorization applies only to the current connection and must not be passed on to higher-level servers or proxies.

5.5.6 Referer

The Referer field allows the client to specify, for the server's benefit, the address (URI) of the document (or element within the document) from which the URI in the request was obtained. This allows a server to generate lists of back-links to documents, for interest, logging, optimized caching etc. It also allows obsolete or mistyped links to be traced for maintenance. The format of the field is:

```
Referer         = "Referer" ":" URI
```

Example:

```
Referer: http://info.cern.ch/hypertext/DataSources/Overview.html
```

If a partial URI is given, then it should be interpreted relative to the URI of the object of the request.

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

5.5.7 From

The From header field, if given, should contain an Internet e-mail address for the human user who controls the requesting user agent. It should contain a machine-usable address as defined by addr-spec in RFC 822:

```
From            = "From" ":" addr-spec
```

An example is:

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

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

The Internet e-mail address in this field does not have to correspond to the Internet host which issued the request. (For example, when a request is passed through a proxy, then the original issuer's address should be used). The address should, if possible, be a valid Internet e-mail address, whether or not it is in fact an Internet e-mail address or the Internet e-mail representation of an address on some other mail system.

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

5.5.8 Accept

The Accept header field can contain a list of media types which are acceptable as a response to the request. The set given by the client should represent what is acceptable given the context of the request.

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

```
Accept          = "Accept" ":" 1#(
                      ("*" | type) "/" ("*" | subtype)
                     *(";" parameter)
                      [ ";" "q" "=" ( "0" | "1" | float ) ]
                      [ ";" "mxb" "=" 1*DIGIT ] )
float           = < ANSI-C floating point text representation,
                  where (0.0 < float < 1.0) >
```

q is the quality factor of how well the client can handle the media type and mxb is the maximum accepted size of the Object-Body in number of octets (decimal). The definition does not prohibit duplicate accept-param's, but leaves the interpretation undefined. See Section 8 for a description of the negotiation algorithm. If at least one Accept header is present, a quality factor of 0 is equivalent to not sending an accept header field containing that media-type or set of media-types. The default values are: q=1 and mxb=undefined (i.e. infinity).

In order to save time, and also allow clients to receive media types of which they may not be aware, an asterisk "*" may be used in place of either the type token and/or the subtype token. The example

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

should verbally be interpreted as "if you have audio/basic, send it; otherwise send me some other audio".

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

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

or

```
Accept: */*
```

A more elaborate example is

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

Verbally, this should be interpreted as "text/html and text/x-c are the preferred media types, but if they do not exist then send the Object-Body in text/x-dvi if the object is less than 100000 bytes. If this is not the case then send text/plain".

> ***Note:*** *In earlier versions of this document, the mxs parameter defined the maximum acceptable delay in seconds before the response would arrive. This has been removed as the server has no means of obtaining a useful reference value. However, this does not prevent the client from internally measuring the response time and optimizing the accept header field accordingly.*

5.5.9 Accept-Encoding

The Accept-Encoding header field is similar to Accept, but lists the Content-Encoding types which are acceptable in the response. Content-Encoding is described in Section 7.4.

```
Accept-Encoding          = "Accept-Encoding" ":"
                           1#(encoding-mechanism [ ";" encoding-param ] )

encoding-mechanism       = extension-encoding
encoding-param           = "q" "=" ( "0" | "1" | float )
extension-encoding       = token
```

q is the quality factor of how well the client can handle the encoding type. The default value is q=1. As no encoding-mechanism's have been registered by IANA [15], all values are of type extension-encoding. However, two well-known extension-encoding's are:

"x-compress", "x-gzip"

If no Accept-Encoding is present then it is assumed that the client accepts *no* encoding at all. An example of use is

```
Accept-Encoding: x-compress
```

5.5.10 Accept-Language

The Accept-Language header field is similar to Accept, but lists the set of natural languages acceptable for the response to the request.

```
Accept-Language = "Accept-Language" ":"
                  1#(language-dialect *1(";" language-param) )

language-dialect  = ("*" | language) ["/" dialect ]
language-param    = "q" "=" ( "0" | "1" | float)
language          = <As defined in ISO 639 but case-insensitive>
dialect           = <As defined in ISO 3166 but case-insensitive>
```

As with the Accept field, a quality factor q can be specified which in this case describe the level of intelligibility to the user. The default value is q=1. The definition does not prohibit duplicate language-param's, but leaves the interpretation undefined. An example of it's use is

```
Accept-Language: dk, en/gb; q=0.5
```

meaning: "Send me a Danish version if you have it; else a British English version."

If no Accept-Language is present, it is assumed that the client accepts *all* natural languages with quality factor 1.

If the server cannot service the request with the language specified, or if the languages specified only represent a subset in case of a multi-linguistic Object-Body, it is not illegal to serve the request in an unspecified language. The character "*" can be used to indicate "any language" and/or "any dialect".

> ***Note:*** *As intelligibility is highly dependent on the individual user, it is recommended that client applications make the choice of linguistic preference available to the user.*

6. Response

If the client has issued an HTTP request, the response from the server shall consist of the following:

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

Simple-Response = [Object-Body]

Full-Response   = Status-Line              ; see Section 6.1
                  *General-Header          ; see Section 4.3
                  *Response-Header         ; see Section 6.4
                  *Object-Header           ; see Section 7
                  CRLF
                  [ Object-Body ]          ; see Section 3.2
```

A Simple-Response should only be sent in response to an HTTP/0.9 Simple-Request. Note that the Simple-Response consists only of the object that was requested and is terminated by the server closing the connection.

6.1 Status-Line

The Status-Line consists 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
```

6.2 HTTP Version

The HTTP-Version element identifies the protocol version being used by the server. The format of this field is identical to the corresponding HTTP-Version field in the Request-Line described in Section 5.3.

6.3 Status Codes and Reason Phrases

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine the Reason-Phrase, nor to pass it on to the human user.

```
Status-Code     = 3DIGIT

Reason-Phrase   = token *( SP token )
```

All responses, regardless of the Status-Code, may contain an Object-Header and/or an Object-Body. This can either be the object pointed to by the requested URI or an object containing further explanation of the Status-Code. In the latter case, the preferred media type is “text/html”, but “text/plain” is also acceptable.

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

- 1xx: Not used, but reserved for future use
- 2xx: Success - The requested action was successfully received and understood
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or is inherently impossible to fulfill
- 5xx: Server Error - The server could not fulfill the request

The values of the numeric status codes and an example set of corresponding Reason-Phrase's are presented below. Every Status-Code has a description of which method it can follow and any metainformation required in the HTTP-header.

6.3.1 Successful 2xx

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

200 OK

- Following: GET, HEAD, POST
- Required metainformation: none

The request could be fulfilled and an Object-Header should be returned to the client in the response. In the case of GET, the response should also contain an Object-Body.

201 Created

- Following: POST, PUT
- Required metainformation: URI-header

This indicates that the POST has been successful or that the PUT resulted in a new object. The newly created object can be referenced by the URI returned in the URI-header field in the response. This action can, at any time, be overridden at the origin server (possibly by human intervention), so this status code is no guarantee that the object continue to be available at the given URI.

202 Accepted

- Following: GET, HEAD, PUT, POST, DELETE
- Required metainformation: none

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

203 Provisional Information

- Following: GET, HEAD, POST
- Required metainformation: none

When received in the response, this indicates that the returned metainformation in the HTTP-header is not the definitive set as available from the origin server, but is gathered from a local or a third party copy. The set presented can either be a subset or a superset of the original version, for example including annotation information about the resource.

204 No Response

- Following: GET, HEAD, POST
- Required metainformation: none

The server has received the request but there is no information to send back, and the client should stay in the same document view. This is mainly to allow input for scripts without changing the document at the same time.

205 Deleted

- Following: DELETE
- Required metainformation: none

The DELETE method was successful and the object has been removed by the requested server. This action can at any time be overridden by the origin server, for example by human interaction, so this status code is no guarantee that the operation has in fact been carried out.

206 Modified

- Following: PUT
- Required metainformation: none

The PUT method was successful and the object has been modified on the requested server. This action can at any time be overridden at the origin server, for example by human interaction so this status code is no guarantee that the operation has in fact been carried out.

6.3.2 Redirection 3xx

This class of status codes indicates that further action needs to be taken by the client in order to fulfill the request. The action required can normally be carried out by the client without interaction with the user, but it is strongly recommended that this only takes place if the method used in the request is either GET or HEAD.

301 Moved Permanently

- Following: GET, HEAD, POST, PUT
- Required metainformation: URI-header, Location

The object requested has been assigned a new permanent URI, and any future references to this object must be done using the returned URI. Clients with link editing capabilities are encouraged to automatically relink references to the URI requested to the new reference returned by the server, where possible.

> ***Note:*** *It is possible for the server to send back this status code in response to a request using the PUT and POST methods. However, as this might change the conditions under which the request was issued, the user agent should not automatically redirect the request unless it can be confirmed by the user.*

302 Moved Temporarily

- Following: GET, HEAD, POST, PUT
- Required metainformation: URI-header, Location

The data requested resides temporarily under a different URI. As the redirection may be altered on occasion, the client should on future requests from the user continue to use the original URI used for this request and not the URI returned in the URI-header field.

> ***Note:*** *It is possible for the server to send back this status code in response to a request using the PUT and POST methods. However, as this might change the conditions under which the request was issued, the user agent should not automatically redirect the request unless it can be confirmed by the user.*

303 Method

- Required metainformation: none

This code is obsolete.

304 Not Modified

- Following: conditional GET
- Required metainformation: none

If the client has performed a conditional GET request and access is allowed, but the document has not been modified since the date and time specified in the If-Modified-Since field, the server shall respond with this status code and must not send the Object-Body to the client. Metainformation contained in the response should only contain information relevant to cache managers and which may have changed independently of the object's Last-Modified date. Examples of relevant header fields are: Date, Server, and Expires. However, none of them are mandatory.

6.3.3 Client Error 4xx

The 4xx class of status codes is intended for cases in which the client seems to have erred. The codes can follow any method described in Section 5.2, and the set consists of:

400 Bad Request

- Required metainformation: none

The request had bad syntax or was inherently impossible to be satisfied. The client is discouraged from repeating the request without modifications.

401 Unauthorized

- Required metainformation: WWW-Authenticate

The server must return a WWW-Authenticate header field as described in Section 6.4.2 containing a list of authorization schemes in which at least one must be fulfilled in order for the client to obtain the Object-Body. The client should then retry the request with a suitable Authorization header field. The HTTP access authentication scheme is explained in Section 9.

402 Payment Required

- Required metainformation: none

This code is not currently supported, but is reserved for future use.

403 Forbidden

- Required metainformation: none

The request is, for some reason unknown to the client, forbidden. Authorization will not help and the request should not be repeated. This status code can also be used if the server does not want to make public whether the request can not be fulfilled due to insufficient authorization from the client or because the object does not exist.

404 Not Found

- Required metainformation: none

The server has not found anything matching the URI given. No indication is given whether the condition is temporary or permanent.

405 Method Not Allowed

- Required metainformation: Allow

The method specified in the Request-Line is not allowed for the object identified by the URI. The server should send back an Allow header containing a list of valid method's as explained in Section 7.1.

406 None Acceptable

- Required metainformation: Content-Type, Content-Encoding, Content-Language

The server has found an object matching the URI given, but not one that matches all of the conditions identified in the Accept, Accept-Encoding, and Accept-Language request headers. The response should include at least the Content-Type, the Content-Encoding, and the Content-Language, but is encouraged to include the object's complete metainformation. No Object-Body can be included in the response.

407 Proxy Authentication Required

- Required metainformation: Proxy-Authenticate

This code is similar to "401 Unauthorized" but it indicates that the user agent must first authenticate itself with the proxy. The proxy must return a Proxy-Authenticate header field as described in Section 6.4.3 containing a list of authorization schemes in which at least one must be fulfilled in order for the client to use the proxy. The client should then create a new request with the proxy as the server destination and with a suitable Proxy-Authorization header field. The HTTP access authentication scheme is explained in Section 9.

408 Request Timeout

- Required metainformation: none

This code indicates that the client did not produce a request within a time that the server was prepared to wait. If the client is still actively generating the request, it should immediately stop sending further information to the server.

6.3.4 Server Errors 5xx

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request. These codes can follow any method at any time.

> ***Note:*** *For all of the 5xx codes, the server is encouraged to send back an HTTP-header and an Object-Body containing an explanation of the error situation, and whether it is a temporary or permanent condition.*

500 Internal Server Error

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

501 Not Implemented

The server does not support the functionality required to fulfil the request.

502 Bad Gateway

This is equivalent to “500 Internal Server Error”, but for the case of a gateway or proxy accessing some other service, this indicates that the response from the other service was invalid. As from the point of view of the client and the HTTP transaction, the other service is hidden within the gateway or proxy, this may be treated identically to “500 Internal Server Error”, but has more diagnostic value.

503 Service Unavailable

- Required metainformation: Retry-After

The server is currently unable to handle the request. This can either be due to overload of the server or servicing of the server. The implication is that this is a temporary condition which will be alleviated after a delay indicated in the Retry-After header. If no Retry-After is present in the response then the client should handle the response as identical to a "500 Internal Server Error".

504 Gateway Timeout

This is equivalent to “500 Internal Server Error”, but for the case of a gateway or proxy accessing some other service, this indicates that the response from the other service did not return within a time that the gateway was prepared to wait. As from the point of view of the client and the HTTP transaction, the other service is hidden within the gateway or proxy, this may be treated identically to “500 Internal Server Error”, but has more diagnostic value.

6.4 Response Header Fields

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

```
Response-Header= Server
               | WWW-Authenticate
               | Proxy-Authenticate
               | Retry-After
```

Unknown header fields should be considered Object-Header fields.

6.4.1 Server

The Server header field contains information about the software being used by the origin server program handling the request. The field is analogous to the User-Agent field and has the following format:

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

Example:

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

If the response is being forwarded through a proxy, the proxy application must not add its data to the product list. Instead, it should include a Forwarded field, as described in Section 4.3.3.

6.4.2 WWW-Authenticate

The WWW-Authenticate header field must be included as part of the response if the server sends back a "401 Unauthorized" Status-Code on a request from the client as part of the Basic Authentication Scheme described in Section 9. This header field indicates the authentication scheme in use and the realm in which the requested URI belongs.

```
WWW-Authenticate     = "WWW-Authenticate" ":" (
                          ( "Basic" realm )
                        | ( extension-scheme realm ) )
realm                = "Realm" "=" 1#( "<" URI ">" )
```

The first word of the field value identifies the authorization scheme in use and is followed by the realm of the protected URI. The realm is a comma separated list of URIs, where relative URLs should be interpreted relative to the URI of the requested resource in the RequestLine. If a request is authenticated and a realm specified, the User-ID and password should be valid for all other requests within this realm.

Note: *The realm may span more than one origin server.*

6.4.3 Proxy-Authenticate

The Proxy-Authenticate header field must be included as part of the response if the proxy sends back a "407 Proxy Authentication Required" Status-Code on a request from the client. This header field indicates the authentication scheme in use and the realm for which the proxy is protected.

```
Proxy-Authenticate   = "Proxy-Authenticate" ":" (
                          ( "Basic" realm )
                        | ( extension-scheme realm )
```

Unlike WWW-Authenticate, the Proxy-Authenticate applies only to the current connection and must not be passed on to lower-level user agents or proxies.

6.4.4 Retry-After

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

```
Retry-After     = "Retry-After" ":" ( HTTP-date | delta-seconds )
delta-seconds   = 1*DIGIT
```

Two examples of use are

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

In the latter example, the delay is 2 minutes.

7. Object Header Fields

Full-Request and Full-Response messages can contain Object-Header fields and an Object-Body (as defined in Section 3). This section specifies the format and contents of the Object-Header fields.

Object-Header fields define metainformation about the Object-Body. All are optional.

```
Object-Header    = Allow
                 | Content-Length
                 | Content-Type
                 | Content-Encoding
                 | Content-Transfer-Encoding
                 | Content-Language
                 | Expires
                 | Last-Modified
                 | URI-header
                 | Location
                 | Version
                 | Derived-From
                 | Title
                 | Link
                 | extension-header

extension-header = HTTP-header
```

In this section, *recipient* refers to either the client or the server, depending on who receives the object. Each object header field is explained in the subsections below. Other header fields are allowed but cannot be assumed to be recognizable by the recipient. Unknown header fields should be ignored by the recipient, but passed on to downstream recipients (if any).

7.1 Allow

The "Allow" header field lists the set of methods supported by the object identified by the requested URI. The purpose of this field is strictly to inform the recipient of valid methods associated with the object. This does not prevent a client from trying other methods. However, it is recommended that the indications given by this field be followed. This field has no default value. If left undefined, the set of allowed methods is defined by the origin server at the time of each request.

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

Example of use:

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

Note: *If a response passes through a proxy which does not understand one or more of the methods indicated in the* Allow *header, the proxy should not try to modify the* Allow *header, since the user agent may have other means of communicating with the origin server.*

7.2 Content-Length

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

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

An example is

```
Content-Length: 3495
```

Even though it is not mandatory, applications are strongly encouraged to use this field to indicate the size of the Object-Body to be transferred regardless of the media type of the object.

Any Content-Length of size greater than or equal to zero is a valid value. If undefined, the length of the Object-Body may be determined by the media type (if a multipart type is used), by the content encoding (if a delimited encoding is used), or by the closing of the connection by the server. However, since many applications do not accept multipart types or delimited encodings, a valid Content-Length is generally required for objects contained in PUT and POST requests.

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

7.3 Content-Type

The Content-Type header field indicates the Internet media type of the Object-Body (as described in Section 4.2) sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.

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

An example of the field is

```
Content-Type: text/html; charset=ISO-8859-1
```

All media-type's registered by IANA should be preferred over extension tokens. However, HTTP does not limit conforming applications to the use of officially registered media types, nor does it encourage the use of an "x-" prefix for unofficial types outside of explicitly short experimental use between consenting applications. Data providers are strongly encouraged to register their media types with IANA via the procedures outlined in RFC 1590 [13].

The Content-Type header field has no default value. If and **only** if the media type is unknown, the receiver may attempt to guess the media type via inspection of its content and/or the filename extension(s) of the URL used to access the object. If the media type remains unknown, the receiver should treat it as type "application/octet-stream".

7.4 Content-Encoding

The Content-Encoding header field, unique to HTTP, is used as a modifier to the media-type. When present, its value indicates the encoding mechanism applied to the associated Object-Body prior to it being enclosed in the message, and thus what decoding mechanism must be applied in order to obtain the media-type referenced by the Content-Type header field. This is primarily used to allow object compression without losing the identity of the underlying media type.

```
Content-Encoding= "Content-Encoding" ":" encoding-mechanism
```

An example of its use is

```
Content-Encoding: x-gzip
```

This field should not be confused with the Content-Transfer-Encoding field defined below. The purpose of the Content-Transfer-Encoding is "to indicate the type of transformation that has been used in order to represent the object in an acceptable manner for transport." In contrast, Content-Encoding is used to indicate any form of compression, encryption, or packetization mechanism used to transform the data prior to being transported, such that it is in a more convenient form for storage and/or transport. The result after encoding is usually binary, but could be any type defined by the Content-Transfer-Encoding field.

7.5 Content-Transfer-Encoding

Because all HTTP communication takes place on an 8-bit clean connection, the default Content-Transfer-Encoding for all messages is "binary". Note that this differs from the required default in MIME [4], so gateways between HTTP and MIME-compliant protocols must add an explicit "`Content-Transfer-Encoding: binary`" to the message header if a Content-Transfer-Encoding is not already present.

```
Content-Transfer-Encoding  = "Content-Transfer-Encoding" ":" cte-mechanism
cte-mechanism              = token
```

7.6 Content-Language

The Content-Language field describes the natural language of the Object-Body. It is defined as:

```
Content-Language           = "Content-Language" ":" 1#lang-dia
lang-dia                   = language ["/" dialect ]
```

An example of its use is

```
Content-Language: dk
```

means that the content of the message is in Danish with no dialect specified. The example

```
Content-Language: en/gb, dk
```

means that the language is Danish and British English.

Multilinguistic Object-Body's can be described using a list of lang-dia codes. This document does not specify any means to indicate the extent to which different natural languages are represented in the Object-Body.

> ***Note:*** *This field can be defined not only for textual documents, but also for audio and possibly other media as well. It should not be considered limited to* Object-Body*'s of type "text/*".*

In the absence of an explicit Content-Language header, the client should make no assumptions about the language of the returned Object-Body, as it might be a language-neutral or a multilingual document.

7.7 Expires

The Expires field gives the date and time after which the information given ceases to be valid and should be retrieved again if it has been kept as a local copy. This allows control of caching mechanisms, but the date and time indicated does not necessarily imply that the original object will cease to exist. This is completely controlled by the server. The format is an absolute date and time as defined by HTTP-date in Section 4.1. The formal description is

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

and an example of the use is

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

Note: *This field can also be used for automatic refreshing of dynamic or volatile data. However, this is completely dependent on the implementation of the client application to automatically issue a new request when the object has expired.*

Object's generated by data-producing processes, or scripts which can be run by such processes, are often dynamic by nature. Therefore, it is strongly recommended that responses containing such Objects's contain an Expires header field.

7.8 Last-Modified

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

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

An example of its use is

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

The exact meaning of this header field depends on the implementation of the sender. For files, it may be just the file system last-mod date. For entities with dynamically included parts, it may be the most recent of the set of last-mod dates of its component parts. For database gateways, it may be the last-update timestamp of the record. For virtual objects, it may be the last time its internal state changed. In any case, the recipient should only know (and care) about the result — whatever gets stuck in the Last-modified: header — and not worry about how it was obtained.

7.9 URI Header

The URI-header field contains a URI by which the object may be found. It should not be confused with the token in the Request-Line described in Section 5.4. As for a normal request, there is no guarantee that the resource can be accessed using the URI specified. The field is normally a part of a response having Status-Code "301 Moved Permanently" or "302 Moved Temporarily".

```
URI-header     = "URI" ":" 1#( "<" URI ">" [ ";" vary ] )

vary           = "vary" "=" <"> 1#vary-param <">
vary-param     = "type" | "language" | "version" | "encoding"
               |  "user-agent" | extension-vary

extension-vary = token
```

All URI's specified in this field can be either absolute or relative to the URI given in the RequestLine as specified by [9].

If a URI refers to a set of variants, then the dimensions in which the variants differ must be given with the vary parameters. Multiple occurrences of vary-param in the vary field give alternative access names or addresses for the object. An example of the field is:

```
URI: <http://info.cern.ch/hypertext/WWW/TheProject.multi>;
     vary="type,language"
```

This indicates that the URI indicated covers a group of possible resources which varies in media type and in natural language. The client can specify which of these objects to be returned in the response to a request using the request header fields: Accept, Accept-Encoding, Accept-Language, and Version. Another example is:

```
URI: <http://info.cern.ch/hypertext/WWW/TheProject.ps>;
     vary="encoding"
```

This indicates that the resource pointed to by the URI exists in different encodings as defined by the Content-Encoding field.

7.10 Location

The Location header field is an earlier form of the URI-header and is considered obsolete. However, HTTP/1.0 clients and servers should continue to support the Location header in order to properly interface with older applications. The purpose of Location is identical to that of the URI-header, except that no variants can be specified and only one absolute location URI is allowed.

```
Location        = "Location" ":" URI
```

An example is

```
Location: http://info.cern.ch/hypertext/WWW/NewLocation.html
```

7.11 Version

The Version field defines a version number referring to the current contents of an evolving object resident on the origin server. Together with the Derived-From field described in Section 7.12, it allows groups of people to work simultaneously on the creation of a work as an iterative process. The field should be used to indicate evolution along a single path of a particular work. It should not be used to indicate derived works or renditions in different representations.

```
Version         = "Version" ":" *text
```

Note: *The field should be present in the response if PUT is an allowed method to perform on the object pointed to by the requested URI. However the presence of the field can not be taken as an indication whether PUT is allowed or not.*

7.12 Derived-From

The Derived-From field contains the most recent value of the Version field before any modifications local to the transmitting application have been carried out on an evolving resource. The definition of the field value must be identical to the Version field value.

```
Derived-From    = "Derived-From" ":" *text
```

The definition of this field allows both the server and the client to employ a code management system to merge different versions of an evolving resource. As for the Version field, the Derived-From may only be used to indicate evolution along a single path of a particular work. It should not be used to indicate derived works or renditions in different representations.

> ***Note:*** *The definition allows different code management systems to be employed by the involved parties. The only requirement is a conforming mapping between any internal versioning system and the one defined by Derived-From and Version.*

7.13 Title

This header field indicates the title of the document, which is not to be considered as part of the object. The definition of the field is:

```
Title          = "Title" ":" *text
```

The field differs from the “Subject” field described in RFC 822 in that title is defined by the creator/author of a resource, but the “Subject” field is defined by the originator. The field is to be considered isomorphic with the <TITLE> element in HTML [16].

7.14 Link

The Link header provides a means for describing the relationship between HTTP-Object’s. An object can have multiple Link elements and can typically indicate relationships like hierarchical structure. The field is semantically equivalent to the <LINK> element in an HTML document.

```
Link           = "Link" ":" 1#("<" URI ">" [ ";" "REL" "=" relation ] )
relation       = "UseIndex" | "UseGlossary" | "Contents"
               |  "Next" | "Previous" | "Parent"
               |  "BookMark" | "Made" | "Help"
```

The reader is referred to the HTML specification [16] for a full explanation of the semantics for LINK relationships. Examples of usage include:

```
Link: <http://info.cern.ch/previous>; REL="Previous"

Link: <mailto:timbl@info.cern.ch>; REL="Made"
```

The first example indicates that this object is logically a continuation of the previous object identified by the URI. The second indicates that the author of the object is identified by the given e-mail address.

> ***Note:*** *It has been proposed that any HTML metainformation element (allowed within the <HEAD> as opposed to <BODY> element of the document) be a valid candidate for an HTTP object header. This document defines the two header fields Link and Title which both are examples of this.*

8. HTTP Negotiation Algorithm

The goal of the negotiation algorithm is to map a set of parameters into a one-dimensional space where the calculated weights represent the "degradation" figure of the resource. The maximum value of this set represents the media-type in which the Object-Body optimally should be returned to the client.

It is assumed that it is possible to assign an absolute value representing the amount of loss of value when the Object-Body is rendered into a specific media type. Whilst this is a very subjective measurement, and in fact largely a function of the document in question, the approximation is made that one can define this degradation figure as a function of merely the representation involved.

It is furthermore assumed that the cost to the user of viewing an Object also is a function of the time taken for the presentation. We first assume that the cost is linear in time, and then assume that the time is linear in the size of the Object-Body.

The calculated weights are normalized to a real number between 0 and 1 where 0 is the minimum value and 1 is the maximum value. This document defines the following parameters to be included in the algorithm:

q The quality factor representing the level of degradation when rendering the Object-Body in a specific media-type in the client application. The value is normalized so that $q \in [0,1]$, where the default value is q=1.

qs Equivalent to the q factor but for the server application in case it can perform media-type conversions. The default value is qs=1.

mxb The maximum number of bytes in the Object-Body accepted by the client. The default value is mxb=undefined (i.e. infinity).

bs The actual number of bytes of the Object-Body as a function of media-type and encoding-mechanism. This value equals the value send in the Content-Length field. The default value is bs=0.

The discrete mapping function is defined as:

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

The maximum of the Q function represents the preferred media-type to be used for transmitting the Object-Body to the client.

Use of content variants and/or this particular negotiation algorthm is not mandatory. However, it is strongly recommended as it can save a significant amount of bandwidth. The hope is that fine decisions will not have to be made, since in most cases the results for different formats will be differ widely, and there will be a clear winner.

> ***Note:*** *The algorithm described does not take into account the cost of any conversion performed by a gateway or proxy. Although this is an important topic, it is considered more important to maintain any gateways or proxy as transparent parties in the transmission between the client and the server.*

9. Basic Access Authentication Scheme

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

The basic authentication scheme is based on the model that the client must authenticate itself using a user-ID and a password as a function of the origin server, and the realm on that server. The server will service the request only if it can validate the user-ID and password in the realm where the resource identified by the URI is located. The protected resources on the server can be divided into a set of realms which can use different password files, so that different realms can be accessible by different users.

A typical example of an authenticated request has one of the following two profiles:

- The client requests a document *without* sending an Authorization header field
- The server responds with a "401 Unauthorized" status code as described in Section 6.3.3 and an WWW-Authenticate header field
- The client knows already a valid user-ID and a password for the realm indicated by the WWW-Authenticate header field or it prompts the user.
- The clients generates a new request with an Authorization header field
- The server replies by sending the requested Object to the client.

In this example, the client has no authorization information prior to the initial request. In the next example, the client does have such information due to a previous request within the realm indicated by the server. The example would then look like:

- The client requests a document *with* an Authorization header field
- The server replies by sending the requested Object to the client.

This specification of the HTTP protocol allows other authentication schemes to be implemented using either the same frame as the basic scheme or additional header fields. However, these can not be assumed to be generally accepted by applications conforming to this specification.

The client is encouraged to only use the second approach after it has verified that the requested URI is pointing to a resource within the realm where the content of the authorization header field is valid for accessing the URI. If no realm has been indicated then the second approach should be used only when the realm is known from previous requests.

As described in Section 1.2, this specification does not require that the connection be closed after each request. However, when HTTP is used on top of TCP, it is recommended that the connection be closed between the first and second client request in an authenticated request.

Proxies must be completely transparent in the basic access authentication scheme. That is, they must forward the WWW-Authenticate and Authorization headers untouched. If a proxy wants to authenticate a client before a request is forwarded to the server, it can be done using the Proxy-Authenticate and Proxy-Authorization headers.

10. Registration Authority

The HTTP Registration Authority is responsible for maintaining lists of:

- Authorization schemes (as described in Section 9)
- Common method semantics for a HTTP request and response
- Data format names (as MIME Content-Types or Internet Media Types)
- Data encoding names (as MIME Content-Encoding)

It is proposed that the Internet Assigned Numbers Authority [15] or their successors take this role.

11. Security Considerations

This section is meant to inform application developers, information providers, and users of the security limitations in HTTP/1.0 as described by this document. The discussion does not include definitive solutions to the problems revealed, though it does make some suggestions for reducing security risks.

11.1 Authentication of Clients

As mentioned in Section 9, the Basic Authentication scheme used in HTTP/1.0 is not considered to be a secure method of user authentication, nor does it prevent the Object-Body from being transmitted in clear text across the physical network used as the carrier. The protocol does allow for additional authentication schemes and encryption mechanisms to be employed to increase the security level.

11.2 Idempotent Methods

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

In particular, the convention has been established that the GET and HEAD methods should never have the significance of taking an action. The link "click here to subscribe"—causing the reading of a special "magic" document—is open to abuse by others making a link "click here to see a pretty picture". These methods should be considered "safe" and should not have side effects. This allows the client software to represent other methods (such as POST, PUT and DELETE) in a special way, so that the user is aware of the fact that an action is being requested.

11.3 Abuse of Server Log Information

A server is in the position to save personal data about information requested by readers. This information is clearly confidential in nature and its handling may be constrained by law in certain countries. Server providers shall ensure that such material is not distributed without the permission of any individuals that are identifiable by the published results.

Two header fields are worth special mention in this context: Referer and From. The Referer field allows reading patterns to be studied and reverse links drawn. Although it can be very useful, its power can be abused if user details are not separated from the information contained in the Referer. Even when the personal information has been removed, the Referer field may have indicated a secure document's URI, whose revelation would itself be a breach of security.

The information sent in the From field might conflict with the user's privacy interests or their site's security policy, and hence it should not be transmitted without the user being able to disable, enable, and modify the contents of the field prior to a request.

12. Acknowledgments

This specification makes heavy use of the augmented BNF and generic constructs defined by David H. Crocker for RFC 822 [7]. Similarly, it reuses the Content-Type definitions provided by Nathaniel Borenstein and Ned Freed for MIME [4]. We hope that their inclusion in this specification will help reduce past confusion over the relationship between HTTP/1.0 and Internet mail.

The HTTP protocol has evolved considerably over the past three years. It has benefited from a large and active developer community—the many people who have participated on the *www-talk* mailing list—and it is that community which has been most responsible for the success of HTTP and of the World-Wide Web in general. Marc Andreessen, Dan Connolly, Ari Luotonen, Rob McCool, Dave Raggett, Tony Sanders, and Marc VanHeningen deserve special recognition for their efforts in defining aspects of the protocol for early versions of this specification. Bob Denny assisted in proof-reading the specification and performing sanity-checks as it was being rewritten.

13. References

[1] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Alberti. "The Internet Gopher Protocol: A distributed document search and retrieval protocol." RFC 1436, University of Minnesota, <URL:http://ds.internic.net/rfc/rfc1436.txt>, March 1993.

[2] T. Berners-Lee. "Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web." RFC 1630, CERN, <URL:http://ds.internic.net/rfc/rfc1630.txt>, June 1994.

[3] T. Berners-Lee, L. Masinter, and M. McCahill. "Uniform Resource Locators (URL)." Internet-Draft (work in progress), CERN, Xerox PARC, University of Minnesota, <URI: http://ds.internic.net/internet-drafts/draft-ietf-uri-url-08.txt>, October 1994.

[4] N. Borenstein and N. Freed. "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies." RFC 1521, Bellcore, Innosoft, <URL:http://ds.internic.net/rfc/rfc1521.ps>, September 1993.

[5] K. Moore. "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text." RFC 1522, University of Tennessee, <URL:http://ds.internic.net/rfc/rfc1522.txt>, September 1993.

[6] R. Braden. "Requirements for Internet hosts - application and support." STD 3, RFC 1123, IETF, <URL:http://ds.internic.net/rfc/rfc1123.txt>, October 1989.

[7] D. H. Crocker. "Standard for the Format of ARPA Internet Text Messages." STD 11, RFC 822, UDEL, <URL:http://ds.internic.net/rfc/rfc822.txt>, August 1982.

[8] F. Davis, B. Kahle, H. Morris, J. Salem, T. Shen, R. Wang, J. Sui, and M. Grinbaum. "WAIS Interface Protocol Prototype Functional Specification." (v1.5), Thinking Machines Corporation, <URL:ftp://quake.think.com/pub/wais/doc/protspec.txt>, April 1990.

[9] R. T. Fielding. "Relative Uniform Resource Locators." Internet-Draft (work in progress), UC Irvine, <URL:http://ds.internic.net/internet-drafts/draft-ietf-uri-relative-url-02.txt>, November 1994.

[10] M. Horton and R. Adams. "Standard for interchange of USENET messages." RFC 1036 (Obsoletes RFC 850), AT&T Bell Laboratories, Center for Seismic Studies, <URL:http://ds.internic.net/rfc/rfc1036.txt>, December 1987.

[11] B. Kantor and P. Lapsley. "Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News." RFC 977, UC San Diego, UC Berkeley, <URL:http://ds.internic.net/rfc/rfc977.txt>, February 1986.

[12] J. Postel. "Simple Mail Transfer Protocol." STD 10, RFC 821, USC/ISI, <URL:http://ds.internic.net/rfc/rfc821.txt>, August 1982.

[13] J. Postel. "Media Type Registration Procedure." RFC 1590, USC/ISI, <URL:http://ds.internic.net/rfc/rfc1590.txt>, March 1994.

[14] J. Postel and J. K. Reynolds. "File Transfer Protocol (FTP)." STD 9, RFC 959, USC/ISI, <URL:http://ds.internic.net/rfc/rfc959.txt>, October 1985.

[15] J. Reynolds and J. Postel. "Assigned Numbers." STD 2, RFC 1700, USC/ISI, <URL:http://ds.internic.net/rfc/rfc1700.txt>, October 1994.

[16] T. Berners-Lee, D. Connolly, et al. "HyperText Markup Language Specification – 2.0." Internet-Draft (work in progress), CERN, HaL Computer Systems, <URL:http://www.ics.uci.edu/pub/ietf/html/>, November 1994.

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

14. Authors Addresses

Tim Berners-Lee
Director, W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.
Tel: +1 (617) 253 9670
Fax: +1 (617) 258 8682
Email: timbl@w3.org

Roy T. Fielding
Department of Information and Computer Science
University of California
Irvine, CA 92717-3425, U.S.A.
Tel: +1 (714) 856-7308
Fax: +1 (714) 856-4056
Email: fielding@ics.uci.edu

Henrik Frystyk Nielsen
World-Wide Web Project
CERN,
1211 Geneva 23, Switzerland
Tel: +41 (22) 767 8265
Fax: +41 (22) 767 8730
Email: frystyk@w3.org

Appendices

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

A. Tolerant Applications

While it may be appropriate for testing applications to verify full conformance to this specification, it is recommended that operational applications be tolerant of deviations. This appendix mentions the most important topics where tolerance is recommended.

A.1 Request-Line, Status-Line, and Header Fields

Clients should be tolerant in parsing the StatusLine and servers tolerant when parsing the RequestLine. In particular, they should accept any amount of SP and HTAB characters between fields, even though only a single SP is specified.

Line terminators for HTTP-header fields should be the sequence CRLF. However it is recommended that applications, when parsing such headers, also recognize a single LF as a line terminator and ignore any leading CR.

Any HTTP-Header which is not recognized should be ignored by operational applications except when listed in the Mandatory field, as described in Section 4.3.4.

It is recommended that servers use URIs free of “variant” characters (whose representation differs in some of the national variant character sets), punctuation characters, and spaces. This will make URIs easier to handle by humans when the need arises (such as for debugging or transmission through non hypertext systems).

If a Content-Type header is not present in a Object-Header, the preferred action is to try and guess the media-type by looking at the first part of the data in the Object-Body. This document does not provide any algorithm for divining the media-type. If the media-type cannot be found by guessing, the application can either look at the file suffix (if a URL is available) or can default to “application/octet-stream”.

A.2 Object Body

As it is not required to convert any Object-Body to a canonical form regardless of the media-type when transmitted using HTTP, the client must be very tolerant when parsing Object-Body's of type "`text/*`". It is recommended that the following terminators be considered as line separators:

CR, LF, CRLF

If the line terminator is CRLF, the CR should be ignored.

A.3 Backward Compatibility

Servers should be capable of both accepting Simple-Request's and generating Simple-Response's. Many dedicated clients will find the Simple-Request sufficient for their purposes, even though it is generally recommended to use a Full-Request instead. For more general purpose client applications, it is recommend the client be able to generate a Simple-Request, and understand a Simple-Response, in order to communicate with native HTTP/0.9 servers.