HTTP/1.1 Messaging
draft-ietf-httpbis-messaging-06

Abstract

   The Hypertext Transfer Protocol (HTTP) is a stateless application-
   level protocol for distributed, collaborative, hypertext information
   systems. This document specifies the HTTP/1.1 message syntax,
   message parsing, connection management, and related security
   concerns.

   This document obsoletes portions of RFC 7230.

Editorial Note

   This note is to be removed before publishing as an RFC.

   Discussion of this draft takes place on the HTTP working group
   mailing list (ietf-http-wg@w3.org), which is archived at
   <https://lists.w3.org/Archives/Public/ietf-http-wg/>.

   Working Group information can be found at <https://httpwg.org/>;
   source code and issues list for this draft can be found at
   <https://github.com/httpwg/http-core>.

   The changes in this draft are summarized in Appendix D.7.

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at https://datatracker.ietf.org/drafts/current/.

   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."

This Internet-Draft will expire on May 7, 2020.

Copyright Notice

Table of Contents

1. Introduction

   The Hypertext Transfer Protocol (HTTP) is a stateless application-
   level request/response protocol that uses extensible semantics and
   self-descriptive messages for flexible interaction with network-based
   hypertext information systems. HTTP is defined by a series of
   documents that collectively form the HTTP/1.1 specification:

   o "HTTP Semantics" [Semantics]

   o "HTTP Caching" [Caching]

   o "HTTP/1.1 Messaging" (this document)

   This document defines HTTP/1.1 message syntax and framing
   requirements and their associated connection management. Our goal is
   to define all of the mechanisms necessary for HTTP/1.1 message
   handling that are independent of message semantics, thereby defining
   the complete set of requirements for message parsers and message-
   forwarding intermediaries.

This document obsoletes the portions of RFC 7230 related to HTTP/1.1
messaging and connection management, with the changes being
summarized in Appendix C.2. The other parts of RFC 7230 are
obsoleted by "HTTP Semantics" [Semantics].

1.1. Requirements Notation

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

Conformance criteria and considerations regarding error handling are
defined in Section 3 of [Semantics].

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF)
notation of [RFC5234], extended with the notation for case-
sensitivity in strings defined in [RFC7405].

It also uses a list extension, defined in Section 12 of [Semantics],
that allows for compact definition of comma-separated lists using a
'#' operator (similar to how the '*' operator indicates repetition).
Appendix A shows the collected grammar with all list operators
expanded to standard ABNF notation.

As a convention, ABNF rule names prefixed with "obs-" denote
"obsolete" grammar rules that appear for historical reasons.

The following core rules are included by reference, as defined in
[RFC5234], Appendix B.1: ALPHA (letters), CR (carriage return), CRLF
(CR LF), CTL (controls), DIGIT (decimal 0-9), DQUOTE (double quote),
HEXDIG (hexadecimal 0-9/A-F/a-f), HTAB (horizontal tab), LF (line
feed), OCTET (any 8-bit sequence of data), SP (space), and VCHAR (any
visible [USASCII] character).

The rules below are defined in [Semantics]:

```
   BWS           = <BWS, see [Semantics], Section 11.1>
   OWS           = <OWS, see [Semantics], Section 11.1>
   RWS           = <RWS, see [Semantics], Section 11.1>
   absolute-URI  = <absolute-URI, see [RFC3986], Section 4.3>
   absolute-path = <absolute-path, see [Semantics], Section 2.4>
   authority     = <authority, see [RFC3986], Section 3.2>
   comment       = <comment, see [Semantics], Section 4.2.3.3>
   field-name    = <field-name, see [Semantics], Section 4.1>
   field-value   = <field-value, see [Semantics], Section 4.2>
   obs-text      = <obs-text, see [Semantics], Section 4.2.3.2>
   port          = <port, see [RFC3986], Section 3.2.3>
   query         = <query, see [RFC3986], Section 3.4>
   quoted-string = <quoted-string, see [Semantics], Section 4.2.3.2>
   token         = <token, see [Semantics], Section 4.2.3.1>
   uri-host      = <host, see [RFC3986], Section 3.2.2>
```

2. Message

2.1. Message Format

An HTTP/1.1 message consists of a start-line followed by a CRLF and a
sequence of octets in a format similar to the Internet Message Format
[RFC5322]: zero or more header fields (collectively referred to as
the "headers" or the "header section"), an empty line indicating the
end of the header section, and an optional message body.

```
   HTTP-message   = start-line CRLF
                    *( header-field CRLF )
                    CRLF
                    [ message-body ]
```

A message can be either a request from client to server or a response
from server to client. Syntactically, the two types of message
differ only in the start-line, which is either a request-line (for
requests) or a status-line (for responses), and in the algorithm for
determining the length of the message body (Section 6).

```
   start-line     = request-line / status-line
```

In theory, a client could receive requests and a server could receive
responses, distinguishing them by their different start-line formats.
In practice, servers are implemented to only expect a request (a
response is interpreted as an unknown or invalid request method) and
clients are implemented to only expect a response.

Although HTTP makes use of some protocol elements similar to the
Multipurpose Internet Mail Extensions (MIME) [RFC2045], see
Appendix B for the differences between HTTP and MIME messages.

2.2. Message Parsing

   The normal procedure for parsing an HTTP message is to read the
   start-line into a structure, read each header field into a hash table
   by field name until the empty line, and then use the parsed data to
   determine if a message body is expected. If a message body has been
   indicated, then it is read as a stream until an amount of octets
   equal to the message body length is read or the connection is closed.

   A recipient MUST parse an HTTP message as a sequence of octets in an
   encoding that is a superset of US-ASCII [USASCII]. Parsing an HTTP
   message as a stream of Unicode characters, without regard for the
   specific encoding, creates security vulnerabilities due to the
   varying ways that string processing libraries handle invalid
   multibyte character sequences that contain the octet LF (%x0A).
   String-based parsers can only be safely used within protocol elements
   after the element has been extracted from the message, such as within
   a header field-value after message parsing has delineated the
   individual fields.

   Although the line terminator for the start-line and header fields is
   the sequence CRLF, a recipient MAY recognize a single LF as a line
   terminator and ignore any preceding CR.

   Older HTTP/1.0 user agent implementations might send an extra CRLF
   after a POST request as a workaround for some early server
   applications that failed to read message body content that was not
   terminated by a line-ending. An HTTP/1.1 user agent MUST NOT preface
   or follow a request with an extra CRLF. If terminating the request
   message body with a line-ending is desired, then the user agent MUST
   count the terminating CRLF octets as part of the message body length.

   In the interest of robustness, a server that is expecting to receive
   and parse a request-line SHOULD ignore at least one empty line (CRLF)
   received prior to the request-line.

   A sender MUST NOT send whitespace between the start-line and the
   first header field. A recipient that receives whitespace between the
   start-line and the first header field MUST either reject the message
   as invalid or consume each whitespace-preceded line without further
   processing of it (i.e., ignore the entire line, along with any
   subsequent lines preceded by whitespace, until a properly formed
   header field is received or the header section is terminated).

   The presence of such whitespace in a request might be an attempt to
   trick a server into ignoring that field or processing the line after
   it as a new request, either of which might result in a security
   vulnerability if other implementations within the request chain

   interpret the same message differently. Likewise, the presence of
   such whitespace in a response might be ignored by some clients or
   cause others to cease parsing.

   When a server listening only for HTTP request messages, or processing
   what appears from the start-line to be an HTTP request message,
   receives a sequence of octets that does not match the HTTP-message
   grammar aside from the robustness exceptions listed above, the server
   SHOULD respond with a 400 (Bad Request) response.

2.3. HTTP Version

   HTTP uses a "<major>.<minor>" numbering scheme to indicate versions
   of the protocol. This specification defines version "1.1".
   Section 3.5 of [Semantics] specifies the semantics of HTTP version
   numbers.

   The version of an HTTP/1.x message is indicated by an HTTP-version
   field in the start-line. HTTP-version is case-sensitive.

     HTTP-version = HTTP-name "/" DIGIT "." DIGIT
     HTTP-name   = %s"HTTP"

   When an HTTP/1.1 message is sent to an HTTP/1.0 recipient [RFC1945]
   or a recipient whose version is unknown, the HTTP/1.1 message is
   constructed such that it can be interpreted as a valid HTTP/1.0
   message if all of the newer features are ignored. This specification
   places recipient-version requirements on some new features so that a
   conformant sender will only use compatible features until it has
   determined, through configuration or the receipt of a message, that
   the recipient supports HTTP/1.1.

   Intermediaries that process HTTP messages (i.e., all intermediaries
   other than those acting as tunnels) MUST send their own HTTP-version
   in forwarded messages. In other words, they are not allowed to
   blindly forward the start-line without ensuring that the protocol
   version in that message matches a version to which that intermediary
   is conformant for both the receiving and sending of messages.
   Forwarding an HTTP message without rewriting the HTTP-version might
   result in communication errors when downstream recipients use the
   message sender's version to determine what features are safe to use
   for later communication with that sender.

   A server MAY send an HTTP/1.0 response to an HTTP/1.1 request if it
   is known or suspected that the client incorrectly implements the HTTP
   specification and is incapable of correctly processing later version
   responses, such as when a client fails to parse the version number
   correctly or when an intermediary is known to blindly forward the

   HTTP-version even when it doesn't conform to the given minor version
   of the protocol. Such protocol downgrades SHOULD NOT be performed
   unless triggered by specific client attributes, such as when one or
   more of the request header fields (e.g., User-Agent) uniquely match
   the values sent by a client known to be in error.

3. Request Line

   A request-line begins with a method token, followed by a single space
   (SP), the request-target, another single space (SP), and ends with
   the protocol version.

     request-line   = method SP request-target SP HTTP-version

   Although the request-line grammar rule requires that each of the
   component elements be separated by a single SP octet, recipients MAY
   instead parse on whitespace-delimited word boundaries and, aside from
   the CRLF terminator, treat any form of whitespace as the SP separator
   while ignoring preceding or trailing whitespace; such whitespace
   includes one or more of the following octets: SP, HTAB, VT (%x0B), FF
   (%x0C), or bare CR. However, lenient parsing can result in request
   smuggling security vulnerabilities if there are multiple recipients
   of the message and each has its own unique interpretation of
   robustness (see Section 11.2).

   HTTP does not place a predefined limit on the length of a request-
   line, as described in Section 3 of [Semantics]. A server that
   receives a method longer than any that it implements SHOULD respond
   with a 501 (Not Implemented) status code. A server that receives a
   request-target longer than any URI it wishes to parse MUST respond
   with a 414 (URI Too Long) status code (see Section 9.5.15 of
   [Semantics]).

   Various ad hoc limitations on request-line length are found in
   practice. It is RECOMMENDED that all HTTP senders and recipients
   support, at a minimum, request-line lengths of 8000 octets.

3.1. Method

   The method token indicates the request method to be performed on the
   target resource. The request method is case-sensitive.

     method   = token

   The request methods defined by this specification can be found in
   Section 7 of [Semantics], along with information regarding the HTTP
   method registry and considerations for defining new methods.

3.2. **Request Target**

   The request-target identifies the target resource upon which to apply
   the request. The client derives a request-target from its desired
   target URI. There are four distinct formats for the request-target,
   depending on both the method being requested and whether the request
   is to a proxy.

```
   request-target = origin-form
                  / absolute-form
                  / authority-form
                  / asterisk-form
```

   No whitespace is allowed in the request-target. Unfortunately, some
   user agents fail to properly encode or exclude whitespace found in
   hypertext references, resulting in those disallowed characters being
   sent as the request-target in a malformed request-line.

   Recipients of an invalid request-line SHOULD respond with either a
   400 (Bad Request) error or a 301 (Moved Permanently) redirect with
   the request-target properly encoded. A recipient SHOULD NOT attempt
   to autocorrect and then process the request without a redirect, since
   the invalid request-line might be deliberately crafted to bypass
   security filters along the request chain.

3.2.1. **origin-form**

   The most common form of request-target is the origin-form.

```
   origin-form    = absolute-path [ "?" query ]
```

   When making a request directly to an origin server, other than a
   CONNECT or server-wide OPTIONS request (as detailed below), a client
   MUST send only the absolute path and query components of the target
   URI as the request-target. If the target URI's path component is
   empty, the client MUST send "/" as the path within the origin-form of
   request-target. A Host header field is also sent, as defined in
   Section 5.4 of [Semantics].

   For example, a client wishing to retrieve a representation of the
   resource identified as

```
   http://www.example.org/where?q=now
```

   directly from the origin server would open (or reuse) a TCP
   connection to port 80 of the host "www.example.org" and send the
   lines:

```
   GET /where?q=now HTTP/1.1
   Host: www.example.org
```

followed by the remainder of the request message.

3.2.2. absolute-form

When making a request to a proxy, other than a CONNECT or server-wide
OPTIONS request (as detailed below), a client MUST send the target
URI in absolute-form as the request-target.

```
   absolute-form  = absolute-URI
```

The proxy is requested to either service that request from a valid
cache, if possible, or make the same request on the client's behalf
to either the next inbound proxy server or directly to the origin
server indicated by the request-target. Requirements on such
"forwarding" of messages are defined in Section 5.5 of [Semantics].

An example absolute-form of request-line would be:

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

To allow for transition to the absolute-form for all requests in some
future version of HTTP, a server MUST accept the absolute-form in
requests, even though HTTP/1.1 clients will only send them in
requests to proxies.

3.2.3. authority-form

The authority-form of request-target is only used for CONNECT
requests (Section 7.3.6 of [Semantics]).

```
   authority-form = authority
```

When making a CONNECT request to establish a tunnel through one or
more proxies, a client MUST send only the target URI's authority
component (excluding any userinfo and its "@" delimiter) as the
request-target. For example,

```
   CONNECT www.example.com:80 HTTP/1.1
```

3.2.4. asterisk-form

The asterisk-form of request-target is only used for a server-wide
OPTIONS request (Section 7.3.7 of [Semantics]).

```
   asterisk-form  = "*"
```

When a client wishes to request OPTIONS for the server as a whole, as
opposed to a specific named resource of that server, the client MUST
send only "*" (%x2A) as the request-target. For example,

    OPTIONS * HTTP/1.1

If a proxy receives an OPTIONS request with an absolute-form of
request-target in which the URI has an empty path and no query
component, then the last proxy on the request chain MUST send a
request-target of "*" when it forwards the request to the indicated
origin server.

For example, the request

    OPTIONS http://www.example.org:8001 HTTP/1.1

would be forwarded by the final proxy as

    OPTIONS * HTTP/1.1
    Host: www.example.org:8001

after connecting to port 8001 of host "www.example.org".

3.3. Effective Request URI

Since the request-target often contains only part of the user agent's
target URI, a server reconstructs the intended target as an effective
request URI to properly service the request (Section 5.3 of
[Semantics]).

If the request-target is in absolute-form, the effective request URI
is the same as the request-target. Otherwise, the effective request
URI is constructed as follows:

   If the server's configuration (or outbound gateway) provides a
   fixed URI scheme, that scheme is used for the effective request
   URI. Otherwise, if the request is received over a TLS-secured TCP
   connection, the effective request URI's scheme is "https"; if not,
   the scheme is "http".

   If the server's configuration (or outbound gateway) provides a
   fixed URI authority component, that authority is used for the
   effective request URI. If not, then if the request-target is in
   authority-form, the effective request URI's authority component is
   the same as the request-target. If not, then if a Host header
   field is supplied with a non-empty field-value, the authority
   component is the same as the Host field-value. Otherwise, the
   authority component is assigned the default name configured for

      the server and, if the connection's incoming TCP port number
      differs from the default port for the effective request URI's
      scheme, then a colon (":") and the incoming port number (in
      decimal form) are appended to the authority component.

      If the request-target is in authority-form or asterisk-form, the
      effective request URI's combined path and query component is
      empty. Otherwise, the combined path and query component is the
      same as the request-target.

      The components of the effective request URI, once determined as
      above, can be combined into absolute-URI form by concatenating the
      scheme, "://", authority, and combined path and query component.

   Example 1: the following message received over an insecure TCP
   connection

     GET /pub/WWW/TheProject.html HTTP/1.1
     Host: www.example.org:8080

   has an effective request URI of

     http://www.example.org:8080/pub/WWW/TheProject.html

   Example 2: the following message received over a TLS-secured TCP
   connection

     OPTIONS * HTTP/1.1
     Host: www.example.org

   has an effective request URI of

     https://www.example.org

   Recipients of an HTTP/1.0 request that lacks a Host header field
   might need to use heuristics (e.g., examination of the URI path for
   something unique to a particular host) in order to guess the
   effective request URI's authority component.

4. Status Line

   The first line of a response message is the status-line, consisting
   of the protocol version, a space (SP), the status code, another
   space, and ending with an OPTIONAL textual phrase describing the
   status code.

     status-line = HTTP-version SP status-code SP [reason-phrase]

   Although the status-line grammar rule requires that each of the
   component elements be separated by a single SP octet, recipients MAY
   instead parse on whitespace-delimited word boundaries and, aside from
   the line terminator, treat any form of whitespace as the SP separator
   while ignoring preceding or trailing whitespace; such whitespace
   includes one or more of the following octets: SP, HTAB, VT (%x0B), FF
   (%x0C), or bare CR. However, lenient parsing can result in response
   splitting security vulnerabilities if there are multiple recipients
   of the message and each has its own unique interpretation of
   robustness (see Section 11.1).

   The status-code element is a 3-digit integer code describing the
   result of the server's attempt to understand and satisfy the client's
   corresponding request. The rest of the response message is to be
   interpreted in light of the semantics defined for that status code.
   See Section 9 of [Semantics] for information about the semantics of
   status codes, including the classes of status code (indicated by the
   first digit), the status codes defined by this specification,
   considerations for the definition of new status codes, and the IANA
   registry.

     status-code   = 3DIGIT

   The reason-phrase element exists for the sole purpose of providing a
   textual description associated with the numeric status code, mostly
   out of deference to earlier Internet application protocols that were
   more frequently used with interactive text clients.

     reason-phrase = 1*(HTAB / SP / VCHAR / obs-text)

   A client SHOULD ignore the reason-phrase content because it is not a
   reliable channel for information (it might be translated for a given
   locale, overwritten by intermediaries, or discarded when the message
   is forwarded via other versions of HTTP). A server MUST send the
   space that separates status-code from the reason-phrase even when the
   reason-phrase is absent (i.e., the status-line would end with the
   three octets SP CR LF).

5. Header Field Syntax

   Each header field consists of a case-insensitive field name followed
   by a colon (":"), optional leading whitespace, the field value, and
   optional trailing whitespace.

     header-field   = field-name ":" OWS field-value OWS

   Most HTTP field names and the rules for parsing within field values
   are defined in Section 4 of [Semantics]. This section covers the

generic syntax for header field inclusion within, and extraction
from, HTTP/1.1 messages. In addition, the following header fields
are defined by this document because they are specific to HTTP/1.1
message processing:

```
+-------------------+----------+---------------+
| Header Field Name | Status   | Reference     |
+-------------------+----------+---------------+
| Connection        | standard | Section 9.1   |
| MIME-Version      | standard | Appendix B.1  |
| TE                | standard | Section 7.4   |
| Transfer-Encoding | standard | Section 6.1   |
| Upgrade           | standard | Section 9.9   |
+-------------------+----------+---------------+
```

                          Table 1

Furthermore, the field name "Close" is reserved, since using that
name as an HTTP header field might conflict with the "close"
connection option of the Connection header field (Section 9.1).

```
+-------------------+----------+----------+------------+
| Header Field Name | Protocol | Status   | Reference  |
+-------------------+----------+----------+------------+
| Close             | http     | reserved | Section 5  |
+-------------------+----------+----------+------------+
```

5.1. Header Field Parsing

Messages are parsed using a generic algorithm, independent of the
individual header field names. The contents within a given field
value are not parsed until a later stage of message interpretation
(usually after the message's entire header section has been
processed).

No whitespace is allowed between the header field-name and colon. In
the past, differences in the handling of such whitespace have led to
security vulnerabilities in request routing and response handling. A
server MUST reject any received request message that contains
whitespace between a header field-name and colon with a response
status code of 400 (Bad Request). A proxy MUST remove any such
whitespace from a response message before forwarding the message
downstream.

A field value might be preceded and/or followed by optional
whitespace (OWS); a single SP preceding the field-value is preferred
for consistent readability by humans. The field value does not
include any leading or trailing whitespace: OWS occurring before the

first non-whitespace octet of the field value or after the last non-whitespace octet of the field value ought to be excluded by parsers when extracting the field value from a header field.

5.2. Obsolete Line Folding

Historically, HTTP header field values could be extended over multiple lines by preceding each extra line with at least one space or horizontal tab (obs-fold). This specification deprecates such line folding except within the message/http media type (Section 10.1).

```
  obs-fold     = OWS CRLF RWS
               ; obsolete line folding
```

A sender MUST NOT generate a message that includes line folding (i.e., that has any field-value that contains a match to the obs-fold rule) unless the message is intended for packaging within the message/http media type.

A server that receives an obs-fold in a request message that is not within a message/http container MUST either reject the message by sending a 400 (Bad Request), preferably with a representation explaining that obsolete line folding is unacceptable, or replace each received obs-fold with one or more SP octets prior to interpreting the field value or forwarding the message downstream.

A proxy or gateway that receives an obs-fold in a response message that is not within a message/http container MUST either discard the message and replace it with a 502 (Bad Gateway) response, preferably with a representation explaining that unacceptable line folding was received, or replace each received obs-fold with one or more SP octets prior to interpreting the field value or forwarding the message downstream.

A user agent that receives an obs-fold in a response message that is not within a message/http container MUST replace each received obs-fold with one or more SP octets prior to interpreting the field value.

6. Message Body

The message body (if any) of an HTTP message is used to carry the payload body (Section 6.3.3 of [Semantics]) of that request or response. The message body is identical to the payload body unless a transfer coding has been applied, as described in Section 6.1.

```
  message-body = *OCTET
```

The rules for determining when a message body is present in an
HTTP/1.1 message differ for requests and responses.

The presence of a message body in a request is signaled by a Content-
Length or Transfer-Encoding header field. Request message framing is
independent of method semantics, even if the method does not define
any use for a message body.

The presence of a message body in a response depends on both the
request method to which it is responding and the response status code
(Section 4), and corresponds to when a payload body is allowed; see
Section 6.3.3 of [Semantics].

6.1. Transfer-Encoding

The Transfer-Encoding header field lists the transfer coding names
corresponding to the sequence of transfer codings that have been (or
will be) applied to the payload body in order to form the message
body. Transfer codings are defined in Section 7.

     Transfer-Encoding = 1#transfer-coding

Transfer-Encoding is analogous to the Content-Transfer-Encoding field
of MIME, which was designed to enable safe transport of binary data
over a 7-bit transport service ([RFC2045], Section 6). However, safe
transport has a different focus for an 8bit-clean transfer protocol.
In HTTP's case, Transfer-Encoding is primarily intended to accurately
delimit a dynamically generated payload and to distinguish payload
encodings that are only applied for transport efficiency or security
from those that are characteristics of the selected resource.

A recipient MUST be able to parse the chunked transfer coding
(Section 7.1) because it plays a crucial role in framing messages
when the payload body size is not known in advance. A sender MUST
NOT apply chunked more than once to a message body (i.e., chunking an
already chunked message is not allowed). If any transfer coding
other than chunked is applied to a request payload body, the sender
MUST apply chunked as the final transfer coding to ensure that the
message is properly framed. If any transfer coding other than
chunked is applied to a response payload body, the sender MUST either
apply chunked as the final transfer coding or terminate the message
by closing the connection.

For example,

     Transfer-Encoding: gzip, chunked

indicates that the payload body has been compressed using the gzip
coding and then chunked using the chunked coding while forming the
message body.

Unlike Content-Encoding (Section 6.1.2 of [Semantics]), Transfer-
Encoding is a property of the message, not of the representation, and
any recipient along the request/response chain MAY decode the
received transfer coding(s) or apply additional transfer coding(s) to
the message body, assuming that corresponding changes are made to the
Transfer-Encoding field-value. Additional information about the
encoding parameters can be provided by other header fields not
defined by this specification.

Transfer-Encoding MAY be sent in a response to a HEAD request or in a
304 (Not Modified) response (Section 9.4.5 of [Semantics]) to a GET
request, neither of which includes a message body, to indicate that
the origin server would have applied a transfer coding to the message
body if the request had been an unconditional GET. This indication
is not required, however, because any recipient on the response chain
(including the origin server) can remove transfer codings when they
are not needed.

A server MUST NOT send a Transfer-Encoding header field in any
response with a status code of 1xx (Informational) or 204 (No
Content). A server MUST NOT send a Transfer-Encoding header field in
any 2xx (Successful) response to a CONNECT request (Section 7.3.6 of
[Semantics]).

Transfer-Encoding was added in HTTP/1.1. It is generally assumed
that implementations advertising only HTTP/1.0 support will not
understand how to process a transfer-encoded payload. A client MUST
NOT send a request containing Transfer-Encoding unless it knows the
server will handle HTTP/1.1 (or later) requests; such knowledge might
be in the form of specific user configuration or by remembering the
version of a prior received response. A server MUST NOT send a
response containing Transfer-Encoding unless the corresponding
request indicates HTTP/1.1 (or later).

A server that receives a request message with a transfer coding it
does not understand SHOULD respond with 501 (Not Implemented).

6.2. Content-Length

When a message does not have a Transfer-Encoding header field, a
Content-Length header field can provide the anticipated size, as a
decimal number of octets, for a potential payload body. For messages
that do include a payload body, the Content-Length field-value
provides the framing information necessary for determining where the

body (and message) ends. For messages that do not include a payload
body, the Content-Length indicates the size of the selected
representation (Section 6.2.4 of [Semantics]).

   Note: HTTP's use of Content-Length for message framing differs
   significantly from the same field's use in MIME, where it is an
   optional field used only within the "message/external-body" media-
   type.

6.3. Message Body Length

The length of a message body is determined by one of the following
(in order of precedence):

1. Any response to a HEAD request and any response with a 1xx
    (Informational), 204 (No Content), or 304 (Not Modified) status
    code is always terminated by the first empty line after the
    header fields, regardless of the header fields present in the
    message, and thus cannot contain a message body.

2. Any 2xx (Successful) response to a CONNECT request implies that
    the connection will become a tunnel immediately after the empty
    line that concludes the header fields. A client MUST ignore any
    Content-Length or Transfer-Encoding header fields received in
    such a message.

3. If a Transfer-Encoding header field is present and the chunked
    transfer coding (Section 7.1) is the final encoding, the message
    body length is determined by reading and decoding the chunked
    data until the transfer coding indicates the data is complete.

    If a Transfer-Encoding header field is present in a response and
    the chunked transfer coding is not the final encoding, the
    message body length is determined by reading the connection until
    it is closed by the server. If a Transfer-Encoding header field
    is present in a request and the chunked transfer coding is not
    the final encoding, the message body length cannot be determined
    reliably; the server MUST respond with the 400 (Bad Request)
    status code and then close the connection.

    If a message is received with both a Transfer-Encoding and a
    Content-Length header field, the Transfer-Encoding overrides the
    Content-Length. Such a message might indicate an attempt to
    perform request smuggling (Section 11.2) or response splitting
    (Section 11.1) and ought to be handled as an error. A sender
    MUST remove the received Content-Length field prior to forwarding
    such a message downstream.

4. If a message is received without Transfer-Encoding and with
    either multiple Content-Length header fields having differing
    field-values or a single Content-Length header field having an
    invalid value, then the message framing is invalid and the
    recipient MUST treat it as an unrecoverable error. If this is a
    request message, the server MUST respond with a 400 (Bad Request)
    status code and then close the connection. If this is a response
    message received by a proxy, the proxy MUST close the connection
    to the server, discard the received response, and send a 502 (Bad
    Gateway) response to the client. If this is a response message
    received by a user agent, the user agent MUST close the
    connection to the server and discard the received response.

5. If a valid Content-Length header field is present without
    Transfer-Encoding, its decimal value defines the expected message
    body length in octets. If the sender closes the connection or
    the recipient times out before the indicated number of octets are
    received, the recipient MUST consider the message to be
    incomplete and close the connection.

6. If this is a request message and none of the above are true, then
    the message body length is zero (no message body is present).

7. Otherwise, this is a response message without a declared message
    body length, so the message body length is determined by the
    number of octets received prior to the server closing the
    connection.

Since there is no way to distinguish a successfully completed, close-
delimited message from a partially received message interrupted by
network failure, a server SHOULD generate encoding or length-
delimited messages whenever possible. The close-delimiting feature
exists primarily for backwards compatibility with HTTP/1.0.

A server MAY reject a request that contains a message body but not a
Content-Length by responding with 411 (Length Required).

Unless a transfer coding other than chunked has been applied, a
client that sends a request containing a message body SHOULD use a
valid Content-Length header field if the message body length is known
in advance, rather than the chunked transfer coding, since some
existing services respond to chunked with a 411 (Length Required)
status code even though they understand the chunked transfer coding.
This is typically because such services are implemented via a gateway
that requires a content-length in advance of being called and the
server is unable or unwilling to buffer the entire request before
processing.

A user agent that sends a request containing a message body MUST send
a valid Content-Length header field if it does not know the server
will handle HTTP/1.1 (or later) requests; such knowledge can be in
the form of specific user configuration or by remembering the version
of a prior received response.

If the final response to the last request on a connection has been
completely received and there remains additional data to read, a user
agent MAY discard the remaining data or attempt to determine if that
data belongs as part of the prior response body, which might be the
case if the prior message's Content-Length value is incorrect. A
client MUST NOT process, cache, or forward such extra data as a
separate response, since such behavior would be vulnerable to cache
poisoning.

7. Transfer Codings

Transfer coding names are used to indicate an encoding transformation
that has been, can be, or might need to be applied to a payload body
in order to ensure "safe transport" through the network. This
differs from a content coding in that the transfer coding is a
property of the message rather than a property of the representation
that is being transferred.

```
transfer-coding = token *( OWS ";" OWS transfer-parameter )
```

Parameters are in the form of a name=value pair.

```
transfer-parameter = token BWS "=" BWS ( token / quoted-string )
```

All transfer-coding names are case-insensitive and ought to be
registered within the HTTP Transfer Coding registry, as defined in
Section 7.3. They are used in the TE (Section 7.4) and Transfer-
Encoding (Section 6.1) header fields.

```
+------------+-------------------------------------------+-----------+
| Name       | Description                               | Reference |
+------------+-------------------------------------------+-----------+
| chunked    | Transfer in a series of chunks            | Section 7 |
|            |                                           | .1        |
| compress   | UNIX "compress" data format [Welch]       | Section 7 |
|            |                                           | .2        |
| deflate    | "deflate" compressed data ([RFC1951])     | Section 7 |
|            | inside the "zlib" data format             | .2        |
|            | ([RFC1950])                               |           |
| gzip       | GZIP file format [RFC1952]                | Section 7 |
|            |                                           | .2        |
| trailers   | (reserved)                                | Section 7 |
| x-compress | Deprecated (alias for compress)           | Section 7 |
|            |                                           | .2        |
| x-gzip     | Deprecated (alias for gzip)               | Section 7 |
|            |                                           | .2        |
+------------+-------------------------------------------+-----------+
```

Table 2

   Note: the coding name "trailers" is reserved because its use would
   conflict with the keyword "trailers" in the TE header field
   (Section 7.4).

7.1. Chunked Transfer Coding

   The chunked transfer coding wraps the payload body in order to
   transfer it as a series of chunks, each with its own size indicator,
   followed by an OPTIONAL trailer section containing trailer fields.
   Chunked enables content streams of unknown size to be transferred as
   a sequence of length-delimited buffers, which enables the sender to
   retain connection persistence and the recipient to know when it has
   received the entire message.

```
   chunked-body   = *chunk
                    last-chunk
                    trailer-section
                    CRLF

   chunk          = chunk-size [ chunk-ext ] CRLF
                    chunk-data CRLF
   chunk-size     = 1*HEXDIG
   last-chunk     = 1*("0") [ chunk-ext ] CRLF

   chunk-data     = 1*OCTET ; a sequence of chunk-size octets
```

   The chunk-size field is a string of hex digits indicating the size of
   the chunk-data in octets. The chunked transfer coding is complete
   when a chunk with a chunk-size of zero is received, possibly followed
   by a trailer section, and finally terminated by an empty line.

   A recipient MUST be able to parse and decode the chunked transfer
   coding.

   The chunked encoding does not define any parameters. Their presence
   SHOULD be treated as an error.

7.1.1. Chunk Extensions

   The chunked encoding allows each chunk to include zero or more chunk
   extensions, immediately following the chunk-size, for the sake of
   supplying per-chunk metadata (such as a signature or hash), mid-
   message control information, or randomization of message body size.

```
     chunk-ext      = *( BWS ";" BWS chunk-ext-name
                         [ BWS "=" BWS chunk-ext-val ] )

     chunk-ext-name = token
     chunk-ext-val  = token / quoted-string
```

   The chunked encoding is specific to each connection and is likely to
   be removed or recoded by each recipient (including intermediaries)
   before any higher-level application would have a chance to inspect
   the extensions. Hence, use of chunk extensions is generally limited
   to specialized HTTP services such as "long polling" (where client and
   server can have shared expectations regarding the use of chunk
   extensions) or for padding within an end-to-end secured connection.

   A recipient MUST ignore unrecognized chunk extensions. A server
   ought to limit the total length of chunk extensions received in a
   request to an amount reasonable for the services provided, in the
   same way that it applies length limitations and timeouts for other
   parts of a message, and generate an appropriate 4xx (Client Error)
   response if that amount is exceeded.

7.1.2. Chunked Trailer Section

   A trailer section allows the sender to include additional fields at
   the end of a chunked message in order to supply metadata that might
   be dynamically generated while the message body is sent, such as a
   message integrity check, digital signature, or post-processing
   status. The proper use and limitations of trailer fields are defined
   in Section 4.3 of [Semantics].

```
   trailer-section   = *( header-field CRLF )
```

   A recipient that decodes and removes the chunked encoding from a
   message (e.g., for storage or forwarding to a non-HTTP/1.1 peer) MUST
   discard any received trailer fields, store/forward them separately
   from the header fields, or selectively merge into the header section
   only those trailer fields corresponding to header field definitions
   that are understood by the recipient to explicitly permit and define
   how their corresponding trailer field value can be safely merged.

7.1.3. Decoding Chunked

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

```
   length := 0
   read chunk-size, chunk-ext (if any), and CRLF
   while (chunk-size > 0) {
      read chunk-data and CRLF
      append chunk-data to decoded-body
      length := length + chunk-size
      read chunk-size, chunk-ext (if any), and CRLF
   }
   read trailer field
   while (trailer field is not empty) {
      if (trailer fields are stored/forwarded separately) {
          append trailer field to existing trailer fields
      }
      else if (trailer field is understood and defined as mergeable) {
          merge trailer field with existing header fields
      }
      else {
          discard trailer field
      }
      read trailer field
   }
   Content-Length := length
   Remove "chunked" from Transfer-Encoding
   Remove Trailer from existing header fields
```

7.2. Transfer Codings for Compression

   The following transfer coding names for compression are defined by
   the same algorithm as their corresponding content coding:

   compress (and x-compress)
      See Section 6.1.2.1 of [Semantics].

   deflate
      See Section 6.1.2.2 of [Semantics].

   gzip (and x-gzip)
      See Section 6.1.2.3 of [Semantics].

   The compression codings do not define any parameters. Their presence
   SHOULD be treated as an error.

7.3. Transfer Coding Registry

   The "HTTP Transfer Coding Registry" defines the namespace for
   transfer coding names. It is maintained at
   <https://www.iana.org/assignments/http-parameters>.

   Registrations MUST include the following fields:

   o Name

   o Description

   o Pointer to specification text

   Names of transfer codings MUST NOT overlap with names of content
   codings (Section 6.1.2 of [Semantics]) unless the encoding
   transformation is identical, as is the case for the compression
   codings defined in Section 7.2.

   The TE header field (Section 7.4) uses a pseudo parameter named "q"
   as rank value when multiple transfer codings are acceptable. Future
   registrations of transfer codings SHOULD NOT define parameters called
   "q" (case-insensitively) in order to avoid ambiguities.

   Values to be added to this namespace require IETF Review (see
   Section 4.8 of [RFC8126]), and MUST conform to the purpose of
   transfer coding defined in this specification.

   Use of program names for the identification of encoding formats is
   not desirable and is discouraged for future encodings.

7.4. TE

   The "TE" header field in a request indicates what transfer codings,
   besides chunked, the client is willing to accept in response, and
   whether or not the client is willing to accept trailer fields in a
   chunked transfer coding.

The TE field-value consists of a comma-separated list of transfer
coding names, each allowing for optional parameters (as described in
Section 7), and/or the keyword "trailers". A client MUST NOT send
the chunked transfer coding name in TE; chunked is always acceptable
for HTTP/1.1 recipients.

```
TE        = #t-codings
t-codings = "trailers" / ( transfer-coding [ t-ranking ] )
t-ranking = OWS ";" OWS "q=" rank
rank      = ( "0" [ "." 0*3DIGIT ] )
          / ( "1" [ "." 0*3("0") ] )
```

Three examples of TE use are below.

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

The presence of the keyword "trailers" indicates that the client is
willing to accept trailer fields in a chunked transfer coding, as
defined in Section 7.1.2, on behalf of itself and any downstream
clients. For requests from an intermediary, this implies that
either: (a) all downstream clients are willing to accept trailer
fields in the forwarded response; or, (b) the intermediary will
attempt to buffer the response on behalf of downstream recipients.
Note that HTTP/1.1 does not define any means to limit the size of a
chunked response such that an intermediary can be assured of
buffering the entire response.

When multiple transfer codings are acceptable, the client MAY rank
the codings by preference using a case-insensitive "q" parameter
(similar to the qvalues used in content negotiation fields,
Section 8.4.1 of [Semantics]). The rank value is a real number in
the range 0 through 1, where 0.001 is the least preferred and 1 is
the most preferred; a value of 0 means "not acceptable".

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

Since the TE header field only applies to the immediate connection, a
sender of TE MUST also send a "TE" connection option within the
Connection header field (Section 9.1) in order to prevent the TE
field from being forwarded by intermediaries that do not support its
semantics.

8. **Handling Incomplete Messages**

   A server that receives an incomplete request message, usually due to
   a canceled request or a triggered timeout exception, MAY send an
   error response prior to closing the connection.

   A client that receives an incomplete response message, which can
   occur when a connection is closed prematurely or when decoding a
   supposedly chunked transfer coding fails, MUST record the message as
   incomplete. Cache requirements for incomplete responses are defined
   in Section 3 of [Caching].

   If a response terminates in the middle of the header section (before
   the empty line is received) and the status code might rely on header
   fields to convey the full meaning of the response, then the client
   cannot assume that meaning has been conveyed; the client might need
   to repeat the request in order to determine what action to take next.

   A message body that uses the chunked transfer coding is incomplete if
   the zero-sized chunk that terminates the encoding has not been
   received. A message that uses a valid Content-Length is incomplete
   if the size of the message body received (in octets) is less than the
   value given by Content-Length. A response that has neither chunked
   transfer coding nor Content-Length is terminated by closure of the
   connection and, thus, is considered complete regardless of the number
   of message body octets received, provided that the header section was
   received intact.

9. **Connection Management**

   HTTP messaging is independent of the underlying transport- or
   session-layer connection protocol(s). HTTP only presumes a reliable
   transport with in-order delivery of requests and the corresponding
   in-order delivery of responses. The mapping of HTTP request and
   response structures onto the data units of an underlying transport
   protocol is outside the scope of this specification.

   As described in Section 5.2 of [Semantics], the specific connection
   protocols to be used for an HTTP interaction are determined by client
   configuration and the target URI. For example, the "http" URI scheme
   (Section 2.5.1 of [Semantics]) indicates a default connection of TCP
   over IP, with a default TCP port of 80, but the client might be
   configured to use a proxy via some other connection, port, or
   protocol.

   HTTP implementations are expected to engage in connection management,
   which includes maintaining the state of current connections,
   establishing a new connection or reusing an existing connection,

   processing messages received on a connection, detecting connection
   failures, and closing each connection. Most clients maintain
   multiple connections in parallel, including more than one connection
   per server endpoint. Most servers are designed to maintain thousands
   of concurrent connections, while controlling request queues to enable
   fair use and detect denial-of-service attacks.

9.1. Connection

   The "Connection" header field allows the sender to indicate desired
   control options for the current connection. In order to avoid
   confusing downstream recipients, a proxy or gateway MUST remove or
   replace any received connection options before forwarding the
   message.

   When a header field aside from Connection is used to supply control
   information for or about the current connection, the sender MUST list
   the corresponding field-name within the Connection header field. A
   proxy or gateway MUST parse a received Connection header field before
   a message is forwarded and, for each connection-option in this field,
   remove any header field(s) from the message with the same name as the
   connection-option, and then remove the Connection header field itself
   (or replace it with the intermediary's own connection options for the
   forwarded message).

   Hence, the Connection header field provides a declarative way of
   distinguishing header fields that are only intended for the immediate
   recipient ("hop-by-hop") from those fields that are intended for all
   recipients on the chain ("end-to-end"), enabling the message to be
   self-descriptive and allowing future connection-specific extensions
   to be deployed without fear that they will be blindly forwarded by
   older intermediaries.

   The Connection header field's value has the following grammar:

     Connection   = 1#connection-option
     connection-option = token

   Connection options are case-insensitive.

   A sender MUST NOT send a connection option corresponding to a header
   field that is intended for all recipients of the payload. For
   example, Cache-Control is never appropriate as a connection option
   (Section 5.2 of [Caching]).

   The connection options do not always correspond to a header field
   present in the message, since a connection-specific header field
   might not be needed if there are no parameters associated with a

connection option. In contrast, a connection-specific header field
that is received without a corresponding connection option usually
indicates that the field has been improperly forwarded by an
intermediary and ought to be ignored by the recipient.

When defining new connection options, specification authors ought to
survey existing header field names and ensure that the new connection
option does not share the same name as an already deployed header
field. Defining a new connection option essentially reserves that
potential field-name for carrying additional information related to
the connection option, since it would be unwise for senders to use
that field-name for anything else.

The "close" connection option is defined for a sender to signal that
this connection will be closed after completion of the response. For
example,

   Connection: close

in either the request or the response header fields indicates that
the sender is going to close the connection after the current
request/response is complete (Section 9.7).

A client that does not support persistent connections MUST send the
"close" connection option in every request message.

A server that does not support persistent connections MUST send the
"close" connection option in every response message that does not
have a 1xx (Informational) status code.

9.2. Establishment

It is beyond the scope of this specification to describe how
connections are established via various transport- or session-layer
protocols. Each connection applies to only one transport link.

9.3. Associating a Response to a Request

HTTP/1.1 does not include a request identifier for associating a
given request message with its corresponding one or more response
messages. Hence, it relies on the order of response arrival to
correspond exactly to the order in which requests are made on the
same connection. More than one response message per request only
occurs when one or more informational responses (1xx, see Section 9.2
of [Semantics]) precede a final response to the same request.

A client that has more than one outstanding request on a connection
MUST maintain a list of outstanding requests in the order sent and

   MUST associate each received response message on that connection to
   the highest ordered request that has not yet received a final (non-
   1xx) response.

   If an HTTP/1.1 client receives data on a connection that doesn't have
   any outstanding requests, it MUST NOT consider them to be a response
   to a not-yet-issued request; it SHOULD close the connection, since
   message delimitation is now ambiguous, unless the data consists only
   of one or more CRLF (which can be discarded, as per Section 2.2).

9.4. Persistence

   HTTP/1.1 defaults to the use of "persistent connections", allowing
   multiple requests and responses to be carried over a single
   connection. The "close" connection option is used to signal that a
   connection will not persist after the current request/response. HTTP
   implementations SHOULD support persistent connections.

   A recipient determines whether a connection is persistent or not
   based on the most recently received message's protocol version and
   Connection header field (if any):

   o If the "close" connection option is present, the connection will
      not persist after the current response; else,

   o If the received protocol is HTTP/1.1 (or later), the connection
      will persist after the current response; else,

   o If the received protocol is HTTP/1.0, the "keep-alive" connection
      option is present, either the recipient is not a proxy or the
      message is a response, and the recipient wishes to honor the
      HTTP/1.0 "keep-alive" mechanism, the connection will persist after
      the current response; otherwise,

   o The connection will close after the current response.

   A client MAY send additional requests on a persistent connection
   until it sends or receives a "close" connection option or receives an
   HTTP/1.0 response without a "keep-alive" connection option.

   In order to remain persistent, all messages on a connection need to
   have a self-defined message length (i.e., one not defined by closure
   of the connection), as described in Section 6. A server MUST read
   the entire request message body or close the connection after sending
   its response, since otherwise the remaining data on a persistent
   connection would be misinterpreted as the next request. Likewise, a
   client MUST read the entire response message body if it intends to
   reuse the same connection for a subsequent request.

A proxy server MUST NOT maintain a persistent connection with an
HTTP/1.0 client (see [Section 19.7.1 of [RFC2068]](#) for information and
discussion of the problems with the Keep-Alive header field
implemented by many HTTP/1.0 clients).

See [Appendix C.1.2](#) for more information on backwards compatibility
with HTTP/1.0 clients.

9.4.1. Retrying Requests

Connections can be closed at any time, with or without intention.
Implementations ought to anticipate the need to recover from
asynchronous close events. The conditions under which a client can
automatically retry a sequence of outstanding requests are defined in
Section 7.2.2 of [[Semantics](#)].

9.4.2. Pipelining

A client that supports persistent connections MAY "pipeline" its
requests (i.e., send multiple requests without waiting for each
response). A server MAY process a sequence of pipelined requests in
parallel if they all have safe methods (Section 7.2.1 of
[[Semantics](#)]), but it MUST send the corresponding responses in the
same order that the requests were received.

A client that pipelines requests SHOULD retry unanswered requests if
the connection closes before it receives all of the corresponding
responses. When retrying pipelined requests after a failed
connection (a connection not explicitly closed by the server in its
last complete response), a client MUST NOT pipeline immediately after
connection establishment, since the first remaining request in the
prior pipeline might have caused an error response that can be lost
again if multiple requests are sent on a prematurely closed
connection (see the TCP reset problem described in [Section 9.7](#)).

Idempotent methods (Section 7.2.2 of [[Semantics](#)]) are significant to
pipelining because they can be automatically retried after a
connection failure. A user agent SHOULD NOT pipeline requests after
a non-idempotent method, until the final response status code for
that method has been received, unless the user agent has a means to
detect and recover from partial failure conditions involving the
pipelined sequence.

An intermediary that receives pipelined requests MAY pipeline those
requests when forwarding them inbound, since it can rely on the
outbound user agent(s) to determine what requests can be safely
pipelined. If the inbound connection fails before receiving a
response, the pipelining intermediary MAY attempt to retry a sequence

of requests that have yet to receive a response if the requests all
have idempotent methods; otherwise, the pipelining intermediary
SHOULD forward any received responses and then close the
corresponding outbound connection(s) so that the outbound user
agent(s) can recover accordingly.

9.5. Concurrency

A client ought to limit the number of simultaneous open connections
that it maintains to a given server.

Previous revisions of HTTP gave a specific number of connections as a
ceiling, but this was found to be impractical for many applications.
As a result, this specification does not mandate a particular maximum
number of connections but, instead, encourages clients to be
conservative when opening multiple connections.

Multiple connections are typically used to avoid the "head-of-line
blocking" problem, wherein a request that takes significant server-
side processing and/or has a large payload blocks subsequent requests
on the same connection. However, each connection consumes server
resources. Furthermore, using multiple connections can cause
undesirable side effects in congested networks.

Note that a server might reject traffic that it deems abusive or
characteristic of a denial-of-service attack, such as an excessive
number of open connections from a single client.

9.6. Failures and Timeouts

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

A client or server that wishes to time out SHOULD issue a graceful
close on the connection. Implementations SHOULD constantly monitor
open connections for a received closure signal and respond to it as
appropriate, since prompt closure of both sides of a connection
enables allocated system resources to be reclaimed.

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

closed while it was idle, but from the client's point of view, a
request is in progress.

A server SHOULD sustain persistent connections, when possible, and
allow the underlying transport's flow-control mechanisms to resolve
temporary overloads, rather than terminate connections with the
expectation that clients will retry. The latter technique can
exacerbate network congestion.

A client sending a message body SHOULD monitor the network connection
for an error response while it is transmitting the request. If the
client sees a response that indicates the server does not wish to
receive the message body and is closing the connection, the client
SHOULD immediately cease transmitting the body and close its side of
the connection.

9.7. Tear-down

The Connection header field (Section 9.1) provides a "close"
connection option that a sender SHOULD send when it wishes to close
the connection after the current request/response pair.

A client that sends a "close" connection option MUST NOT send further
requests on that connection (after the one containing "close") and
MUST close the connection after reading the final response message
corresponding to this request.

A server that receives a "close" connection option MUST initiate a
close of the connection (see below) after it sends the final response
to the request that contained "close". The server SHOULD send a
"close" connection option in its final response on that connection.
The server MUST NOT process any further requests received on that
connection.

A server that sends a "close" connection option MUST initiate a close
of the connection (see below) after it sends the response containing
"close". The server MUST NOT process any further requests received
on that connection.

A client that receives a "close" connection option MUST cease sending
requests on that connection and close the connection after reading
the response message containing the "close"; if additional pipelined
requests had been sent on the connection, the client SHOULD NOT
assume that they will be processed by the server.

If a server performs an immediate close of a TCP connection, there is
a significant risk that the client will not be able to read the last
HTTP response. If the server receives additional data from the

   client on a fully closed connection, such as another request that was
   sent by the client before receiving the server's response, the
   server's TCP stack will send a reset packet to the client;
   unfortunately, the reset packet might erase the client's
   unacknowledged input buffers before they can be read and interpreted
   by the client's HTTP parser.

   To avoid the TCP reset problem, servers typically close a connection
   in stages. First, the server performs a half-close by closing only
   the write side of the read/write connection. The server then
   continues to read from the connection until it receives a
   corresponding close by the client, or until the server is reasonably
   certain that its own TCP stack has received the client's
   acknowledgement of the packet(s) containing the server's last
   response. Finally, the server fully closes the connection.

   It is unknown whether the reset problem is exclusive to TCP or might
   also be found in other transport connection protocols.

9.8. **TLS Connection Closure**

   TLS provides a facility for secure connection closure. When a valid
   closure alert is received, an implementation can be assured that no
   further data will be received on that connection. TLS
   implementations MUST initiate an exchange of closure alerts before
   closing a connection. A TLS implementation MAY, after sending a
   closure alert, close the connection without waiting for the peer to
   send its closure alert, generating an "incomplete close". Note that
   an implementation which does this MAY choose to reuse the session.
   This SHOULD only be done when the application knows (typically
   through detecting HTTP message boundaries) that it has received all
   the message data that it cares about.

   As specified in [RFC8446], any implementation which receives a
   connection close without first receiving a valid closure alert (a
   "premature close") MUST NOT reuse that session. Note that a
   premature close does not call into question the security of the data
   already received, but simply indicates that subsequent data might
   have been truncated. Because TLS is oblivious to HTTP request/
   response boundaries, it is necessary to examine the HTTP data itself
   (specifically the Content-Length header) to determine whether the
   truncation occurred inside a message or between messages.

   When encountering a premature close, a client SHOULD treat as
   completed all requests for which it has received as much data as
   specified in the Content-Length header.

A client detecting an incomplete close SHOULD recover gracefully. It
MAY resume a TLS session closed in this fashion.

Clients MUST send a closure alert before closing the connection.
Clients which are unprepared to receive any more data MAY choose not
to wait for the server's closure alert and simply close the
connection, thus generating an incomplete close on the server side.

Servers SHOULD be prepared to receive an incomplete close from the
client, since the client can often determine when the end of server
data is. Servers SHOULD be willing to resume TLS sessions closed in
this fashion.

Servers MUST attempt to initiate an exchange of closure alerts with
the client before closing the connection. Servers MAY close the
connection after sending the closure alert, thus generating an
incomplete close on the client side.

9.9. Upgrade

The "Upgrade" header field is intended to provide a simple mechanism
for transitioning from HTTP/1.1 to some other protocol on the same
connection.

A client MAY send a list of protocol names in the Upgrade header
field of a request to invite the server to switch to one or more of
the named protocols, in order of descending preference, before
sending the final response. A server MAY ignore a received Upgrade
header field if it wishes to continue using the current protocol on
that connection. Upgrade cannot be used to insist on a protocol
change.

```
  Upgrade          = 1#protocol

  protocol         = protocol-name ["/" protocol-version]
  protocol-name    = token
  protocol-version = token
```

Although protocol names are registered with a preferred case,
recipients SHOULD use case-insensitive comparison when matching each
protocol-name to supported protocols.

A server that sends a 101 (Switching Protocols) response MUST send an
Upgrade header field to indicate the new protocol(s) to which the
connection is being switched; if multiple protocol layers are being
switched, the sender MUST list the protocols in layer-ascending
order. A server MUST NOT switch to a protocol that was not indicated
by the client in the corresponding request's Upgrade header field. A

server MAY choose to ignore the order of preference indicated by the client and select the new protocol(s) based on other factors, such as the nature of the request or the current load on the server.

A server that sends a 426 (Upgrade Required) response MUST send an Upgrade header field to indicate the acceptable protocols, in order of descending preference.

A server MAY send an Upgrade header field in any other response to advertise that it implements support for upgrading to the listed protocols, in order of descending preference, when appropriate for a future request.

The following is a hypothetical example sent by a client:

```
GET /hello HTTP/1.1
Host: www.example.com
Connection: upgrade
Upgrade: websocket, IRC/6.9, RTA/x11
```

The capabilities and nature of the application-level communication after the protocol change is entirely dependent upon the new protocol(s) chosen. However, immediately after sending the 101 (Switching Protocols) response, the server is expected to continue responding to the original request as if it had received its equivalent within the new protocol (i.e., the server still has an outstanding request to satisfy after the protocol has been changed, and is expected to do so without requiring the request to be repeated).

For example, if the Upgrade header field is received in a GET request and the server decides to switch protocols, it first responds with a 101 (Switching Protocols) message in HTTP/1.1 and then immediately follows that with the new protocol's equivalent of a response to a GET on the target resource. This allows a connection to be upgraded to protocols with the same semantics as HTTP without the latency cost of an additional round trip. A server MUST NOT switch protocols unless the received message semantics can be honored by the new protocol; an OPTIONS request can be honored by any protocol.

The following is an example response to the above hypothetical request:

```
      HTTP/1.1 101 Switching Protocols
      Connection: upgrade
      Upgrade: websocket

      [... data stream switches to websocket with an appropriate response
      (as defined by new protocol) to the "GET /hello" request ...]
```

When Upgrade is sent, the sender MUST also send a Connection header
field (Section 9.1) that contains an "upgrade" connection option, in
order to prevent Upgrade from being accidentally forwarded by
intermediaries that might not implement the listed protocols. A
server MUST ignore an Upgrade header field that is received in an
HTTP/1.0 request.

A client cannot begin using an upgraded protocol on the connection
until it has completely sent the request message (i.e., the client
can't change the protocol it is sending in the middle of a message).
If a server receives both an Upgrade and an Expect header field with
the "100-continue" expectation (Section 8.1.1 of [Semantics]), the
server MUST send a 100 (Continue) response before sending a 101
(Switching Protocols) response.

The Upgrade header field only applies to switching protocols on top
of the existing connection; it cannot be used to switch the
underlying connection (transport) protocol, nor to switch the
existing communication to a different connection. For those
purposes, it is more appropriate to use a 3xx (Redirection) response
(Section 9.4 of [Semantics]).

9.9.1. Upgrade Protocol Names

This specification only defines the protocol name "HTTP" for use by
the family of Hypertext Transfer Protocols, as defined by the HTTP
version rules of Section 3.5 of [Semantics] and future updates to
this specification. Additional protocol names ought to be registered
using the registration procedure defined in Section 9.9.2.

```
+------+------------------+-------------------+------------------+
| Name | Description      | Expected Version  | Reference        |
|      |                  | Tokens            |                  |
+------+------------------+-------------------+------------------+
| HTTP | Hypertext        | any DIGIT.DIGIT   | Section 3.5 of   |
|      | Transfer Protocol | (e.g, "2.0")      | [Semantics]      |
+------+------------------+-------------------+------------------+
```

[9.9.2](#). **Upgrade Token Registry**

   The "Hypertext Transfer Protocol (HTTP) Upgrade Token Registry"
   defines the namespace for protocol-name tokens used to identify
   protocols in the Upgrade header field. The registry is maintained at
   <https://www.iana.org/assignments/http-upgrade-tokens>.

   Each registered protocol name is associated with contact information
   and an optional set of specifications that details how the connection
   will be processed after it has been upgraded.

   Registrations happen on a "First Come First Served" basis (see
   [Section 4.4 of [RFC8126]](#)) and are subject to the following rules:

   1. A protocol-name token, once registered, stays registered forever.

   2. A protocol-name token is case-insensitive and registered with the
       preferred case to be generated by senders.

   3. The registration MUST name a responsible party for the
       registration.

   4. The registration MUST name a point of contact.

   5. The registration MAY name a set of specifications associated with
       that token. Such specifications need not be publicly available.

   6. The registration SHOULD name a set of expected "protocol-version"
       tokens associated with that token at the time of registration.

   7. The responsible party MAY change the registration at any time.
       The IANA will keep a record of all such changes, and make them
       available upon request.

   8. The IESG MAY reassign responsibility for a protocol token. This
       will normally only be used in the case when a responsible party
       cannot be contacted.

[10](#). **Enclosing Messages as Data**

[10.1](#). **Media Type message/http**

   The message/http media type can be used to enclose a single HTTP
   request or response message, provided that it obeys the MIME
   restrictions for all "message" types regarding line length and
   encodings.

   Type name: message

      Subtype name: http

      Required parameters: N/A

      Optional parameters: version, msgtype

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

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

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

      Security considerations: see Section 11

      Interoperability considerations: N/A

      Published specification: This specification (see Section 10.1).

      Applications that use this media type: N/A

      Fragment identifier considerations: N/A

      Additional information:

         Magic number(s): N/A

         Deprecated alias names for this type: N/A

         File extension(s): N/A

         Macintosh file type code(s): N/A

      Person and email address to contact for further information:
         See Authors' Addresses section.

      Intended usage: COMMON

      Restrictions on usage: N/A

      Author: See Authors' Addresses section.

   Change controller: IESG

[10.2]. Media Type application/http

   The application/http media type can be used to enclose a pipeline of
   one or more HTTP request or response messages (not intermixed).

   Type name: application

   Subtype name: http

   Required parameters: N/A

   Optional parameters: version, msgtype

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

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

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

   Security considerations: see Section 11

   Interoperability considerations: N/A

   Published specification: This specification (see Section 10.2).

   Applications that use this media type: N/A

   Fragment identifier considerations: N/A

   Additional information:

      Deprecated alias names for this type: N/A

      Magic number(s): N/A

      File extension(s): N/A

      Macintosh file type code(s): N/A

   Person and email address to contact for further information:
      See Authors' Addresses section.

   Intended usage: COMMON

   Restrictions on usage: N/A

   Author: See Authors' Addresses section.

   Change controller: IESG

11. Security Considerations

   This section is meant to inform developers, information providers,
   and users of known security considerations relevant to HTTP message
   syntax, parsing, and routing. Security considerations about HTTP
   semantics and payloads are addressed in [Semantics].

11.1. Response Splitting

   Response splitting (a.k.a, CRLF injection) is a common technique,
   used in various attacks on Web usage, that exploits the line-based
   nature of HTTP message framing and the ordered association of
   requests to responses on persistent connections [Klein]. This
   technique can be particularly damaging when the requests pass through
   a shared cache.

   Response splitting exploits a vulnerability in servers (usually
   within an application server) where an attacker can send encoded data
   within some parameter of the request that is later decoded and echoed
   within any of the response header fields of the response. If the
   decoded data is crafted to look like the response has ended and a
   subsequent response has begun, the response has been split and the
   content within the apparent second response is controlled by the
   attacker. The attacker can then make any other request on the same
   persistent connection and trick the recipients (including
   intermediaries) into believing that the second half of the split is
   an authoritative answer to the second request.

   For example, a parameter within the request-target might be read by
   an application server and reused within a redirect, resulting in the
   same parameter being echoed in the Location header field of the
   response. If the parameter is decoded by the application and not
   properly encoded when placed in the response field, the attacker can
   send encoded CRLF octets and other content that will make the
   application's single response look like two or more responses.

   A common defense against response splitting is to filter requests for
   data that looks like encoded CR and LF (e.g., "%0D" and "%0A").
   However, that assumes the application server is only performing URI
   decoding, rather than more obscure data transformations like charset
   transcoding, XML entity translation, base64 decoding, sprintf
   reformatting, etc. A more effective mitigation is to prevent
   anything other than the server's core protocol libraries from sending
   a CR or LF within the header section, which means restricting the
   output of header fields to APIs that filter for bad octets and not
   allowing application servers to write directly to the protocol
   stream.

11.2. Request Smuggling

   Request smuggling ([Linhart]) is a technique that exploits
   differences in protocol parsing among various recipients to hide
   additional requests (which might otherwise be blocked or disabled by
   policy) within an apparently harmless request. Like response
   splitting, request smuggling can lead to a variety of attacks on HTTP
   usage.

   This specification has introduced new requirements on request
   parsing, particularly with regard to message framing in Section 6.3,
   to reduce the effectiveness of request smuggling.

11.3. Message Integrity

   HTTP does not define a specific mechanism for ensuring message
   integrity, instead relying on the error-detection ability of
   underlying transport protocols and the use of length or chunk-
   delimited framing to detect completeness. Additional integrity
   mechanisms, such as hash functions or digital signatures applied to
   the content, can be selectively added to messages via extensible
   metadata header fields. Historically, the lack of a single integrity
   mechanism has been justified by the informal nature of most HTTP
   communication. However, the prevalence of HTTP as an information
   access mechanism has resulted in its increasing use within
   environments where verification of message integrity is crucial.

   User agents are encouraged to implement configurable means for
   detecting and reporting failures of message integrity such that those
   means can be enabled within environments for which integrity is
   necessary. For example, a browser being used to view medical history
   or drug interaction information needs to indicate to the user when
   such information is detected by the protocol to be incomplete,
   expired, or corrupted during transfer. Such mechanisms might be
   selectively enabled via user agent extensions or the presence of
   message integrity metadata in a response. At a minimum, user agents

   ought to provide some indication that allows a user to distinguish
   between a complete and incomplete response message (Section 8) when
   such verification is desired.

11.4. Message Confidentiality

   HTTP relies on underlying transport protocols to provide message
   confidentiality when that is desired. HTTP has been specifically
   designed to be independent of the transport protocol, such that it
   can be used over many different forms of encrypted connection, with
   the selection of such transports being identified by the choice of
   URI scheme or within user agent configuration.

   The "https" scheme can be used to identify resources that require a
   confidential connection, as described in Section 2.5.2 of
   [Semantics].

12. IANA Considerations

   The change controller for the following registrations is: "IETF
   (iesg@ietf.org) - Internet Engineering Task Force".

12.1. Header Field Registration

   Please update the "Hypertext Transfer Protocol (HTTP) Header Field
   Registry" registry at <https://www.iana.org/assignments/http-headers>
   with the header field names listed in the two tables of Section 5.

12.2. Media Type Registration

   Please update the "Media Types" registry at
   <https://www.iana.org/assignments/media-types> with the registration
   information in Section 10.1 and Section 10.2 for the media types
   "message/http" and "application/http", respectively.

12.3. Transfer Coding Registration

   Please update the "HTTP Transfer Coding Registry" at
   <https://www.iana.org/assignments/http-parameters/> with the
   registration procedure of Section 7.3 and the content coding names
   summarized in the table of Section 7.

12.4. Upgrade Token Registration

   Please update the "Hypertext Transfer Protocol (HTTP) Upgrade Token
   Registry" at <https://www.iana.org/assignments/http-upgrade-tokens>
   with the registration procedure of Section 9.9.2 and the upgrade
   token names summarized in the table of Section 9.9.1.

13. References

13.1. Normative References

[Caching]   Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke,
            Ed., "HTTP Caching", draft-ietf-httpbis-cache-06 (work in
            progress), November 2019.

[RFC1950]   Deutsch, L. and J-L. Gailly, "ZLIB Compressed Data Format
            Specification version 3.3", RFC 1950,
            DOI 10.17487/RFC1950, May 1996,
            <https://www.rfc-editor.org/info/rfc1950>.

[RFC1951]   Deutsch, P., "DEFLATE Compressed Data Format Specification
            version 1.3", RFC 1951, DOI 10.17487/RFC1951, May 1996,
            <https://www.rfc-editor.org/info/rfc1951>.

[RFC1952]   Deutsch, P., Gailly, J-L., Adler, M., Deutsch, L., and G.
            Randers-Pehrson, "GZIP file format specification version
            4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996,
            <https://www.rfc-editor.org/info/rfc1952>.

[RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
            Requirement Levels", BCP 14, RFC 2119,
            DOI 10.17487/RFC2119, March 1997,
            <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986]   Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform
            Resource Identifier (URI): Generic Syntax", STD 66,
            RFC 3986, DOI 10.17487/RFC3986, January 2005,
            <https://www.rfc-editor.org/info/rfc3986>.

[RFC5234]   Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax
            Specifications: ABNF", STD 68, RFC 5234,
            DOI 10.17487/RFC5234, January 2008,
            <https://www.rfc-editor.org/info/rfc5234>.

[RFC7405]   Kyzivat, P., "Case-Sensitive String Support in ABNF",
            RFC 7405, DOI 10.17487/RFC7405, December 2014,
            <https://www.rfc-editor.org/info/rfc7405>.

[RFC8446]   Rescorla, E., "The Transport Layer Security (TLS) Protocol
            Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018,
            <https://www.rfc-editor.org/info/rfc8446>.

[Semantics]
          Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke,
          Ed., "HTTP Semantics", draft-ietf-httpbis-semantics-06
          (work in progress), November 2019.

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

[Welch]   Welch, T., "A Technique for High-Performance Data
          Compression", IEEE Computer 17(6), June 1984.

13.2. Informative References

[Err4667] RFC Errata, Erratum ID 4667, RFC 7230,
          <https://www.rfc-editor.org/errata/eid4667>.

[Klein]   Klein, A., "Divide and Conquer - HTTP Response Splitting,
          Web Cache Poisoning Attacks, and Related Topics", March
          2004, <http://packetstormsecurity.com/papers/general/
          whitepaper_httpresponse.pdf>.

[Linhart] Linhart, C., Klein, A., Heled, R., and S. Orrin, "HTTP
          Request Smuggling", June 2005,
          <http://www.watchfire.com/news/whitepapers.aspx>.

[RFC1945] Berners-Lee, T., Fielding, R., and H. Nielsen, "Hypertext
          Transfer Protocol -- HTTP/1.0", RFC 1945,
          DOI 10.17487/RFC1945, May 1996,
          <https://www.rfc-editor.org/info/rfc1945>.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail
          Extensions (MIME) Part One: Format of Internet Message
          Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996,
          <https://www.rfc-editor.org/info/rfc2045>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail
          Extensions (MIME) Part Two: Media Types", RFC 2046,
          DOI 10.17487/RFC2046, November 1996,
          <https://www.rfc-editor.org/info/rfc2046>.

[RFC2049] Freed, N. and N. Borenstein, "Multipurpose Internet Mail
          Extensions (MIME) Part Five: Conformance Criteria and
          Examples", RFC 2049, DOI 10.17487/RFC2049, November 1996,
          <https://www.rfc-editor.org/info/rfc2049>.

   [RFC2068] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., and T.
              Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1",
              RFC 2068, DOI 10.17487/RFC2068, January 1997,
              <https://www.rfc-editor.org/info/rfc2068>.

   [RFC2557] Palme, F., Hopmann, A., Shelness, N., and E. Stefferud,
              "MIME Encapsulation of Aggregate Documents, such as HTML
              (MHTML)", RFC 2557, DOI 10.17487/RFC2557, March 1999,
              <https://www.rfc-editor.org/info/rfc2557>.

   [RFC5322] Resnick, P., "Internet Message Format", RFC 5322,
              DOI 10.17487/RFC5322, October 2008,
              <https://www.rfc-editor.org/info/rfc5322>.

   [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Message Syntax and Routing",
              RFC 7230, DOI 10.17487/RFC7230, June 2014,
              <https://www.rfc-editor.org/info/rfc7230>.

   [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Semantics and Content", RFC 7231,
              DOI 10.17487/RFC7231, June 2014,
              <https://www.rfc-editor.org/info/rfc7231>.

   [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for
              Writing an IANA Considerations Section in RFCs", BCP 26,
              RFC 8126, DOI 10.17487/RFC8126, June 2017,
              <https://www.rfc-editor.org/info/rfc8126>.

Appendix A. Collected ABNF

   In the collected ABNF below, list rules are expanded as per
   Section 12 of [Semantics].

   BWS = <BWS, see [Semantics], Section 4.3>

   Connection = [connection-option] *(OWS "," OWS [connection-option
    ])

   HTTP-message = start-line CRLF *(header-field CRLF) CRLF [
    message-body]
   HTTP-name = %x48.54.54.50 ; HTTP
   HTTP-version = HTTP-name "/" DIGIT "." DIGIT

   OWS = <OWS, see [Semantics], Section 4.3>

   RWS = <RWS, see [Semantics], Section 4.3>

   TE = [t-codings] *(OWS "," OWS [t-codings])
   Transfer-Encoding = [transfer-coding] *(OWS "," OWS [
    transfer-coding])

   Upgrade = [protocol] *(OWS "," OWS [protocol])

   absolute-URI = <absolute-URI, see [RFC3986], Section 4.3>
   absolute-form = absolute-URI
   absolute-path = <absolute-path, see [Semantics], Section 2.4>
   asterisk-form = "*"
   authority = <authority, see [RFC3986], Section 3.2>
   authority-form = authority

   chunk = chunk-size [chunk-ext] CRLF chunk-data CRLF
   chunk-data = 1*OCTET
   chunk-ext = *(BWS ";" BWS chunk-ext-name [BWS "=" BWS chunk-ext-val
    ])
   chunk-ext-name = token
   chunk-ext-val = token / quoted-string
   chunk-size = 1*HEXDIG
   chunked-body = *chunk last-chunk trailer-section CRLF
   comment = <comment, see [Semantics], Section 4.2.3>
   connection-option = token

   field-name = <field-name, see [Semantics], Section 4.2>
   field-value = <field-value, see [Semantics], Section 4.2>

   header-field = field-name ":" OWS field-value OWS

```
   last-chunk = 1*"0" [ chunk-ext ] CRLF

   message-body = *OCTET
   method = token

   obs-fold = OWS CRLF RWS
   obs-text = <obs-text, see [Semantics], Section 4.2.3>
   origin-form = absolute-path [ "?" query ]

   port = <port, see [RFC3986], Section 3.2.3>
   protocol = protocol-name [ "/" protocol-version ]
   protocol-name = token
   protocol-version = token

   query = <query, see [RFC3986], Section 3.4>
   quoted-string = <quoted-string, see [Semantics], Section 4.2.3>

   rank = ( "0" [ "." *3DIGIT ] ) / ( "1" [ "." *3"0" ] )
   reason-phrase = 1*( HTAB / SP / VCHAR / obs-text )
   request-line = method SP request-target SP HTTP-version
   request-target = origin-form / absolute-form / authority-form /
    asterisk-form

   start-line = request-line / status-line
   status-code = 3DIGIT
   status-line = HTTP-version SP status-code SP [ reason-phrase ]

   t-codings = "trailers" / ( transfer-coding [ t-ranking ] )
   t-ranking = OWS ";" OWS "q=" rank
   token = <token, see [Semantics], Section 4.2.3>
   trailer-section = *( header-field CRLF )
   transfer-coding = token *( OWS ";" OWS transfer-parameter )
   transfer-parameter = token BWS "=" BWS ( token / quoted-string )

   uri-host = <host, see [RFC3986], Section 3.2.2>
```

Appendix B. Differences between HTTP and MIME

   HTTP/1.1 uses many of the constructs defined for the Internet Message
   Format [RFC5322] and the Multipurpose Internet Mail Extensions (MIME)
   [RFC2045] to allow a message body to be transmitted in an open
   variety of representations and with extensible header fields.
   However, RFC 2045 is focused only on email; applications of HTTP have
   many characteristics that differ from email; hence, HTTP has features
   that differ from MIME. These differences were carefully chosen to
   optimize performance over binary connections, to allow greater
   freedom in the use of new media types, to make date comparisons

easier, and to acknowledge the practice of some early HTTP servers
and clients.

This appendix describes specific areas where HTTP differs from MIME.
Proxies and gateways to and from strict MIME environments need to be
aware of these differences and provide the appropriate conversions
where necessary.

B.1. MIME-Version

HTTP is not a MIME-compliant protocol. However, messages can include
a single MIME-Version header field to indicate what version of the
MIME protocol was used to construct the message. Use of the MIME-
Version header field indicates that the message is in full
conformance with the MIME protocol (as defined in [RFC2045]).
Senders are responsible for ensuring full conformance (where
possible) when exporting HTTP messages to strict MIME environments.

B.2. Conversion to Canonical Form

MIME requires that an Internet mail body part be converted to
canonical form prior to being transferred, as described in Section 4
of [RFC2049]. Section 6.1.1.2 of [Semantics] describes the forms
allowed for subtypes of the "text" media type when transmitted over
HTTP. [RFC2046] requires that content with a type of "text"
represent line breaks as CRLF and forbids the use of CR or LF outside
of line break sequences. HTTP allows CRLF, bare CR, and bare LF to
indicate a line break within text content.

A proxy or gateway from HTTP to a strict MIME environment ought to
translate all line breaks within text media types to the RFC 2049
canonical form of CRLF. Note, however, this might be complicated by
the presence of a Content-Encoding and by the fact that HTTP allows
the use of some charsets that do not use octets 13 and 10 to
represent CR and LF, respectively.

Conversion will break any cryptographic checksums applied to the
original content unless the original content is already in canonical
form. Therefore, the canonical form is recommended for any content
that uses such checksums in HTTP.

B.3. Conversion of Date Formats

HTTP/1.1 uses a restricted set of date formats (Section 10.1.1.1 of
[Semantics]) to simplify the process of date comparison. Proxies and
gateways from other protocols ought to ensure that any Date header
field present in a message conforms to one of the HTTP/1.1 formats
and rewrite the date if necessary.

B.4. Conversion of Content-Encoding

   MIME does not include any concept equivalent to HTTP/1.1's Content-
   Encoding header field. Since this acts as a modifier on the media
   type, proxies and gateways from HTTP to MIME-compliant protocols
   ought to either change the value of the Content-Type header field or
   decode the representation before forwarding the message. (Some
   experimental applications of Content-Type for Internet mail have used
   a media-type parameter of ";conversions=<content-coding>" to perform
   a function equivalent to Content-Encoding. However, this parameter
   is not part of the MIME standards).

B.5. Conversion of Content-Transfer-Encoding

   HTTP does not use the Content-Transfer-Encoding field of MIME.
   Proxies and gateways from MIME-compliant protocols to HTTP need to
   remove any Content-Transfer-Encoding prior to delivering the response
   message to an HTTP client.

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

B.6. MHTML and Line Length Limitations

   HTTP implementations that share code with MHTML [RFC2557]
   implementations need to be aware of MIME line length limitations.
   Since HTTP does not have this limitation, HTTP does not fold long
   lines. MHTML messages being transported by HTTP follow all
   conventions of MHTML, including line length limitations and folding,
   canonicalization, etc., since HTTP transfers message-bodies as
   payload and, aside from the "multipart/byteranges" type
   (Section 6.3.5 of [Semantics]), does not interpret the content or any
   MIME header lines that might be contained therein.

Appendix C. HTTP Version History

   HTTP has been in use since 1990. The first version, later referred
   to as HTTP/0.9, was a simple protocol for hypertext data transfer
   across the Internet, using only a single request method (GET) and no
   metadata. HTTP/1.0, as defined by [RFC1945], added a range of
   request methods and MIME-like messaging, allowing for metadata to be
   transferred and modifiers placed on the request/response semantics.
   However, HTTP/1.0 did not sufficiently take into consideration the

effects of hierarchical proxies, caching, the need for persistent
connections, or name-based virtual hosts. The proliferation of
incompletely implemented applications calling themselves "HTTP/1.0"
further necessitated a protocol version change in order for two
communicating applications to determine each other's true
capabilities.

HTTP/1.1 remains compatible with HTTP/1.0 by including more stringent
requirements that enable reliable implementations, adding only those
features that can either be safely ignored by an HTTP/1.0 recipient
or only be sent when communicating with a party advertising
conformance with HTTP/1.1.

HTTP/1.1 has been designed to make supporting previous versions easy.
A general-purpose HTTP/1.1 server ought to be able to understand any
valid request in the format of HTTP/1.0, responding appropriately
with an HTTP/1.1 message that only uses features understood (or
safely ignored) by HTTP/1.0 clients. Likewise, an HTTP/1.1 client
can be expected to understand any valid HTTP/1.0 response.

Since HTTP/0.9 did not support header fields in a request, there is
no mechanism for it to support name-based virtual hosts (selection of
resource by inspection of the Host header field). Any server that
implements name-based virtual hosts ought to disable support for
HTTP/0.9. Most requests that appear to be HTTP/0.9 are, in fact,
badly constructed HTTP/1.x requests caused by a client failing to
properly encode the request-target.

C.1. Changes from HTTP/1.0

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

C.1.1. Multihomed Web Servers

The requirements that clients and servers support the Host header
field (Section 5.4 of [Semantics]), report an error if it is missing
from an HTTP/1.1 request, and accept absolute URIs (Section 3.2) are
among the most important changes defined by HTTP/1.1.

Older HTTP/1.0 clients assumed a one-to-one relationship of IP
addresses and servers; there was no other established mechanism for
distinguishing the intended server of a request than the IP address
to which that request was directed. The Host header field was
introduced during the development of HTTP/1.1 and, though it was
quickly implemented by most HTTP/1.0 browsers, additional
requirements were placed on all HTTP/1.1 requests in order to ensure
complete adoption. At the time of this writing, most HTTP-based

services are dependent upon the Host header field for targeting
requests.

C.1.2. Keep-Alive Connections

In HTTP/1.0, each connection is established by the client prior to
the request and closed by the server after sending the response.
However, some implementations implement the explicitly negotiated
("Keep-Alive") version of persistent connections described in
Section 19.7.1 of [RFC2068].

Some clients and servers might wish to be compatible with these
previous approaches to persistent connections, by explicitly
negotiating for them with a "Connection: keep-alive" request header
field. However, some experimental implementations of HTTP/1.0
persistent connections are faulty; for example, if an HTTP/1.0 proxy
server doesn't understand Connection, it will erroneously forward
that header field to the next inbound server, which would result in a
hung connection.

One attempted solution was the introduction of a Proxy-Connection
header field, targeted specifically at proxies. In practice, this
was also unworkable, because proxies are often deployed in multiple
layers, bringing about the same problem discussed above.

As a result, clients are encouraged not to send the Proxy-Connection
header field in any requests.

Clients are also encouraged to consider the use of Connection: keep-
alive in requests carefully; while they can enable persistent
connections with HTTP/1.0 servers, clients using them will need to
monitor the connection for "hung" requests (which indicate that the
client ought stop sending the header field), and this mechanism ought
not be used by clients at all when a proxy is being used.

C.1.3. Introduction of Transfer-Encoding

HTTP/1.1 introduces the Transfer-Encoding header field (Section 6.1).
Transfer codings need to be decoded prior to forwarding an HTTP
message over a MIME-compliant protocol.

C.2. Changes from RFC 7230

Most of the sections introducing HTTP's design goals, history,
architecture, conformance criteria, protocol versioning, URIs,
message routing, and header fields have been moved to [Semantics].
This document has been reduced to just the messaging syntax and
connection management requirements specific to HTTP/1.1.

Trailer field semantics now transcend the specifics of chunked
encoding. The decoding algorithm for chunked (Section 7.1.3) has
been updated to encourage storage/forwarding of trailer fields
separately from the header section, to only allow merging into the
header section if the recipient knows the corresponding field
definition permits and defines how to merge, and otherwise to discard
the trailer fields instead of merging. The trailer part is now
called the trailer section to be more consistent with the header
section and more distinct from a body part (Section 7.1.2).

In the ABNF for chunked extensions, re-introduced (bad) whitespace
around ";" and "=" (Section 7.1.1). Whitespace was removed in
[RFC7230], but that change was found to break existing
implementations (see [Err4667]).

Disallowed transfer coding parameters called "q" in order to avoid
conflicts with the use of ranks in the TE header field (Section 7.3).

Appendix D. Change Log

This section is to be removed before publishing as an RFC.

D.1. Between RFC7230 and draft 00

The changes were purely editorial:

o Change boilerplate and abstract to indicate the "draft" status,
   and update references to ancestor specifications.

o Adjust historical notes.

o Update links to sibling specifications.

o Replace sections listing changes from RFC 2616 by new empty
   sections referring to RFC 723x.

o Remove acknowledgements specific to RFC 723x.

o Move "Acknowledgements" to the very end and make them unnumbered.

D.2. Since draft-ietf-httpbis-messaging-00

The changes in this draft are editorial, with respect to HTTP as a
whole, to move all core HTTP semantics into [Semantics]:

o Moved introduction, architecture, conformance, and ABNF extensions
   from RFC 7230 (Messaging) to semantics [Semantics].

o Moved discussion of MIME differences from RFC 7231 (Semantics) to
   Appendix B since they mostly cover transforming 1.1 messages.

o Moved all extensibility tips, registration procedures, and
   registry tables from the IANA considerations to normative
   sections, reducing the IANA considerations to just instructions
   that will be removed prior to publication as an RFC.

D.3. Since draft-ietf-httpbis-messaging-01

o Cite RFC 8126 instead of RFC 5226 (<https://github.com/httpwg/
   http-core/issues/75>)

o Resolved erratum 4779, no change needed here
   (<https://github.com/httpwg/http-core/issues/87>,
   <https://www.rfc-editor.org/errata/eid4779>)

o In Section 7, fixed prose claiming transfer parameters allow bare
   names (<https://github.com/httpwg/http-core/issues/88>,
   <https://www.rfc-editor.org/errata/eid4839>)

o Resolved erratum 4225, no change needed here
   (<https://github.com/httpwg/http-core/issues/90>,
   <https://www.rfc-editor.org/errata/eid4225>)

o Replace "response code" with "response status code"
   (<https://github.com/httpwg/http-core/issues/94>,
   <https://www.rfc-editor.org/errata/eid4050>)

o In Section 9.4, clarify statement about HTTP/1.0 keep-alive
   (<https://github.com/httpwg/http-core/issues/96>,
   <https://www.rfc-editor.org/errata/eid4205>)

o In Section 7.1.1, re-introduce (bad) whitespace around ";" and "="
   (<https://github.com/httpwg/http-core/issues/101>,
   <https://www.rfc-editor.org/errata/eid4667>, <https://www.rfc-
   editor.org/errata/eid4825>)

o In Section 7.3, state that transfer codings should not use
   parameters named "q" (<https://github.com/httpwg/http-core/
   issues/15>, <https://www.rfc-editor.org/errata/eid4683>)

o In Section 7, mark coding name "trailers" as reserved in the IANA
   registry (<https://github.com/httpwg/http-core/issues/108>)

D.4. Since draft-ietf-httpbis-messaging-02

   o In Section 4, explain why the reason phrase should be ignored by
      clients (<https://github.com/httpwg/http-core/issues/60>).

   o Add Section 9.3 to explain how request/response correlation is
      performed (<https://github.com/httpwg/http-core/issues/145>)

D.5. Since draft-ietf-httpbis-messaging-03

   o In Section 9.3, caution against treating data on a connection as
      part of a not-yet-issued request (<https://github.com/httpwg/http-
      core/issues/26>)

   o In Section 7, remove the predefined codings from the ABNF and make
      it generic instead (<https://github.com/httpwg/http-core/
      issues/66>)

   o Use RFC 7405 ABNF notation for case-sensitive string constants
      (<https://github.com/httpwg/http-core/issues/133>)

D.6. Since draft-ietf-httpbis-messaging-04

   o In Section 9.9, clarify that protocol-name is to be matched case-
      insensitively (<https://github.com/httpwg/http-core/issues/8>)

   o In Section 5.2, add leading optional whitespace to obs-fold ABNF
      (<https://github.com/httpwg/http-core/issues/19>,
      <https://www.rfc-editor.org/errata/eid4189>)

   o In Section 4, add clarifications about empty reason phrases
      (<https://github.com/httpwg/http-core/issues/197>)

   o Move discussion of retries from Section 9.4.1 into [Semantics]
      (<https://github.com/httpwg/http-core/issues/230>)

D.7. Since draft-ietf-httpbis-messaging-05

   o In Section 7.1.2, the trailer part has been renamed the trailer
      section (for consistency with the header section) and trailers are
      no longer merged as header fields by default, but rather can be
      discarded, kept separate from header fields, or merged with header
      fields only if understood and defined as being mergeable
      (<https://github.com/httpwg/http-core/issues/16>)

   o In Section 2.1 and related Sections, move the trailing CRLF from
      the line grammars into the message format
      (<https://github.com/httpwg/http-core/issues/62>)

   o Moved Section 2.3 down (<https://github.com/httpwg/http-core/
      issues/68>)

   o In Section 9.9, use 'websocket' instead of 'HTTP/2.0' in examples
      (<https://github.com/httpwg/http-core/issues/112>)

   o Move version non-specific text from Section 6 into semantics as
      "payload body" (<https://github.com/httpwg/http-core/issues/159>)

   o In Section 9.8, add text from RFC 2818
      (<https://github.com/httpwg/http-core/issues/236>)

Index

Acknowledgments

   See Appendix "Acknowledgments" of [Semantics].

Authors' Addresses

   Roy T. Fielding (editor)
   Adobe
   345 Park Ave
   San Jose, CA 95110
   United States of America

   EMail: fielding@gbiv.com
   URI: https://roy.gbiv.com/

   Mark Nottingham (editor)
   Fastly

   EMail: mnot@mnot.net
   URI: https://www.mnot.net/

   Julian F. Reschke (editor)
   greenbytes GmbH
   Hafenweg 16
   Muenster 48155
   Germany

   EMail: julian.reschke@greenbytes.de
   URI: https://greenbytes.de/tech/webdav/