Network Working Group
INTERNET-DRAFT
<draft-fielding-http-spec-00.ps>
Expires May 28, 1995

T. Berners-Lee
R. T. Fielding
H. Frystyk Nielsen
November 28, 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, 8] or name (URN), for indicating the object on which a method is to be applied. In addition, messages are passed in a format similar to that used by Internet Mail [6] 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 [11], NNTP [10], FTP [13], Gopher [1], and WAIS [7]. HTTP/1.0 is designed to allow such gateways, via proxy agents, 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 [14], 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, 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 from which a given resource originates.

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 agent* 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 agent with a local cache of server responses — frequently requested objects can be serviced from the cache rather than from the origin server. A *proxy server* is a proxy agent that also acts as an origin server.

gateway
A proxy agent 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 the augmented Backus-Naur Form (BNF) of RFC 822 [6]. 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 slash ("/") are alternatives, e.g. "foo / bar" will accept foo or bar.

(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 (","). This makes the usual form of lists very easy; a rule such as "(element *("," 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

```
                                                         ; ( Octet in Decimal)
    OCTET              = <any 8-bit character>           ; ( 0-255 )
    CHAR               = <any US-ASCII character>        ; ( 0 - 127 ) [16]
    HIGHALPHA          = <any US-ASCII uppercase letter> ; ( 65 - 90 )
    LOWALPHA           = <any US-ASCII lowercase letter> ; ( 97 -122 )
    ALPHA              = HIGHALPHA / LOWALPHA
    DIGIT              = <any US-ASCII digit “0” through “9”> ; ( 48 - 57 )
    CTL                = <any US-ASCII control character ; ( 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 quote mark>           ; ( 34 )
    CRLF               = CR LF
    LWSP-char          = SP / HTAB                       ; semantics = SP
    linear-white-space= 1*( [CRLF] LWSP-char)            ; semantics = SP
                                                         ; CRLF => folding

    tspecials          = “(“ / “)” / “<“ / “>” / “@”     ; Must be in quoted-string,
                       /  “,” / “;” / “:” / “\”/ <">     ; to use within parameter values
                       /  “/” / “[“ / “]” / “?” / “=”

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

    quoted-string      = <"> *(qtext) <">                ; Regular qtext or quoted chars.
    qtext              = <any CHAR excepting <">, """ & CR, ; => may be folded
                         & including linear-white-space>

    phrase             = 1*word                          ; Sequence of words
    word               = token / quoted-string
    delimiters         = tspecials / linear-white-space

    text               = <any CHAR, including bare CR,   ; for generic field contents
                         but NOT including CRLF>
```

3. HTTP Message

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

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

Full-Request and Full-Response use the generic message format of RFC 822 [6] for transferring data 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 ]
```

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 [6]. 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 )
                  [ CRLF 1*LWSP-char field-value ]

field-content   = <the US-ASCII characters making up the field-value and consisting
                  of combinations of token, tspecials, and quoted-string,
                  or else consisting of text>
```

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. However, because HTTP is rarely seen by human eyes, use of comments is discouraged.

```
comment         = “(“ *(ctext / comment) “)”
ctext           = <any CHAR excluding “(“, “)”, “‘ & CR,
                  & including linear-white-space>
```

3.2 Object Body

The data object (if any) sent with an HTTP/1.0 request or response is in a format and encoding defined by the Object-Header fields, the default being of type “`plain/text`” with “`binary`” encoding. Note that, while all other information in the request or response is in US-ASCII with lines delimited by CRLF, the Object-Body may contain 8-bit binary data.

```
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 content-length when the data object 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, [6]) and the Multipurpose Internet Mail Extensions (MIME, [4]) to allow objects 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 [5] (an update to RFC 822 [6]). The second format is in common use today, but is based on the obsolete RFC 850 [9] 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"
```

It should be noted that the 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 Content Types

HTTP uses Internet Media Types [12], 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 content 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 when the user agent has indicated acceptability of the multipart type in addition to the content types of each constituent body part.

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, the boundary parameter must always be enclosed in quotes (<">) and 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           ; taken from Content-type field.
                                                ; There must be no space
                                                ; between “--” and boundary.

close-delimiter  = “--” boundary “--” CRLF      ; Again, no space by “--”,

discard-text     = *(*text CRLF)                ; to be ignored upon receipt.

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 which can be identified by a URI.

4.2.1.1 Multipart/alternative

The "`multipart/alternative`" content-type is used in MIME to send content-type variants of a single entity when the receiver's capabilities are not known. This is not the case with HTTP. Multipart/alternative can be used to provide metainformation of many instances of an object, as in the case of a redirection response. This allows, for example, URIs of a set of instances of an object to be returned by a name server.

4.2.1.2 Multipart/mixed

The "`multipart/mixed`" media type can be used when the first body-part contains references to other parts which the server 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 the user agent or a caching proxy.

4.2.1.3 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 type would be appropriate for situations where simultaneous presentation is an important aspect of the information, such as for audio-annotated slides and movies.

4.2.1.4 Other Multipart Types

The other multipart types defined by IANA [14] 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.

4.3 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 appropriate) and should be implemented by future HTTP/1.0 clients and servers. These headers apply only to the message being transmitted.

```
General-Header   = Date / Forwarded / Message-ID / MIME-Version
```

4.3.1 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. Note that only user agents and origin servers can originate a message. However, 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 data object. This has been changed to reflect actual (and proper) usage.

4.3.2 Forwarded

The Forwarded header is to be used by gateways and proxy agents 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>
```

In the following example, a message is sent from a client on *ptsun00.cern.ch* to a server at *www.ics.uci.edu* port 80 via an intermediate HTTP proxy agent at *info.cern.ch* port 8000. The request received by the server at *www.ics.uci.edu* will 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 gateway or proxy agent that has forwarded the message. It is strongly recommended that proxy agents 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.

4.3.3 Message-ID

The Message-ID field in HTTP is identical to that used by Internet Mail and USENET messages, as defined in [9]. 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*.

Note: Unlike the URI-header field, the Message-ID does not give a way of accessing the resource, so the Message-ID cannot be used to refer to that resource. However, in the case of USENET news articles, the Message-ID may in fact be used within a URI for retrieval of the message via NNTP.

4.3.4 MIME-Version

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. It is important to note, however, that this should not be considered as an indication of full compliance with MIME as it has been defined in [4]. Gateways are responsible for ensuring this compliance (where possible) when exporting HTTP messages to strict MIME environments.

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

MIME version "`1.0`" is the default for use in HTTP/1.0.

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. However, applications using HTTP/0.9 should be upgraded to HTTP/1.0.

```
Request         = Simple-Request / Full-Request

Simple-Request  = "GET" 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 LWSP-chars. No CR or LF are allowed except in the final CRLF sequence.

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

5.2 Method

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

```
Method          = "GET" / "HEAD" / "PUT" / "POST"   ; case-sensitive
                / "DELETE" / 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 object 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 URI, as this can change dynamically. The set of common methods for HTTP/1.0 are specified 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 data 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.

If supported by the server, the GET method can be used for text searches. The URI used in the request is the URI identifying the object, suffixed by a "?" character and the text to be searched as specified by RFC 1630 [2]. If the object being searched is in fact itself the result of a search (i.e. the URI contains a "?"), then those search terms are first stripped off, so the search is performed on the original object. The body part of the response shall, except in the case of error situations, always be the result of the search.

5.2.2 HEAD

The HEAD method is similar to GET but the server may not return any Object-Body in the response. This method can be used for obtaining metainformation about the object identified by the URI such as the last-modified date for use in caching schemes etc.

5.2.3 POST

The POST method is used to request that the origin server accept the enclosed object as a new subordinate of the object given by the request URI. Depending upon the URI in question, this new object may be a document annotation, a message intended for a bulletin board or newsgroup, an additional item to be appended to a list, or a block of data (usually a form) to be processed via a gateway program.

The POST method is designed as a generic way to allow a user agent to add more information to a server without insisting that it be stored under a specific URI. Where possible and appropriate, a new URI will be allocated by the origin server and returned to the client for future reference to that object. Note that this is not the case when the HTTP protocol is used by user agent posting a SMTP mail message through a proxy agent. 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 POST method is designed to allow a uniform function to cover

- Annotation of existing documents;
- Posting a message to a bulletin board topic, newsgroup, mailing list, or similar group of articles;
- Providing form data to a gateway program;
- Extending a document during authorship.

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 be surprised if a link is not immediately (or never) created.

If the URI does not refer to a gateway, the origin server is requested to see to the storage of the new object. That is, the origin server does not have to store it permanently, but should (if the method was successful) return a URI by which the object can be referenced and later retrieved. The semantics of this method imply nothing of any undertakings by the origin server to maintain the availability of the object.

If the user agent includes a URI-header in the request, the server should treat that URI as advisory only. It may store the object under a different URI, in which case the origin server must inform the user agent of the new URI via a URI-header in the response.

5.2.3.1 Returned Object Headers

The POST method shall return a set (possibly empty) of object headers for the newly posted object. The server may return the entire metainformation for the object (as in the HEAD method), or a subset of it. The posted object should not be returned in the response, though the response Object-Body may include a status or error message which references the new object.

5.2.3.2 Link Type

The Link header field described in Section 7.15 may be specified by explicitly giving a (possibly reverse) link in the Object-Header of the linked object. If one or more Link's are contained in the Object-Header, then this information should be used. If no such link or links are specified, then the server should generate a link. The link type in this case is determined by the server.

Note: The server may perform other operations as a result of the new object being added: lists and indexes might be updated, for example. However, no mandatory operation is imposed on the origin server.

5.2.3.3 Submission

When articles are submitted, the analogy of being added to a body of knowledge by being linked is close. When a form is submitted, this can be done with POST, though in this case side-effects will be expected.

5.2.4 PUT

The PUT method specifies that the data in the Object-Body of the request is to be stored under the supplied URI. If the URI points to an already existing object, 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 object, and that URI is capable of being defined as a new object from the requesting user agent, the origin server can place the object at that URI.

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

5.2.5 DELETE

The DELETE method requests that the server delete the object corresponding to the given URI. After a successful DELETE request, the URI becomes invalid for any future references. This method can at all times be overridden by human interaction or other means on the remote server, so the client cannot be guaranteed that the operation has been carried out, even if the status code returned from the server indicates that the action has been completed successfully.

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 obey 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 object upon which to apply the request. It can either be a URL (Uniform Resource Locator [3]) or a URN (Uniform Resource Name). At the time of this writing, no suitable naming system exists for URNs, but this protocol will accept such names so long as they do not include whitespace and are distinguishable from the existing URL name space.

```
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

In the case of a client sending a request through a proxy or gateway, the protocol and host name must be explicitly declared.

Note: The URI should be encoded using the escaping scheme described in [2]. Note that escaping is permitted for any character but the default escaping should be limited to reserved characters and characters which are considered unsafe.

Note: The part of an URL after the host name and optional port number is completely opaque to the client: The client may make no deductions, such as the file type, about the object based on the URL pointing to it.

5.5 Request Header Fields

The request header fields provide the possibility for the client to pass additional information about the request and the client itself to the server. All header fields are RFC 822 conforming and are optional for the client to transmit. However, the client is strongly encouraged to specify as many as possible.

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

These header fields are explained in the following subsections. 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-cormance to the protocol. See Section 6.4.1 for a description of what should be done for requests passed through a gateway or proxy.

Note: Some current proxiy 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.

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. The format of this field is an absolute date and time in the HTTP-date format of Section 4.1:

```
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 possibly followed by a GET request.

5.5.3 Pragma

Pragma directives should be understood by servers to which they are relevant, e.g. a proxy. They give the client the possibility of affecting the default behavior of the server. The syntax is the same as for other multiple-value fields in HTTP, however the current specification only contains one valid parameter:

```
Pragma            = "Pragma" ":" 1*#pragma-param

pragma-param      = "no-cache" / extension-pragma
extension-pragma  = token
```

When the "no-cache" parameter 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.

Note: Pragmas should 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-param 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 requesting a URI on an HTTP server.

```
Authorization          = "Authorization" ":" (("Basic" uu-encoded)
                         /  (extension-scheme extension-encrypted))

uu-encoded             = <UU encoding of userid-password>
userid-password        = [word] ":" [word]

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

The format of this field is in extensible form so that it can handle other and more advanced encryption schemes. The first word is a specification of the authorization system in use followed by the encrypted version of the User-ID and the password separated by a ":" in clear text. It is important to note that the Basic scheme only provides a low-level security similar to the methods used by unmodified FTP, Telnet, etc.

Note: The specification indicates that the userid-password can be the sequence ":" as both the User-ID and the password are optional, however it is not the recommended usage.

5.5.5 Proxy-Authorization

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

```
Proxy-Authorization     = "Proxy-Authorization" ":" (("Basic" uu-encoded)
                        / (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 bad 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 parsed relative to the URI of the object of the request.

Note: The Referer field allows reading patterns to be studied, and reverse links drawn, however, the field may in fact contain a secure URI, whose revelation itself can be considered as a breach of security. It is therefore strongly recommended that the user be able to disable and enable this field prior to a request.

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 gateway or proxy agent, 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

This field contains a list of representation schemes (Content-Type metainformation tokens) which are accepted in the response to the request. The set given may of course vary from request to request from the same client.

The field may be wrapped onto several lines according to RFC 822, and also more than one occurrence of the field is allowed with the significance being the same as if all the entries has been in one field. The format of each entry in the list is:

```
Accept         = "Accept" ":" 1#( ("*" / type) "/" ("*" / subtype) *(";" accept-param) )
accept-param   = ("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 content type and mxb is the maximum accepted size of the Object-Body in number of octets. The definition does not prohibit duplicate accept-param's, but leaves the interpretation undefined. See Section 8 for a description of the negotiation algorithm and penalty model. A quality factor of 0 is equivalent to not sending an accept header field containing the actual content-type. The default values are: q=1 and mxb=infinity.

In order to save time, and also allow clients to receive content 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 field 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 content 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: The client should not try to divine which content types are relevant to send in the request header on behalf of the information given in the URI, for example by looking at the file suffix in a URL. This information is completely opaque to the client.

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 usable reference value. However, this does not prevent the client from internally measuring the response time and optimize the accept header field accordingly by using the quality factor.

5.5.9 Accept-Encoding

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

```
Accept-Encoding  = "Accept-Encoding" ":" 1#encoding-mechanism
encoding-mechanism = "x-compress" / "x-gzip" / extension-encoding
extension-encoding = token
```

Example

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

5.5.10 Accept-Language

The “Accept-Language” field is similar to the accept header field, but it lists the set of natural languages accepted in the response. The language-encoding field is described in Section 7.6. The format of the field is defined as:

```
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: “If you have a Danish version, send it; else if you have an British English version, send it”.

Note: If the server can not serve the request with the language specified or if the languages specified only represent a subset in case of a multi-linguistic data object, 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 any client applications makes the choice of linguistic preferences 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 should, like the Request-Line, consist only of the elements specified separated by LWSP-chars. That is, no CR or LF are allowed except in the final CRLF sequence.

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

6.2 HTTP Version

The HTTP-Version field 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 field contains an 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 or to pass it on to the human user.

```
Status-Code     = 3DIGIT

Reason-Phrase   = *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 content-type is "`text/html`", but "`text/plain`" is also accepted. As for the Reason-Phrase, it is not mandatory for the client to pass any explanation information in the Object-Body to the user.

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 a default 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. The Status-Code of any extension-method is not defined in this document, and no rules are given for introducing additional codes.

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 teh 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 resending 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 data object.

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.

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.

Note: 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.

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 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 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 Error", but has more diagnostic value.

503 Service Unavailable

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 may be alleviated at other times.

504 Gateway Timeout

This is equivalent to "500 Internal 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 Error", but has more diagnostic value.

6.4 Response Header Fields

The response header fields provide the server the possibility to pass additional information about the response which can not be placed in the Status-Line. These header fields are not intended to give information about an Object-Body returned in the response but uniquely about the server itself. The Object-Header fields which should be used for the latter purpose are described in Section 7. The response header fields specified in this document are:

```
Response-Header= Server
               / WWW-Authenticate
               / Proxy-Authenticate
```

Unknown header fields should be considered Object-Header fields.

6.4.1 Server

This field contains information about the server software program used for handling the request. The field is equivalent to the User-Agent field in the request and has the following format:

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

Example:

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

If the request is going through a gateway or proxy, then these applications must not add their data to the list received. Instead, they should use the Forwarded field described in Section 4.3.1.

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 used and the realm in which the requested URI belongs. The syntax is defined as:

```
WWW-Authenticate        = "WWW-Authenticate" ":" ("Basic" realm)
                          / (extension-scheme realm)
realm                   = token
```

The contents of this field is extensible as is the case for the Authorization field in the request header. The first word is a specification of the authorization system in use followed by the realm of the protected URI requested.

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 used and the realm in which the requested URI belongs. The syntax is defined as:

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

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.

7. Object Header Fields

Any Full-Request or Full-Response message 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, but applications are strongly encouraged to specify as many as possible. This document defines the following header fields:

```
Object-Header    = Allow
                 / Content-Length
                 / Content-Type
                 / Content-Encoding
                 / Content-Transfer-Encoding
                 / Content-Language
                 / Expires
                 / Last-Modified
                 / URI-header
                 / Location
                 / Version
                 / Derived-From
                 / Release
                 / 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 the client from trying other methods. However, it is recommended to follow the indications given in this field. If not specified, the default value of the allowed methods equals the total set of methods described in Section 5.2. The format of the field is:

Allow = “Allow” “:” 1#method

Example of use:

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

7.2 Content-Length

This field indicates in number of octets the size of the Object-Body sent to the recipient. The format of the field is

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 indicated the size of the Object-Body to be transferred no matter of the content-type of the object.

Note: Any Content-Length of size greater than or equal to zero is a valid value. The field has no default value.

Note: The meaning of this field is significantly different from the corresponding specification in the MIME specifications where it is an optional field used within the “message/external-body” Content-Type. In HTTP, it should be used whereever possible.

7.3 Content-Type

The format of this field which is also described in Section 4.2 is defined as:

Content-Type = “Content-Type” “:” media-type

An example of the field is

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

Note: All content types defined by MIME including extension tokens are of course valid tokens. The field has no default value.

7.4 Content-Encoding

The Content-Encoding header field, unique to HTTP, is used as a modifier to the content-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. It has the following format:

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

An example of its use is

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

Note: The Content-Encoding field differs from the "Content-Transfer-Encoding" field defined as encoding in the MIME specifications. The purpose of the "Content-Transfer-Encoding" field is "to indicate the type of transformation that has been used in order to represent the object in an acceptable manner for transport". The Content-Encoding field is used to indicate any form of compression mechanism used to decrease the amount of data to be transported. The result after encoding can still 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 should add an explicit "`Content-Transfer-Encoding: binary`" to the message header if one is not already present.

Content-Transfer-Encoding = type ["/" subtype] *(";" parameter)

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.

Note: Multi-linguistic data objects can be described using a list of lang-dia codes. This document does not specify any means to indicates the amount of different natural languages represented in the data object.

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 data objects of type "text".

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.

Note: Data objects 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 such data objects do contain an Expires header field.

7.8 Last-Modified

The Last-Modified field indicates the date and time of when the data object was last modified. The format is an HTTP-date, as described in Section 4.1, and the syntax is:

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

An example of its use is

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

Note: The definition of this header field does not specify what is meant by "modification", as this depends on the actual server implementation. In particular, the field is not connected to any date and time indications given by the operation system on which the server application is running.

Note: If the expires header field is not present in the Object-Header, then this field can be used by a cache mechanism to estimate when the object expires.

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 object can be retrieved 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"
/ extension-vary

extension-vary = token

If the URI is used to refer to a set of variants, then the dimensions in which the variants may 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 data objects which varies in content-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 data object 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 location URI is allowed.

```
URI-header      = "Location" ":" URI
```

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" ":" 1*DIGIT *( "." 1*DIGIT )
```

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 data object. The definition of the field must therefore be similar to the Version field described in Section 7.11:

```
Derived-From    = "Derived-From" ":" 1*DIGIT *( "." 1*DIGIT )
```

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 data object. 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 Release

This field is a string oriented value containing a code representing the publisher's local versioning of the document, for example relative to a local code management system. The definition of this field contains no explicit semantics as publishers often will have quite complex version information containing hidden local semantics.

```
Release         = "Release" ":" *text
```

Note: This field is significantly different from the Version field defined in Section 7.11 which should be used together with the PUT method. The Release does not have an implied interpretation and can be used freely as metainformation for any Object-Body. The value should be considered opaque.

7.14 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 data object, but the "Subject" field is defined by the originator. The field is to be considered isomorphic with the <TITLE> element in HTML [15].

7.15 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 *1(";" "REL" "=" relation) )
relation       = "UseIndex" / "UseGlossary" / "Contents" / "Next" / "Previous"
                 / "Parent" / "BookMark" / "Made" / "Help"
```

The reader is referred to the HTML specification [15] 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 data object. The maximum value of this set represents the Content-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 data object is rendered into a specific content-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 a data 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 data object in a specific Content-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 Content-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 Content-Type and Content-Encoding. 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 content-type to be used for transmitting the Object-Body to the client.

Note: It is not mandatory for the server application to actually do the mapping and to determine the maximum value and hence the optimal Content-Type to return to the client. 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, as in most cases the results for different formats will be very different, 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 as 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 it not generally true, the basic access authentication scheme should be used accordingly. However, it must be implemented in order for an application to be HTTP 1.0 conforming.

The basic authentication scheme is based on the model that the client must authenticate itself using a user-ID and a password. The server will serve the request only if the server can validate the authentication sent by the client, for example by using a password file. 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 with the requested data resource.

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 with the requested data resource.

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.

Note: The client is encouraged only to 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.

Note: 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.

Note: Gateways and proxy agents shall be completely transparent in the basic access authentication scheme. That is, they must forward the WWW-Authenticate header and the Authorization header untouched. In case a gateway or a proxy wants to authenticate a client before a request is forwarded to the server, it can be done using the scheme presented above but not in the same authenticated request as the one that the gateway or proxy is forwarding. That is, it must be done using an explicit request from the client to the gateway or proxy.

10. Registration Authority

The HTTP Registration Authority is responsible for maintaining lists of:

- Authorization schemes (see Authorization: field above)
- Common method semantics
- 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 [14] or their successors take this role.

11. Security Considerations

The following section reveals some of the security aspects of HTTP. They are meant to inform application providers of the security limitations in HTTP described by this document, but do not suggest definitive solutions to the problems discussed.

11.1 Authentication of Clients

As mentioned in Section 9, the Basic Authentication scheme used in HTTP is not to be considered as a secure way of protecting information on servers, nor does it prevent the Object-Body from being transmitted in clear text across the physical network used as the carrier for HTTP. The protocol allows through the definition of the relevant header field additional authentication schemes 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 must be established that the GET and HEAD methods 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 [6]. 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. Ari Luotonen, Rob McCool, 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, <URL: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] R. Braden. "Requirements for Internet hosts - application and support." STD 3, RFC 1123, IETF, <URL:http://ds.internic.net/rfc/rfc1123.txt>, October 1989.

[6] 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.

[7] 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.

[8] R. Fielding. "Relative Uniform Resource Locators." Internet-Draft (work in progress), UC Irvine, <URL:http://ds.internic.net/internet-drafts/draft-ietf-uri-relative-url-01.txt>, October 1994.

[9] 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.

[10] 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.

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

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

[13] 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.

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

[15] 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.

[16] 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. Format of a uuencoded file

This section has been taken from the man page on uuencode on SunOS.

Files output by *uuencode* consist of a header line, followed by a number of body lines, and a trailer line. *uudecode* will ignore any lines preceding the header or following the trailer. Lines preceding a header must not, of course, look like a header.

The header line is distinguished by having the first 6 characters "begin ". The word begin is followed by a mode (in octal), and a string which names the remote file. SP characters separate the three items in the header line.

The body consists of a number of lines, each at most 62 characters long (including the trailing LF). These consist of a character count, followed by encoded characters, followed by a LF. The character count is a single printing character, and represents an integer, the number of bytes the rest of the line represents. Such integers are always in the range from 0 to 63 and can be determined by subtracting the character SP (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a SP to make the characters printing. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra garbage will be included to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII SP.

The trailer line consists of end on a line by itself.

B. Server tolerance of bad clients

Whilst it is seen appropriate for testing parsers to check full conformance to this specification, it is recommended that operational parsers be tolerant of deviations.

In particular, lines should be regarded as terminated by the Line Feed, and the preceding Carriage Return character ignored.

Any HTTP Header Field Name which is not recognized should be ignored in operational parsers.

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 (such as debugging, or transmission through non hypertext systems) arises.

C. Client tolerance of bad servers

Servers not implementing the specification as written are not HTTP compliant. Servers should always be made completely compliant. However, clients should also tolerate deviant servers where possible.

C.1 Back compatibility

In order that clients using the HTTP protocol should be able to communicate with servers using the protocol originally implemented in the W3 data model, clients should tolerate responses which do not start with a numeric version number and response codes.

In this case, they should assume that the rest of the response is a document body in type text/html.

C.2 White space

Clients should be tolerant in parsing response status lines, in particular they should accept any sequence of white space (SP and HTAB) characters between fields.

Lines should be regarded as terminated by a line feed (LF), and the preceding carriage return (CR) character ignored.