Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing

Abstract

   The Hypertext Transfer Protocol (HTTP) is a stateless application-
   level protocol for distributed, collaborative, hypertext information
   systems. This document provides an overview of HTTP architecture and
   its associated terminology, defines the "http" and "https" Uniform
   Resource Identifier (URI) schemes, defines the HTTP/1.1 message
   syntax and parsing requirements, and describes related security
   concerns for implementations.

Status of This Memo

   This is an Internet Standards Track document.

   This document is a product of the Internet Engineering Task Force
   (IETF). It represents the consensus of the IETF community. It has
   received public review and has been approved for publication by the
   Internet Engineering Steering Group (IESG). Further information on
   Internet Standards is available in Section 2 of RFC 5741.

   Information about the current status of this document, any errata,
   and how to provide feedback on it may be obtained at
   http://www.rfc-editor.org/info/rfc7230.

Fielding & Reschke   Standards Track   [Page 1]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

Table of Contents

Fielding & Reschke   Standards Track   [Page 2]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

Fielding & Reschke   Standards Track   [Page 3]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

Fielding & Reschke   Standards Track   [Page 4]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

1. Introduction

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

1. "Message Syntax and Routing" (this document)

2. "Semantics and Content" [RFC7231]

3. "Conditional Requests" [RFC7232]

4. "Range Requests" [RFC7233]

5. "Caching" [RFC7234]

6. "Authentication" [RFC7235]

This HTTP/1.1 specification obsoletes RFC 2616 and RFC 2145 (on HTTP
versioning). This specification also updates the use of CONNECT to
establish a tunnel, previously defined in RFC 2817, and defines the
"https" URI scheme that was described informally in RFC 2818.

HTTP is a generic interface protocol for information systems. It is
designed to hide the details of how a service is implemented by
presenting a uniform interface to clients that is independent of the
types of resources provided. Likewise, servers do not need to be
aware of each client's purpose: an HTTP request can be considered in
isolation rather than being associated with a specific type of client
or a predetermined sequence of application steps. The result is a
protocol that can be used effectively in many different contexts and
for which implementations can evolve independently over time.

HTTP is also designed for use as an intermediation protocol for
translating communication to and from non-HTTP information systems.
HTTP proxies and gateways can provide access to alternative
information services by translating their diverse protocols into a
hypertext format that can be viewed and manipulated by clients in the
same way as HTTP services.

One consequence of this flexibility is that the protocol cannot be
defined in terms of what occurs behind the interface. Instead, we
are limited to defining the syntax of communication, the intent of
received communication, and the expected behavior of recipients. If
the communication is considered in isolation, then successful actions

Fielding & Reschke   Standards Track   [Page 5]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

ought to be reflected in corresponding changes to the observable
interface provided by servers. However, since multiple clients might
act in parallel and perhaps at cross-purposes, we cannot require that
such changes be observable beyond the scope of a single response.

This document describes the architectural elements that are used or
referred to in HTTP, defines the "http" and "https" URI schemes,
describes overall network operation and connection management, and
defines HTTP message framing and forwarding requirements. Our goal
is to define all of the mechanisms necessary for HTTP message
handling that are independent of message semantics, thereby defining
the complete set of requirements for message parsers and message-
forwarding intermediaries.

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 2.5.

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF)
notation of [RFC5234] with a list extension, defined in Section 7,
that allows for compact definition of comma-separated lists using a
'#' operator (similar to how the '*' operator indicates repetition).
Appendix B shows the collected grammar with all list operators
expanded to standard ABNF notation.

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).

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

2. Architecture

HTTP was created for the World Wide Web (WWW) architecture and has
evolved over time to support the scalability needs of a worldwide
hypertext system. Much of that architecture is reflected in the
terminology and syntax productions used to define HTTP.

Fielding & Reschke   Standards Track   [Page 6]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

2.1. Client/Server Messaging

   HTTP is a stateless request/response protocol that operates by
   exchanging messages (Section 3) across a reliable transport- or
   session-layer "connection" (Section 6). An HTTP "client" is a
   program that establishes a connection to a server for the purpose of
   sending one or more HTTP requests. An HTTP "server" is a program
   that accepts connections in order to service HTTP requests by sending
   HTTP responses.

   The terms "client" and "server" refer only to the roles that these
   programs perform for a particular connection. The same program might
   act as a client on some connections and a server on others. The term
   "user agent" refers to any of the various client programs that
   initiate a request, including (but not limited to) browsers, spiders
   (web-based robots), command-line tools, custom applications, and
   mobile apps. The term "origin server" refers to the program that can
   originate authoritative responses for a given target resource. The
   terms "sender" and "recipient" refer to any implementation that sends
   or receives a given message, respectively.

   HTTP relies upon the Uniform Resource Identifier (URI) standard
   [RFC3986] to indicate the target resource (Section 5.1) and
   relationships between resources. Messages are passed in a format
   similar to that used by Internet mail [RFC5322] and the Multipurpose
   Internet Mail Extensions (MIME) [RFC2045] (see Appendix A of
   [RFC7231] for the differences between HTTP and MIME messages).

   Most HTTP communication consists of a retrieval request (GET) for a
   representation of some resource identified by a URI. In the simplest
   case, this might be accomplished via a single bidirectional
   connection (===) between the user agent (UA) and the origin
   server (O).

```
            request   >
       UA ======================================= O
                                < response
```

   A client sends an HTTP request to a server in the form of a request
   message, beginning with a request-line that includes a method, URI,
   and protocol version (Section 3.1.1), followed by header fields
   containing request modifiers, client information, and representation
   metadata (Section 3.2), an empty line to indicate the end of the
   header section, and finally a message body containing the payload
   body (if any, Section 3.3).

Fielding & Reschke   Standards Track   [Page 7]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   A server responds to a client's request by sending one or more HTTP
   response messages, each beginning with a status line that includes
   the protocol version, a success or error code, and textual reason
   phrase (Section 3.1.2), possibly followed by header fields containing
   server information, resource metadata, and representation metadata
   (Section 3.2), an empty line to indicate the end of the header
   section, and finally a message body containing the payload body (if
   any, Section 3.3).

   A connection might be used for multiple request/response exchanges,
   as defined in Section 6.3.

   The following example illustrates a typical message exchange for a
   GET request (Section 4.3.1 of [RFC7231]) on the URI
   "http://www.example.com/hello.txt":

   Client request:

     GET /hello.txt HTTP/1.1
     User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
     Host: www.example.com
     Accept-Language: en, mi

   Server response:

     HTTP/1.1 200 OK
     Date: Mon, 27 Jul 2009 12:28:53 GMT
     Server: Apache
     Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
     ETag: "34aa387-d-1568eb00"
     Accept-Ranges: bytes
     Content-Length: 51
     Vary: Accept-Encoding
     Content-Type: text/plain

     Hello World! My payload includes a trailing CRLF.

2.2. Implementation Diversity

   When considering the design of HTTP, it is easy to fall into a trap
   of thinking that all user agents are general-purpose browsers and all
   origin servers are large public websites. That is not the case in
   practice. Common HTTP user agents include household appliances,
   stereos, scales, firmware update scripts, command-line programs,
   mobile apps, and communication devices in a multitude of shapes and
   sizes. Likewise, common HTTP origin servers include home automation

Fielding & Reschke   Standards Track   [Page 8]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   units, configurable networking components, office machines,
   autonomous robots, news feeds, traffic cameras, ad selectors, and
   video-delivery platforms.

   The term "user agent" does not imply that there is a human user
   directly interacting with the software agent at the time of a
   request. In many cases, a user agent is installed or configured to
   run in the background and save its results for later inspection (or
   save only a subset of those results that might be interesting or
   erroneous). Spiders, for example, are typically given a start URI
   and configured to follow certain behavior while crawling the Web as a
   hypertext graph.

   The implementation diversity of HTTP means that not all user agents
   can make interactive suggestions to their user or provide adequate
   warning for security or privacy concerns. In the few cases where
   this specification requires reporting of errors to the user, it is
   acceptable for such reporting to only be observable in an error
   console or log file. Likewise, requirements that an automated action
   be confirmed by the user before proceeding might be met via advance
   configuration choices, run-time options, or simple avoidance of the
   unsafe action; confirmation does not imply any specific user
   interface or interruption of normal processing if the user has
   already made that choice.

2.3. Intermediaries

   HTTP enables the use of intermediaries to satisfy requests through a
   chain of connections. There are three common forms of HTTP
   intermediary: proxy, gateway, and tunnel. In some cases, a single
   intermediary might act as an origin server, proxy, gateway, or
   tunnel, switching behavior based on the nature of each request.

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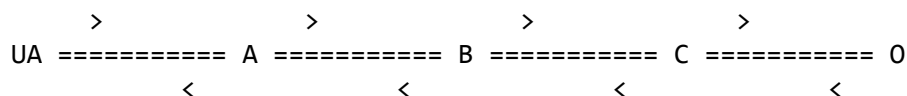
   The figure above shows three intermediaries (A, B, and C) between the
   user agent and origin server. A request or response message that
   travels the whole chain will pass through four separate connections.
   Some HTTP communication options might apply only to the connection
   with the nearest,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

Fielding & Reschke   Standards Track   [Page 9]

```
RFC 7230          HTTP/1.1 Message Syntax and Routing          June 2014
```

request. Likewise, later requests might be sent through a different
path of connections, often based on dynamic configuration for load
balancing.

The terms "upstream" and "downstream" are used to describe
directional requirements in relation to the message flow: all
messages flow from upstream to downstream. The terms "inbound" and
"outbound" are used to describe directional requirements in relation
to the request route: "inbound" means toward the origin server and
"outbound" means toward the user agent.

A "proxy" is a message-forwarding agent that is selected by the
client, usually via local configuration rules, to receive requests
for some type(s) of absolute URI and attempt to satisfy those
requests via translation through the HTTP interface. Some
translations are minimal, such as for proxy requests for "http" URIs,
whereas other requests might require translation to and from entirely
different application-level protocols. Proxies are often used to
group an organization's HTTP requests through a common intermediary
for the sake of security, annotation services, or shared caching.
Some proxies are designed to apply transformations to selected
messages or payloads while they are being forwarded, as described in
Section 5.7.2.

A "gateway" (a.k.a. "reverse proxy") is an intermediary that acts as
an origin server for the outbound connection but translates received
requests and forwards them inbound to another server or servers.
Gateways are often used to encapsulate legacy or untrusted
information services, to improve server performance through
"accelerator" caching, and to enable partitioning or load balancing
of HTTP services across multiple machines.

All HTTP requirements applicable to an origin server also apply to
the outbound communication of a gateway. A gateway communicates with
inbound servers using any protocol that it desires, including private
extensions to HTTP that are outside the scope of this specification.
However, an HTTP-to-HTTP gateway that wishes to interoperate with
third-party HTTP servers ought to conform to user agent requirements
on the gateway's inbound connection.

A "tunnel" acts as a blind relay between two connections without
changing the messages. Once active, a tunnel is not considered a
party to the HTTP communication, though the tunnel might have been
initiated by an HTTP request. A tunnel ceases to exist when both
ends of the relayed connection are closed. Tunnels are used to
extend a virtual connection through an intermediary, such as when
Transport Layer Security (TLS, [RFC5246]) is used to establish
confidential communication through a shared firewall proxy.

```
Fielding & Reschke          Standards Track                    [Page 10]
```

The above categories for intermediary only consider those acting as
participants in the HTTP communication. There are also
intermediaries that can act on lower layers of the network protocol
stack, filtering or redirecting HTTP traffic without the knowledge or
permission of message senders. Network intermediaries are
indistinguishable (at a protocol level) from a man-in-the-middle
attack, often introducing security flaws or interoperability problems
due to mistakenly violating HTTP semantics.

For example, an "interception proxy" [RFC3040] (also commonly known
as a "transparent proxy" [RFC1919] or "captive portal") differs from
an HTTP proxy because it is not selected by the client. Instead, an
interception proxy filters or redirects outgoing TCP port 80 packets
(and occasionally other common port traffic). Interception proxies
are commonly found on public network access points, as a means of
enforcing account subscription prior to allowing use of non-local
Internet services, and within corporate firewalls to enforce network
usage policies.

HTTP is defined as a stateless protocol, meaning that each request
message can be understood in isolation. Many implementations depend
on HTTP's stateless design in order to reuse proxied connections or
dynamically load balance requests across multiple servers. Hence, a
server MUST NOT assume that two requests on the same connection are
from the same user agent unless the connection is secured and
specific to that agent. Some non-standard HTTP extensions (e.g.,
[RFC4559]) have been known to violate this requirement, resulting in
security and interoperability problems.

2.4. Caches

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

The effect of a cache is that the request/response chain is shortened
if one of the participants along the chain has a cached response
applicable to that request. The following illustrates the resulting
chain if B has a cached copy of an earlier response from O (via C)
for a request that has not been cached by UA or A.

```
           >            >
      UA =========== A =========== B - - - - - - C - - - - - - O
                  <            <
```

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014
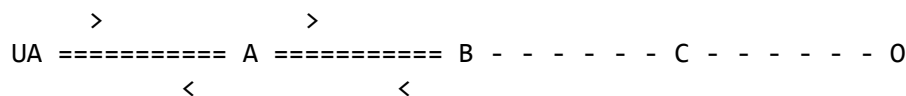
A response is "cacheable" if a cache is allowed to store a copy of
the response message for use in answering subsequent requests. Even
when a response is cacheable, there might be additional constraints
placed by the client or by the origin server on when that cached
response can be used for a particular request. HTTP requirements for
cache behavior and cacheable responses are defined in Section 2 of
[RFC7234].

There is a wide variety of architectures and configurations of caches
deployed across the World Wide Web and inside large organizations.
These include national hierarchies of proxy caches to save
transoceanic bandwidth, collaborative systems that broadcast or
multicast cache entries, archives of pre-fetched cache entries for
use in off-line or high-latency environments, and so on.

2.5. Conformance and Error Handling

This specification targets conformance criteria according to the role
of a participant in HTTP communication. Hence, HTTP requirements are
placed on senders, recipients, clients, servers, user agents,
intermediaries, origin servers, proxies, gateways, or caches,
depending on what behavior is being constrained by the requirement.
Additional (social) requirements are placed on implementations,
resource owners, and protocol element registrations when they apply
beyond the scope of a single communication.

The verb "generate" is used instead of "send" where a requirement
differentiates between creating a protocol element and merely
forwarding a received element downstream.

An implementation is considered conformant if it complies with all of
the requirements associated with the roles it partakes in HTTP.

Conformance includes both the syntax and semantics of protocol
elements. A sender MUST NOT generate protocol elements that convey a
meaning that is known by that sender to be false. A sender MUST NOT
generate protocol elements that do not match the grammar defined by
the corresponding ABNF rules. Within a given message, a sender MUST
NOT generate protocol elements or syntax alternatives that are only
allowed to be generated by participants in other roles (i.e., a role
that the sender does not have for that message).

When a received protocol element is parsed, the recipient MUST be
able to parse any value of reasonable length that is applicable to
the recipient's role and that matches the grammar defined by the
corresponding ABNF rules. Note, however, that some received protocol
elements might not be parsed. For example, an intermediary

Fielding & Reschke   Standards Track   [Page 12]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   forwarding a message might parse a header-field into generic
   field-name and field-value components, but then forward the header
   field without further parsing inside the field-value.

   HTTP does not have specific length limitations for many of its
   protocol elements because the lengths that might be appropriate will
   vary widely, depending on the deployment context and purpose of the
   implementation. Hence, interoperability between senders and
   recipients depends on shared expectations regarding what is a
   reasonable length for each protocol element. Furthermore, what is
   commonly understood to be a reasonable length for some protocol
   elements has changed over the course of the past two decades of HTTP
   use and is expected to continue changing in the future.

   At a minimum, a recipient MUST be able to parse and process protocol
   element lengths that are at least as long as the values that it
   generates for those same protocol elements in other messages. For
   example, an origin server that publishes very long URI references to
   its own resources needs to be able to parse and process those same
   references when received as a request target.

   A recipient MUST interpret a received protocol element according to
   the semantics defined for it by this specification, including
   extensions to this specification, unless the recipient has determined
   (through experience or configuration) that the sender incorrectly
   implements what is implied by those semantics. For example, an
   origin server might disregard the contents of a received
   Accept-Encoding header field if inspection of the User-Agent header
   field indicates a specific implementation version that is known to
   fail on receipt of certain content codings.

   Unless noted otherwise, a recipient MAY attempt to recover a usable
   protocol element from an invalid construct. HTTP does not define
   specific error handling mechanisms except when they have a direct
   impact on security, since different applications of the protocol
   require different error handling strategies. For example, a Web
   browser might wish to transparently recover from a response where the
   Location header field doesn't parse according to the ABNF, whereas a
   systems control client might consider any form of error recovery to
   be dangerous.

2.6. Protocol Versioning

   HTTP uses a "<major>.<minor>" numbering scheme to indicate versions
   of the protocol. This specification defines version "1.1". The
   protocol version as a whole indicates the sender's conformance with
   the set of requirements laid out in that version's corresponding
   specification of HTTP.

Fielding & Reschke   Standards Track   [Page 13]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

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

     HTTP-version = HTTP-name "/" DIGIT "." DIGIT
     HTTP-name   = %x48.54.54.50 ; "HTTP", case-sensitive

   The HTTP version number consists of two decimal digits separated by a
   "." (period or decimal point). The first digit ("major version")
   indicates the HTTP messaging syntax, whereas the second digit ("minor
   version") indicates the highest minor version within that major
   version to which the sender is conformant and able to understand for
   future communication. The minor version advertises the sender's
   communication capabilities even when the sender is only using a
   backwards-compatible subset of the protocol, thereby letting the
   recipient know that more advanced features can be used in response
   (by servers) or in future requests (by clients).

   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.

   The interpretation of a header field does not change between minor
   versions of the same major HTTP version, though the default behavior
   of a recipient in the absence of such a field can change. Unless
   specified otherwise, header fields defined in HTTP/1.1 are defined
   for all versions of HTTP/1.x. In particular, the Host and Connection
   header fields ought to be implemented by all HTTP/1.x implementations
   whether or not they advertise conformance with HTTP/1.1.

   New header fields can be introduced without changing the protocol
   version if their defined semantics allow them to be safely ignored by
   recipients that do not recognize them. Header field extensibility is
   discussed in Section 3.2.1.

   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 first line of an HTTP message 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

Fielding & Reschke   Standards Track   [Page 14]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   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 client SHOULD send a request version equal to the highest version
   to which the client is conformant and whose major version is no
   higher than the highest version supported by the server, if this is
   known. A client MUST NOT send a version to which it is not
   conformant.

   A client MAY send a lower request version if it is known that the
   server incorrectly implements the HTTP specification, but only after
   the client has attempted at least one normal request and determined
   from the response status code or header fields (e.g., Server) that
   the server improperly handles higher request versions.

   A server SHOULD send a response version equal to the highest version
   to which the server is conformant that has a major version less than
   or equal to the one received in the request. A server MUST NOT send
   a version to which it is not conformant. A server can send a 505
   (HTTP Version Not Supported) response if it wishes, for any reason,
   to refuse service of the client's major protocol version.

   A server MAY send an HTTP/1.0 response to a 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.

   The intention of HTTP's versioning design is that the major number
   will only be incremented if an incompatible message syntax is
   introduced, and that the minor number will only be incremented when
   changes made to the protocol have the effect of adding to the message
   semantics or implying additional capabilities of the sender.
   However, the minor version was not incremented for the changes
   introduced between [RFC2068] and [RFC2616], and this revision has
   specifically avoided any such changes to the protocol.

   When an HTTP message is received with a major version number that the
   recipient implements, but a higher minor version number than what the
   recipient implements, the recipient SHOULD process the message as if
   it were in the highest minor version within that major version to
   which the recipient is conformant. A recipient can assume that a

Fielding & Reschke   Standards Track   [Page 15]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

message with a higher minor version, when sent to a recipient that
has not yet indicated support for that higher version, is
sufficiently backwards-compatible to be safely processed by any
implementation of the same major version.

2.7. Uniform Resource Identifiers

Uniform Resource Identifiers (URIs) [RFC3986] are used throughout
HTTP as the means for identifying resources (Section 2 of [RFC7231]).
URI references are used to target requests, indicate redirects, and
define relationships.

The definitions of "URI-reference", "absolute-URI", "relative-part",
"scheme", "authority", "port", "host", "path-abempty", "segment",
"query", and "fragment" are adopted from the URI generic syntax. An
"absolute-path" rule is defined for protocol elements that can
contain a non-empty path component. (This rule differs slightly from
the path-abempty rule of RFC 3986, which allows for an empty path to
be used in references, and path-absolute rule, which does not allow
paths that begin with "//".) A "partial-URI" rule is defined for
protocol elements that can contain a relative URI but not a fragment
component.

```
  URI-reference = <URI-reference, see [RFC3986], Section 4.1>
  absolute-URI  = <absolute-URI, see [RFC3986], Section 4.3>
  relative-part = <relative-part, see [RFC3986], Section 4.2>
  scheme        = <scheme, see [RFC3986], Section 3.1>
  authority     = <authority, see [RFC3986], Section 3.2>
  uri-host      = <host, see [RFC3986], Section 3.2.2>
  port          = <port, see [RFC3986], Section 3.2.3>
  path-abempty  = <path-abempty, see [RFC3986], Section 3.3>
  segment       = <segment, see [RFC3986], Section 3.3>
  query         = <query, see [RFC3986], Section 3.4>
  fragment      = <fragment, see [RFC3986], Section 3.5>

  absolute-path = 1*( "/" segment )
  partial-URI   = relative-part [ "?" query ]
```

Each protocol element in HTTP that allows a URI reference will
indicate in its ABNF production whether the element allows any form
of reference (URI-reference), only a URI in absolute form
(absolute-URI), only the path and optional query components, or some
combination of the above. Unless otherwise indicated, URI references
are parsed relative to the effective request URI (Section 5.5).

Fielding & Reschke   Standards Track   [Page 16]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

2.7.1. http URI Scheme

   The "http" URI scheme is hereby defined for the purpose of minting
   identifiers according to their association with the hierarchical
   namespace governed by a potential HTTP origin server listening for
   TCP ([RFC0793]) connections on a given port.

     http-URI = "http:" "//" authority path-abempty ["?" query]
                ["#" fragment]

   The origin server for an "http" URI is identified by the authority
   component, which includes a host identifier and optional TCP port
   ([RFC3986], Section 3.2.2). The hierarchical path component and
   optional query component serve as an identifier for a potential
   target resource within that origin server's name space. The optional
   fragment component allows for indirect identification of a secondary
   resource, independent of the URI scheme, as defined in Section 3.5 of
   [RFC3986].

   A sender MUST NOT generate an "http" URI with an empty host
   identifier. A recipient that processes such a URI reference MUST
   reject it as invalid.

   If the host identifier is provided as an IP address, the origin
   server is the listener (if any) on the indicated TCP port at that IP
   address. If host is a registered name, the registered name is an
   indirect identifier for use with a name resolution service, such as
   DNS, to find an address for that origin server. If the port
   subcomponent is empty or not given, TCP port 80 (the reserved port
   for WWW services) is the default.

   Note that the presence of a URI with a given authority component does
   not imply that there is always an HTTP server listening for
   connections on that host and port. Anyone can mint a URI. What the
   authority component determines is who has the right to respond
   authoritatively to requests that target the identified resource. The
   delegated nature of registered names and IP addresses creates a
   federated namespace, based on control over the indicated host and
   port, whether or not an HTTP server is present. See Section 9.1 for
   security considerations related to establishing authority.

   When an "http" URI is used within a context that calls for access to
   the indicated resource, a client MAY attempt access by resolving the
   host to an IP address, establishing a TCP connection to that address
   on the indicated port, and sending an HTTP request message
   (Section 3) containing the URI's identifying data (Section 5) to the
   server. If the server responds to that request with a non-interim

Fielding & Reschke   Standards Track   [Page 17]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   HTTP response message, as described in Section 6 of [RFC7231], then
   that response is considered an authoritative answer to the client's
   request.

   Although HTTP is independent of the transport protocol, the "http"
   scheme is specific to TCP-based services because the name delegation
   process depends on TCP for establishing authority. An HTTP service
   based on some other underlying connection protocol would presumably
   be identified using a different URI scheme, just as the "https"
   scheme (below) is used for resources that require an end-to-end
   secured connection. Other protocols might also be used to provide
   access to "http" identified resources -- it is only the authoritative
   interface that is specific to TCP.

   The URI generic syntax for authority also includes a deprecated
   userinfo subcomponent ([RFC3986], Section 3.2.1) for including user
   authentication information in the URI. Some implementations make use
   of the userinfo component for internal configuration of
   authentication information, such as within command invocation
   options, configuration files, or bookmark lists, even though such
   usage might expose a user identifier or password. A sender MUST NOT
   generate the userinfo subcomponent (and its "@" delimiter) when an
   "http" URI reference is generated within a message as a request
   target or header field value. Before making use of an "http" URI
   reference received from an untrusted source, a recipient SHOULD parse
   for userinfo and treat its presence as an error; it is likely being
   used to obscure the authority for the sake of phishing attacks.

2.7.2. https URI Scheme

   The "https" URI scheme is hereby defined for the purpose of minting
   identifiers according to their association with the hierarchical
   namespace governed by a potential HTTP origin server listening to a
   given TCP port for TLS-secured connections ([RFC5246]).

   All of the requirements listed above for the "http" scheme are also
   requirements for the "https" scheme, except that TCP port 443 is the
   default if the port subcomponent is empty or not given, and the user
   agent MUST ensure that its connection to the origin server is secured
   through the use of strong encryption, end-to-end, prior to sending
   the first HTTP request.

     https-URI = "https:" "//" authority path-abempty ["?" query]
                 ["#" fragment]

   Note that the "https" URI scheme depends on both TLS and TCP for
   establishing authority. Resources made available via the "https"
   scheme have no shared identity with the "http" scheme even if their

Fielding & Reschke   Standards Track   [Page 18]

   resource identifiers indicate the same authority (the same host
   listening to the same TCP port). They are distinct namespaces and
   are considered to be distinct origin servers. However, an extension
   to HTTP that is defined to apply to entire host domains, such as the
   Cookie protocol [RFC6265], can allow information set by one service
   to impact communication with other services within a matching group
   of host domains.

   The process for authoritative access to an "https" identified
   resource is defined in [RFC2818].

2.7.3. http and https URI Normalization and Comparison

   Since the "http" and "https" schemes conform to the URI generic
   syntax, such URIs are normalized and compared according to the
   algorithm defined in Section 6 of [RFC3986], using the defaults
   described above for each scheme.

   If the port is equal to the default port for a scheme, the normal
   form is to omit the port subcomponent. When not being used in
   absolute form as the request target of an OPTIONS request, an empty
   path component is equivalent to an absolute path of "/", so the
   normal form is to provide a path of "/" instead. The scheme and host
   are case-insensitive and normally provided in lowercase; all other
   components are compared in a case-sensitive manner. Characters other
   than those in the "reserved" set are equivalent to their
   percent-encoded octets: the normal form is to not encode them (see
   Sections 2.1 and 2.2 of [RFC3986]).

   For example, the following three URIs are equivalent:

      http://example.com:80/~smith/home.html
      http://EXAMPLE.com/%7Esmith/home.html
      http://EXAMPLE.com:/%7esmith/home.html

3. Message Format

   All HTTP/1.1 messages consist of a start-line followed by 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
                       *(header-field CRLF)
                       CRLF
                       [message-body]

```
RFC 7230          HTTP/1.1 Message Syntax and Routing          June 2014
```

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.

An HTTP message can be parsed as a stream for incremental processing
or forwarding downstream. However, recipients cannot rely on
incremental delivery of partial messages, since some implementations
will buffer or delay message forwarding for the sake of network
efficiency, security checks, or payload transformations.

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.

3.1. Start Line

An HTTP 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 3.3).

```
Fielding & Reschke           Standards Track                  [Page 20]
```

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   In theory, a client could receive requests and a server could receive
   responses, distinguishing them by their different start-line formats,
   but, 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.

      start-line   = request-line / status-line

3.1.1. Request Line

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

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

   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 4 of [RFC7231], along with information regarding the HTTP
   method registry and considerations for defining new methods.

   The request-target identifies the target resource upon which to apply
   the request, as defined in Section 5.3.

   Recipients typically parse the request-line into its component parts
   by splitting on whitespace (see Section 3.5), since no whitespace is
   allowed in the three components. Unfortunately, some user agents
   fail to properly encode or exclude whitespace found in hypertext
   references, resulting in those disallowed characters being sent in a
   request-target.

   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.

   HTTP does not place a predefined limit on the length of a
   request-line, as described in Section 2.5. 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

Fielding & Reschke   Standards Track   [Page 21]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   request-target longer than any URI it wishes to parse MUST respond
   with a 414 (URI Too Long) status code (see Section 6.5.12 of
   [RFC7231]).

   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.

3.1.2. 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, a possibly empty textual phrase describing the status code,
   and ending with CRLF.

     status-line = HTTP-version SP status-code SP reason-phrase CRLF

   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 6 of [RFC7231] 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. A client SHOULD
   ignore the reason-phrase content.

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

3.2. Header Fields

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

Fielding & Reschke   Standards Track   [Page 22]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

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

      field-name     = token
      field-value    = *( field-content / obs-fold )
      field-content  = field-vchar [ 1*( SP / HTAB ) field-vchar ]
      field-vchar    = VCHAR / obs-text

      obs-fold       = CRLF 1*( SP / HTAB )
                     ; obsolete line folding
                     ; see Section 3.2.4
```

   The field-name token labels the corresponding field-value as having
   the semantics defined by that header field. For example, the Date
   header field is defined in Section 7.1.1.2 of [RFC7231] as containing
   the origination timestamp for the message in which it appears.

3.2.1. Field Extensibility

   Header fields are fully extensible: there is no limit on the
   introduction of new field names, each presumably defining new
   semantics, nor on the number of header fields used in a given
   message. Existing fields are defined in each part of this
   specification and in many other specifications outside this document
   set.

   New header fields can be defined such that, when they are understood
   by a recipient, they might override or enhance the interpretation of
   previously defined header fields, define preconditions on request
   evaluation, or refine the meaning of responses.

   A proxy MUST forward unrecognized header fields unless the field-name
   is listed in the Connection header field (Section 6.1) or the proxy
   is specifically configured to block, or otherwise transform, such
   fields. Other recipients SHOULD ignore unrecognized header fields.
   These requirements allow HTTP's functionality to be enhanced without
   requiring prior update of deployed intermediaries.

   All defined header fields ought to be registered with IANA in the
   "Message Headers" registry, as described in Section 8.3 of [RFC7231].

3.2.2. Field Order

   The order in which header fields with differing field names are
   received is not significant. However, it is good practice to send
   header fields that contain control data first, such as Host on
   requests and Date on responses, so that implementations can decide
   when not to handle a message as early as possible. A server MUST NOT
   apply a request to the target resource until the entire request

Fielding & Reschke   Standards Track   [Page 23]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

header section is received, since later header fields might include
conditionals, authentication credentials, or deliberately misleading
duplicate header fields that would impact request processing.

A sender MUST NOT generate multiple header fields with the same field
name in a message unless either the entire field value for that
header field is defined as a comma-separated list [i.e., #(values)]
or the header field is a well-known exception (as noted below).

A recipient MAY combine multiple header fields with the same field
name into one "field-name: field-value" pair, without changing the
semantics of the message, by appending each subsequent field value to
the combined field value in order, separated by a comma. The order
in which header fields with the same field name are received is
therefore significant to the interpretation of the combined field
value; a proxy MUST NOT change the order of these field values when
forwarding a message.

   Note: In practice, the "Set-Cookie" header field ([RFC6265]) often
   appears multiple times in a response message and does not use the
   list syntax, violating the above requirements on multiple header
   fields with the same name. Since it cannot be combined into a
   single field-value, recipients ought to handle "Set-Cookie" as a
   special case while processing header fields. (See Appendix A.2.3
   of [Kri2001] for details.)

3.2.3. Whitespace

This specification uses three rules to denote the use of linear
whitespace: OWS (optional whitespace), RWS (required whitespace), and
BWS ("bad" whitespace).

The OWS rule is used where zero or more linear whitespace octets
might appear. For protocol elements where optional whitespace is
preferred to improve readability, a sender SHOULD generate the
optional whitespace as a single SP; otherwise, a sender SHOULD NOT
generate optional whitespace except as needed to white out invalid or
unwanted protocol elements during in-place message filtering.

The RWS rule is used when at least one linear whitespace octet is
required to separate field tokens. A sender SHOULD generate RWS as a
single SP.

The BWS rule is used where the grammar allows optional whitespace
only for historical reasons. A sender MUST NOT generate BWS in
messages. A recipient MUST parse for such bad whitespace and remove
it before interpreting the protocol element.

Fielding & Reschke   Standards Track   [Page 24]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

```
   OWS               = *( SP / HTAB )
                     ; optional whitespace
   RWS               = 1*( SP / HTAB )
                     ; required whitespace
   BWS               = OWS
                     ; "bad" whitespace
```

3.2.4. 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). Consequently, this specification does not use ABNF rules
   to define each "Field-Name: Field Value" pair, as was done in
   previous editions. Instead, this specification uses ABNF rules that
   are named according to each registered field name, wherein the rule
   defines the valid grammar for that field's corresponding field values
   (i.e., after the field-value has been extracted from the header
   section by a generic field parser).

   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 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.

   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 8.3.1). 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.

Fielding & Reschke   Standards Track   [Page 25]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   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.

   Historically, HTTP has allowed field content with text in the
   ISO-8859-1 charset [ISO-8859-1], supporting other charsets only
   through use of [RFC2047] encoding. In practice, most HTTP header
   field values use only a subset of the US-ASCII charset [USASCII].
   Newly defined header fields SHOULD limit their field values to
   US-ASCII octets. A recipient SHOULD treat other octets in field
   content (obs-text) as opaque data.

3.2.5. Field Limits

   HTTP does not place a predefined limit on the length of each header
   field or on the length of the header section as a whole, as described
   in Section 2.5. Various ad hoc limitations on individual header
   field length are found in practice, often depending on the specific
   field semantics.

   A server that receives a request header field, or set of fields,
   larger than it wishes to process MUST respond with an appropriate 4xx
   (Client Error) status code. Ignoring such header fields would
   increase the server's vulnerability to request smuggling attacks
   (Section 9.5).

   A client MAY discard or truncate received header fields that are
   larger than the client wishes to process if the field semantics are
   such that the dropped value(s) can be safely ignored without changing
   the message framing or response semantics.

Fielding & Reschke   Standards Track   [Page 26]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

3.2.6. Field Value Components

   Most HTTP header field values are defined using common syntax
   components (token, quoted-string, and comment) separated by
   whitespace or specific delimiting characters. Delimiters are chosen
   from the set of US-ASCII visual characters not allowed in a token
   (DQUOTE and "(),/:;<=>?@[\]{}").

```
      token          = 1*tchar

      tchar          = "!" / "#" / "$" / "%" / "&" / "'" / "*"
                     / "+" / "-" / "." / "^" / "_" / "`" / "|" / "~"
                     / DIGIT / ALPHA
                     ; any VCHAR, except delimiters
```

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

```
      quoted-string  = DQUOTE *( qdtext / quoted-pair ) DQUOTE
      qdtext         = HTAB / SP /%x21 / %x23-5B / %x5D-7E / obs-text
      obs-text       = %x80-FF
```

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

```
      comment        = "(" *( ctext / quoted-pair / comment ) ")"
      ctext          = HTAB / SP / %x21-27 / %x2A-5B / %x5D-7E / obs-text
```

   The backslash octet ("\") can be used as a single-octet quoting
   mechanism within quoted-string and comment constructs. Recipients
   that process the value of a quoted-string MUST handle a quoted-pair
   as if it were replaced by the octet following the backslash.

```
      quoted-pair    = "\" ( HTAB / SP / VCHAR / obs-text )
```

   A sender SHOULD NOT generate a quoted-pair in a quoted-string except
   where necessary to quote DQUOTE and backslash octets occurring within
   that string. A sender SHOULD NOT generate a quoted-pair in a comment
   except where necessary to quote parentheses ["(" and ")"] and
   backslash octets occurring within that comment.

Fielding & Reschke   Standards Track   [Page 27]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

3.3. Message Body

The message body (if any) of an HTTP message is used to carry the
payload body 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 3.3.1.

      message-body = *OCTET

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

The presence of a message body in a request is signaled by 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 3.1.2). Responses to the HEAD request method (Section 4.3.2
of [RFC7231]) never include a message body because the associated
response header fields (e.g., Transfer-Encoding, Content-Length,
etc.), if present, indicate only what their values would have been if
the request method had been GET (Section 4.3.1 of [RFC7231]). 2xx
(Successful) responses to a CONNECT request method (Section 4.3.6 of
[RFC7231]) switch to tunnel mode instead of having a message body.
All 1xx (Informational), 204 (No Content), and 304 (Not Modified)
responses do not include a message body. All other responses do
include a message body, although the body might be of zero length.

3.3.1. 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 4.

      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.

Fielding & Reschke   Standards Track   [Page 28]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   A recipient MUST be able to parse the chunked transfer coding
   (Section 4.1) 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 3.1.2.1 of [RFC7231]),
   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 4.1 of [RFC7232]) 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 4.3.6 of
   [RFC7231]).

   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

Fielding & Reschke   Standards Track   [Page 29]

```
RFC 7230          HTTP/1.1 Message Syntax and Routing        June 2014
```

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).

3.3.2. 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 3 of [RFC7231]).

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

An example is

```
  Content-Length: 3495
```

A sender MUST NOT send a Content-Length header field in any message
that contains a Transfer-Encoding header field.

A user agent SHOULD send a Content-Length in a request message when
no Transfer-Encoding is sent and the request method defines a meaning
for an enclosed payload body. For example, a Content-Length header
field is normally sent in a POST request even when the value is 0
(indicating an empty payload body). A user agent SHOULD NOT send a
Content-Length header field when the request message does not contain
a payload body and the method semantics do not anticipate such a
body.

A server MAY send a Content-Length header field in a response to a
HEAD request (Section 4.3.2 of [RFC7231]); a server MUST NOT send
Content-Length in such a response unless its field-value equals the
decimal number of octets that would have been sent in the payload
body of a response if the same request had used the GET method.

A server MAY send a Content-Length header field in a 304 (Not
Modified) response to a conditional GET request (Section 4.1 of
[RFC7232]); a server MUST NOT send Content-Length in such a response

```
Fielding & Reschke          Standards Track                    [Page 30]
```

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   unless its field-value equals the decimal number of octets that would
   have been sent in the payload body of a 200 (OK) response to the same
   request.

   A server MUST NOT send a Content-Length header field in any response
   with a status code of 1xx (Informational) or 204 (No Content). A
   server MUST NOT send a Content-Length header field in any 2xx
   (Successful) response to a CONNECT request (Section 4.3.6 of
   [RFC7231]).

   Aside from the cases defined above, in the absence of
   Transfer-Encoding, an origin server SHOULD send a Content-Length
   header field when the payload body size is known prior to sending the
   complete header section. This will allow downstream recipients to
   measure transfer progress, know when a received message is complete,
   and potentially reuse the connection for additional requests.

   Any Content-Length field value greater than or equal to zero is
   valid. Since there is no predefined limit to the length of a
   payload, a recipient MUST anticipate potentially large decimal
   numerals and prevent parsing errors due to integer conversion
   overflows (Section 9.3).

   If a message is received that has multiple Content-Length header
   fields with field-values consisting of the same decimal value, or a
   single Content-Length header field with a field value containing a
   list of identical decimal values (e.g., "Content-Length: 42, 42"),
   indicating that duplicate Content-Length header fields have been
   generated or combined by an upstream message processor, then the
   recipient MUST either reject the message as invalid or replace the
   duplicated field-values with a single valid Content-Length field
   containing that decimal value prior to determining the message body
   length or forwarding the message.

      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.

Fielding & Reschke   Standards Track   [Page 31]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

3.3.3. 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 4.1) 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 9.5) or response splitting
       (Section 9.4) 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

Fielding & Reschke   Standards Track   [Page 32]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

      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

Fielding & Reschke   Standards Track   [Page 33]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

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.

3.4. 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 [RFC7234].

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.

3.5. Message Parsing Robustness

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.

Fielding & Reschke   Standards Track   [Page 34]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   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.

   Although the request-line and status-line grammar rules require 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 security vulnerabilities if there are multiple recipients
   of the message and each has its own unique interpretation of
   robustness (see Section 9.5).

   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.

4. 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    = "chunked" ; Section 4.1
                        / "compress" ; Section 4.2.1
                        / "deflate" ; Section 4.2.2
                        / "gzip" ; Section 4.2.3
                        / transfer-extension
     transfer-extension = token *( OWS ";" OWS transfer-parameter )
```

   Parameters are in the form of a name or 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 8.4. They are used in the TE (Section 4.3) and
   Transfer-Encoding (Section 3.3.1) header fields.

Fielding & Reschke   Standards Track   [Page 35]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

4.1. 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 containing header 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-part
                    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, and finally terminated by an empty line.

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

4.1.1. 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      = *( ";" chunk-ext-name [ "=" 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

Fielding & Reschke   Standards Track   [Page 36]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

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.

4.1.2. Chunked Trailer Part

A trailer 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 trailer fields are identical to header fields, except
they are sent in a chunked trailer instead of the message's header
section.

      trailer-part   = *(header-field CRLF)

A sender MUST NOT generate a trailer that contains a field necessary
for message framing (e.g., Transfer-Encoding and Content-Length),
routing (e.g., Host), request modifiers (e.g., controls and
conditionals in Section 5 of [RFC7231]), authentication (e.g., see
[RFC7235] and [RFC6265]), response control data (e.g., see Section
7.1 of [RFC7231]), or determining how to process the payload (e.g.,
Content-Encoding, Content-Type, Content-Range, and Trailer).

When a chunked message containing a non-empty trailer is received,
the recipient MAY process the fields (aside from those forbidden
above) as if they were appended to the message's header section. A
recipient MUST ignore (or consider as an error) any fields that are
forbidden to be sent in a trailer, since processing them as if they
were present in the header section might bypass external security
filters.

Unless the request includes a TE header field indicating "trailers"
is acceptable, as described in Section 4.3, a server SHOULD NOT
generate trailer fields that it believes are necessary for the user
agent to receive. Without a TE containing "trailers", the server
ought to assume that the trailer fields might be silently discarded
along the path to the user agent. This requirement allows
intermediaries to forward a de-chunked message to an HTTP/1.0
recipient without buffering the entire response.

Fielding & Reschke   Standards Track   [Page 37]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

4.1.3. 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 field is allowed to be sent in a trailer) {
       append trailer field to existing header fields
   }
   read trailer-field
}
Content-Length := length
Remove "chunked" from Transfer-Encoding
Remove Trailer from existing header fields
```

4.2. Compression Codings

The codings defined below can be used to compress the payload of a
message.

4.2.1. Compress Coding

The "compress" coding is an adaptive Lempel-Ziv-Welch (LZW) coding
[Welch] that is commonly produced by the UNIX file compression
program "compress". A recipient SHOULD consider "x-compress" to be
equivalent to "compress".

4.2.2. Deflate Coding

The "deflate" coding is a "zlib" data format [RFC1950] containing a
"deflate" compressed data stream [RFC1951] that uses a combination of
the Lempel-Ziv (LZ77) compression algorithm and Huffman coding.

   Note: Some non-conformant implementations send the "deflate"
   compressed data without the zlib wrapper.

Fielding & Reschke   Standards Track   [Page 38]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

4.2.3. Gzip Coding

   The "gzip" coding is an LZ77 coding with a 32-bit Cyclic Redundancy
   Check (CRC) that is commonly produced by the gzip file compression
   program [RFC1952]. A recipient SHOULD consider "x-gzip" to be
   equivalent to "gzip".

4.3. 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 4), 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 4.1.2, 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

Fielding & Reschke   Standards Track   [Page 39]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   5.3.1 of [RFC7231]). 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 6.1) in order to prevent the TE
   field from being forwarded by intermediaries that do not support its
   semantics.

4.4. Trailer

   When a message includes a message body encoded with the chunked
   transfer coding and the sender desires to send metadata in the form
   of trailer fields at the end of the message, the sender SHOULD
   generate a Trailer header field before the message body to indicate
   which fields will be present in the trailers. This allows the
   recipient to prepare for receipt of that metadata before it starts
   processing the body, which is useful if the message is being streamed
   and the recipient wishes to confirm an integrity check on the fly.

     Trailer = 1#field-name

5. Message Routing

   HTTP request message routing is determined by each client based on
   the target resource, the client's proxy configuration, and
   establishment or reuse of an inbound connection. The corresponding
   response routing follows the same connection chain back to the
   client.

5.1. Identifying a Target Resource

   HTTP is used in a wide variety of applications, ranging from
   general-purpose computers to home appliances. In some cases,
   communication options are hard-coded in a client's configuration.
   However, most HTTP clients rely on the same resource identification
   mechanism and configuration techniques as general-purpose Web
   browsers.

   HTTP communication is initiated by a user agent for some purpose.
   The purpose is a combination of request semantics, which are defined
   in [RFC7231], and a target resource upon which to apply those
   semantics. A URI reference (Section 2.7) is typically used as an

Fielding & Reschke   Standards Track   [Page 40]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   identifier for the "target resource", which a user agent would
   resolve to its absolute form in order to obtain the "target URI".
   The target URI excludes the reference's fragment component, if any,
   since fragment identifiers are reserved for client-side processing
   ([RFC3986], Section 3.5).

5.2. Connecting Inbound

   Once the target URI is determined, a client needs to decide whether a
   network request is necessary to accomplish the desired semantics and,
   if so, where that request is to be directed.

   If the client has a cache [RFC7234] and the request can be satisfied
   by it, then the request is usually directed there first.

   If the request is not satisfied by a cache, then a typical client
   will check its configuration to determine whether a proxy is to be
   used to satisfy the request. Proxy configuration is implementation-
   dependent, but is often based on URI prefix matching, selective
   authority matching, or both, and the proxy itself is usually
   identified by an "http" or "https" URI. If a proxy is applicable,
   the client connects inbound by establishing (or reusing) a connection
   to that proxy.

   If no proxy is applicable, a typical client will invoke a handler
   routine, usually specific to the target URI's scheme, to connect
   directly to an authority for the target resource. How that is
   accomplished is dependent on the target URI scheme and defined by its
   associated specification, similar to how this specification defines
   origin server access for resolution of the "http" (Section 2.7.1) and
   "https" (Section 2.7.2) schemes.

   HTTP requirements regarding connection management are defined in
   Section 6.

5.3. Request Target

   Once an inbound connection is obtained, the client sends an HTTP
   request message (Section 3) with a request-target derived from the
   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

Fielding & Reschke   Standards Track   [Page 41]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

5.3.1. 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.

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.

5.3.2. 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 5.7.

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

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

Fielding & Reschke   Standards Track   [Page 42]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

    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.

5.3.3. authority-form

    The authority-form of request-target is only used for CONNECT
    requests (Section 4.3.6 of [RFC7231]).

       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

5.3.4. asterisk-form

    The asterisk-form of request-target is only used for a server-wide
    OPTIONS request (Section 4.3.7 of [RFC7231]).

       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".

Fielding & Reschke   Standards Track   [Page 43]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

5.4. Host

   The "Host" header field in a request provides the host and port
   information from the target URI, enabling the origin server to
   distinguish among resources while servicing requests for multiple
   host names on a single IP address.

     Host = uri-host [":" port] ; Section 2.7.1

   A client MUST send a Host header field in all HTTP/1.1 request
   messages. If the target URI includes an authority component, then a
   client MUST send a field-value for Host that is identical to that
   authority component, excluding any userinfo subcomponent and its "@"
   delimiter (Section 2.7.1). If the authority component is missing or
   undefined for the target URI, then a client MUST send a Host header
   field with an empty field-value.

   Since the Host field-value is critical information for handling a
   request, a user agent SHOULD generate Host as the first header field
   following the request-line.

   For example, a GET request to the origin server for
   <http://www.example.org/pub/WWW/> would begin with:

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

   A client MUST send a Host header field in an HTTP/1.1 request even if
   the request-target is in the absolute-form, since this allows the
   Host information to be forwarded through ancient HTTP/1.0 proxies
   that might not have implemented Host.

   When a proxy receives a request with an absolute-form of
   request-target, the proxy MUST ignore the received Host header field
   (if any) and instead replace it with the host information of the
   request-target. A proxy that forwards such a request MUST generate a
   new Host field-value based on the received request-target rather than
   forward the received Host field-value.

   Since the Host header field acts as an application-level routing
   mechanism, it is a frequent target for malware seeking to poison a
   shared cache or redirect a request to an unintended server. An
   interception proxy is particularly vulnerable if it relies on the
   Host field-value for redirecting requests to internal servers, or for
   use as a cache key in a shared cache, without first verifying that
   the intercepted connection is targeting a valid IP address for that
   host.

Fielding & Reschke   Standards Track   [Page 44]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   A server MUST respond with a 400 (Bad Request) status code to any
   HTTP/1.1 request message that lacks a Host header field and to any
   request message that contains more than one Host header field or a
   Host header field with an invalid field-value.

5.5. 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. This
   reconstruction involves both the server's local configuration and
   information communicated in the request-target, Host header field,
   and connection context.

   For a user agent, the effective request URI is the target URI.

   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.

Fielding & Reschke   Standards Track   [Page 45]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   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.

   Once the effective request URI has been constructed, an origin server
   needs to decide whether or not to provide service for that URI via
   the connection in which the request was received. For example, the
   request might have been misdirected, deliberately or accidentally,
   such that the information within a received request-target or Host
   header field differs from the host or port upon which the connection
   has been made. If the connection is from a trusted gateway, that
   inconsistency might be expected; otherwise, it might indicate an
   attempt to bypass security filters, trick the server into delivering
   non-public content, or poison a cache. See Section 9 for security
   considerations regarding message routing.

5.6. Associating a Response to a Request

   HTTP 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 6.2 of
   [RFC7231]) precede a final response to the same request.

Fielding & Reschke   Standards Track   [Page 46]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   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.

5.7. Message Forwarding

   As described in Section 2.3, intermediaries can serve a variety of
   roles in the processing of HTTP requests and responses. Some
   intermediaries are used to improve performance or availability.
   Others are used for access control or to filter content. Since an
   HTTP stream has characteristics similar to a pipe-and-filter
   architecture, there are no inherent limits to the extent an
   intermediary can enhance (or interfere) with either direction of the
   stream.

   An intermediary not acting as a tunnel MUST implement the Connection
   header field, as specified in Section 6.1, and exclude fields from
   being forwarded that are only intended for the incoming connection.

   An intermediary MUST NOT forward a message to itself unless it is
   protected from an infinite request loop. In general, an intermediary
   ought to recognize its own server names, including any aliases, local
   variations, or literal IP addresses, and respond to such requests
   directly.

5.7.1. Via

   The "Via" header field indicates the presence of intermediate
   protocols and recipients between the user agent and the server (on
   requests) or between the origin server and the client (on responses),
   similar to the "Received" header field in email (Section 3.6.7 of
   [RFC5322]). Via can be used for tracking message forwards, avoiding
   request loops, and identifying the protocol capabilities of senders
   along the request/response chain.

     Via = 1#(received-protocol RWS received-by [RWS comment])

     received-protocol = [protocol-name "/"] protocol-version
                       ; see Section 6.7
     received-by   = (uri-host [":" port]) / pseudonym
     pseudonym   = token

   Multiple Via field values represent each proxy or gateway that has
   forwarded the message. Each intermediary appends its own information
   about how the message was received, such that the end result is
   ordered according to the sequence of forwarding recipients.

Fielding & Reschke   Standards Track   [Page 47]

A proxy MUST send an appropriate Via header field, as described
below, in each message that it forwards. An HTTP-to-HTTP gateway
MUST send an appropriate Via header field in each inbound request
message and MAY send a Via header field in forwarded response
messages.

For each intermediary, the received-protocol indicates the protocol
and protocol version used by the upstream sender of the message.
Hence, the Via field value records the advertised protocol
capabilities of the request/response chain such that they remain
visible to downstream recipients; this can be useful for determining
what backwards-incompatible features might be safe to use in
response, or within a later request, as described in Section 2.6.
For brevity, the protocol-name is omitted when the received protocol
is HTTP.

The received-by portion of the field value is normally the host and
optional port number of a recipient server or client that
subsequently forwarded the message. However, if the real host is
considered to be sensitive information, a sender MAY replace it with
a pseudonym. If a port is not provided, a recipient MAY interpret
that as meaning it was received on the default TCP port, if any, for
the received-protocol.

A sender MAY generate comments in the Via header field to identify
the software of each recipient, analogous to the User-Agent and
Server header fields. However, all comments in the Via field are
optional, and a recipient MAY remove them prior to forwarding the
message.

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

  Via: 1.0 fred, 1.1 p.example.net

An intermediary used as a portal through a network firewall SHOULD
NOT forward the names and ports of hosts within the firewall region
unless it is explicitly enabled to do so. If not enabled, such an
intermediary SHOULD replace each received-by host of any host behind
the firewall by an appropriate pseudonym for that host.

Fielding & Reschke   Standards Track   [Page 48]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   An intermediary MAY combine an ordered subsequence of Via header
   field entries into a single such entry if the entries have identical
   received-protocol values. For example,

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

   could be collapsed to

     Via: 1.0 ricky, 1.1 mertz, 1.0 lucy

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

5.7.2. Transformations

   Some intermediaries include features for transforming messages and
   their payloads. A proxy might, for example, convert between image
   formats in order to save cache space or to reduce the amount of
   traffic on a slow link. However, operational problems might occur
   when these transformations are applied to payloads intended for
   critical applications, such as medical imaging or scientific data
   analysis, particularly when integrity checks or digital signatures
   are used to ensure that the payload received is identical to the
   original.

   An HTTP-to-HTTP proxy is called a "transforming proxy" if it is
   designed or configured to modify messages in a semantically
   meaningful way (i.e., modifications, beyond those required by normal
   HTTP processing, that change the message in a way that would be
   significant to the original sender or potentially significant to
   downstream recipients). For example, a transforming proxy might be
   acting as a shared annotation server (modifying responses to include
   references to a local annotation database), a malware filter, a
   format transcoder, or a privacy filter. Such transformations are
   presumed to be desired by whichever client (or client organization)
   selected the proxy.

   If a proxy receives a request-target with a host name that is not a
   fully qualified domain name, it MAY add its own domain to the host
   name it received when forwarding the request. A proxy MUST NOT
   change the host name if the request-target contains a fully qualified
   domain name.

Fielding & Reschke   Standards Track   [Page 49]

A proxy MUST NOT modify the "absolute-path" and "query" parts of the
received request-target when forwarding it to the next inbound
server, except as noted above to replace an empty path with "/" or
"*".

A proxy MAY modify the message body through application or removal of
a transfer coding (Section 4).

A proxy MUST NOT transform the payload (Section 3.3 of [RFC7231]) of
a message that contains a no-transform cache-control directive
(Section 5.2 of [RFC7234]).

A proxy MAY transform the payload of a message that does not contain
a no-transform cache-control directive. A proxy that transforms a
payload MUST add a Warning header field with the warn-code of 214
("Transformation Applied") if one is not already in the message (see
Section 5.5 of [RFC7234]). A proxy that transforms the payload of a
200 (OK) response can further inform downstream recipients that a
transformation has been applied by changing the response status code
to 203 (Non-Authoritative Information) (Section 6.3.4 of [RFC7231]).

A proxy SHOULD NOT modify header fields that provide information
about the endpoints of the communication chain, the resource state,
or the selected representation (other than the payload) unless the
field's definition specifically allows such modification or the
modification is deemed necessary for privacy or security.

6. 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, 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 2.7.1) 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.

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   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.

6.1. 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 [RFC7234]).

Fielding & Reschke   Standards Track   [Page 51]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   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 6.6).

   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.

6.2. 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.

6.3. 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.

Fielding & Reschke   Standards Track   [Page 52]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   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, the recipient is not a proxy, 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 3.3. 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 A.1.2 for more information on backwards compatibility
   with HTTP/1.0 clients.

6.3.1. 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.

Fielding & Reschke   Standards Track   [Page 53]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

When an inbound connection is closed prematurely, a client MAY open a
new connection and automatically retransmit an aborted sequence of
requests if all of those requests have idempotent methods (Section
4.2.2 of [RFC7231]). A proxy MUST NOT automatically retry
non-idempotent requests.

A user agent MUST NOT automatically retry a request with a non-
idempotent method unless it has some means to know that the request
semantics are actually idempotent, regardless of the method, or some
means to detect that the original request was never applied. For
example, a user agent that knows (through design or configuration)
that a POST request to a given resource is safe can repeat that
request automatically. Likewise, a user agent designed specifically
to operate on a version control repository might be able to recover
from partial failure conditions by checking the target resource
revision(s) after a failed connection, reverting or fixing any
changes that were partially applied, and then automatically retrying
the requests that failed.

A client SHOULD NOT automatically retry a failed automatic retry.

6.3.2. 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 4.2.1 of [RFC7231]),
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 6.6).

Idempotent methods (Section 4.2.2 of [RFC7231]) 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.

Fielding & Reschke   Standards Track   [Page 54]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

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.

6.4. 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.

6.5. 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.

Fielding & Reschke   Standards Track   [Page 55]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   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.

6.6. Tear-down

   The Connection header field (Section 6.1) 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.

Fielding & Reschke   Standards Track   [Page 56]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

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.

6.7. 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 protocols in the Upgrade
header field of a request to invite the server to switch to one or
more of those 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

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

Fielding & Reschke   Standards Track   [Page 57]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

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.txt HTTP/1.1
    Host: www.example.com
    Connection: upgrade
    Upgrade: HTTP/2.0, SHTTP/1.3, 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.

Fielding & Reschke   Standards Track   [Page 58]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

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

     HTTP/1.1 101 Switching Protocols
     Connection: upgrade
     Upgrade: HTTP/2.0

     [... data stream switches to HTTP/2.0 with an appropriate response
     (as defined by new protocol) to the "GET /hello.txt" request ...]

   When Upgrade is sent, the sender MUST also send a Connection header
   field (Section 6.1) 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 5.1.1 of [RFC7231]), 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 6.4 of [RFC7231]).

   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 2.6 and future updates to this
   specification. Additional tokens ought to be registered with IANA
   using the registration procedure defined in Section 8.6.

7. ABNF List Extension: #rule

   A #rule extension to the ABNF rules of [RFC5234] is used to improve
   readability in the definitions of some header field values.

   A construct "#" is defined, similar to "*", for defining
   comma-delimited lists of elements. The full form is "<n>#<m>element"
   indicating at least <n> and at most <m> elements, each separated by a
   single comma (",") and optional whitespace (OWS).

```
RFC 7230           HTTP/1.1 Message Syntax and Routing        June 2014
```

In any production that uses the list construct, a sender MUST NOT
generate empty list elements. In other words, a sender MUST generate
lists that satisfy the following syntax:

```
  1#element => element *( OWS "," OWS element )
```

and:

```
  #element => [ 1#element ]
```

and for n >= 1 and m > 1:

```
  <n>#<m>element => element <n-1>*<m-1>( OWS "," OWS element )
```

For compatibility with legacy list rules, a recipient MUST parse and
ignore a reasonable number of empty list elements: enough to handle
common mistakes by senders that merge values, but not so much that
they could be used as a denial-of-service mechanism. In other words,
a recipient MUST accept lists that satisfy the following syntax:

```
  #element => [ ( "," / element ) *( OWS "," [ OWS element ] ) ]

  1#element => *( "," OWS ) element *( OWS "," [ OWS element ] )
```

Empty elements do not contribute to the count of elements present.
For example, given these ABNF productions:

```
  example-list      = 1#example-list-elmt
  example-list-elmt = token ; see Section 3.2.6
```

Then the following are valid values for example-list (not including
the double quotes, which are present for delimitation only):

```
  "foo,bar"
  "foo ,bar,"
  "foo , ,bar,charlie   "
```

In contrast, the following values would be invalid, since at least
one non-empty element is required by the example-list production:

```
  ""
  ","
  ",   ,"
```

Appendix B shows the collected ABNF for recipients after the list
constructs have been expanded.

```
Fielding & Reschke           Standards Track                   [Page 60]
```

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

8. IANA Considerations

8.1. Header Field Registration

HTTP header fields are registered within the "Message Headers"
registry maintained at
<http://www.iana.org/assignments/message-headers/>.

This document defines the following HTTP header fields, so the
"Permanent Message Header Field Names" registry has been updated
accordingly (see [BCP90]).

```
+-------------------+----------+----------+---------------+
| Header Field Name | Protocol | Status   | Reference     |
+-------------------+----------+----------+---------------+
| Connection        | http     | standard | Section 6.1   |
| Content-Length    | http     | standard | Section 3.3.2 |
| Host              | http     | standard | Section 5.4   |
| TE                | http     | standard | Section 4.3   |
| Trailer           | http     | standard | Section 4.4   |
| Transfer-Encoding | http     | standard | Section 3.3.1 |
| Upgrade           | http     | standard | Section 6.7   |
| Via               | http     | standard | Section 5.7.1 |
+-------------------+----------+----------+---------------+
```

Furthermore, the header field-name "Close" has been registered as
"reserved", since using that name as an HTTP header field might
conflict with the "close" connection option of the Connection header
field (Section 6.1).

```
+-------------------+----------+----------+-------------+
| Header Field Name | Protocol | Status   | Reference   |
+-------------------+----------+----------+-------------+
| Close             | http     | reserved | Section 8.1 |
+-------------------+----------+----------+-------------+
```

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

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

8.2. URI Scheme Registration

   IANA maintains the registry of URI Schemes [BCP115] at
   <http://www.iana.org/assignments/uri-schemes/>.

   This document defines the following URI schemes, so the "Permanent
   URI Schemes" registry has been updated accordingly.

```
   +------------+-------------------------------+---------------+
   | URI Scheme | Description                   | Reference     |
   +------------+-------------------------------+---------------+
   | http       | Hypertext Transfer Protocol   | Section 2.7.1 |
   | https      | Hypertext Transfer Protocol Secure | Section 2.7.2 |
   +------------+-------------------------------+---------------+
```

8.3. Internet Media Type Registration

   IANA maintains the registry of Internet media types [BCP13] at
   <http://www.iana.org/assignments/media-types>.

   This document serves as the specification for the Internet media
   types "message/http" and "application/http". The following has been
   registered with IANA.

8.3.1. Internet Media Type message/http

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

   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

Fielding & Reschke   Standards Track   [Page 62]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   Security considerations: see Section 9

   Interoperability considerations: N/A

   Published specification: This specification (see Section 8.3.1).

   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

8.3.2. Internet Media Type application/http

   The application/http 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.

Fielding & Reschke   Standards Track   [Page 63]

       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 9

   Interoperability considerations: N/A

   Published specification: This specification (see Section 8.3.2).

   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

8.4. Transfer Coding Registry

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

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

8.4.1. Procedure

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 3.1.2.1 of [RFC7231]) unless the encoding
transformation is identical, as is the case for the compression
codings defined in Section 4.2.

Values to be added to this namespace require IETF Review (see Section
4.1 of [RFC5226]), 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.

8.4.2. Registration

The "HTTP Transfer Coding Registry" has been updated with the
registrations below:

```
+------------+--------------------------------------+---------------+
| Name       | Description                          | Reference     |
+------------+--------------------------------------+---------------+
| chunked    | Transfer in a series of chunks       | Section 4.1   |
| compress   | UNIX "compress" data format [Welch]  | Section 4.2.1 |
| deflate    | "deflate" compressed data            | Section 4.2.2 |
|            | ([RFC1951]) inside the "zlib" data   |               |
|            | format ([RFC1950])                   |               |
| gzip       | GZIP file format [RFC1952]           | Section 4.2.3 |
| x-compress | Deprecated (alias for compress)      | Section 4.2.1 |
| x-gzip     | Deprecated (alias for gzip)          | Section 4.2.3 |
+------------+--------------------------------------+---------------+
```

Fielding & Reschke   Standards Track   [Page 65]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

8.5. Content Coding Registration

   IANA maintains the "HTTP Content Coding Registry" at
   <http://www.iana.org/assignments/http-parameters>.

   The "HTTP Content Coding Registry" has been updated with the
   registrations below:

```
+------------+-----------------------------------+---------------+
| Name       | Description                       | Reference     |
+------------+-----------------------------------+---------------+
| compress   | UNIX "compress" data format [Welch] | Section 4.2.1 |
| deflate    | "deflate" compressed data         | Section 4.2.2 |
|            | ([RFC1951]) inside the "zlib" data |               |
|            | format ([RFC1950])                |               |
| gzip       | GZIP file format [RFC1952]        | Section 4.2.3 |
| x-compress | Deprecated (alias for compress)   | Section 4.2.1 |
| x-gzip     | Deprecated (alias for gzip)       | Section 4.2.3 |
+------------+-----------------------------------+---------------+
```

8.6. 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
   <http://www.iana.org/assignments/http-upgrade-tokens>.

8.6.1. Procedure

   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 4.1 of [RFC5226]) and are subject to the following rules:

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

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

   3. The registration MUST name a point of contact.

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

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

Fielding & Reschke   Standards Track   [Page 66]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   6. 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.

   7. 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.

   This registration procedure for HTTP Upgrade Tokens replaces that
   previously defined in Section 7.2 of [RFC2817].

8.6.2. Upgrade Token Registration

   The "HTTP" entry in the upgrade token registry has been updated with
   the registration below:

   +-------+----------------------+----------------------+-------------+
   | Value | Description | Expected Version | Reference |
   | | | Tokens | |
   +-------+----------------------+----------------------+-------------+
   | HTTP | Hypertext Transfer | any DIGIT.DIGIT | Section 2.6 |
   | | Protocol | (e.g, "2.0") | |
   +-------+----------------------+----------------------+-------------+

   The responsible party is: "IETF (iesg@ietf.org) - Internet
   Engineering Task Force".

9. 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 [RFC7231].

9.1. Establishing Authority

   HTTP relies on the notion of an authoritative response: a response
   that has been determined by (or at the direction of) the authority
   identified within the target URI to be the most appropriate response
   for that request given the state of the target resource at the time
   of response message origination. Providing a response from a
   non-authoritative source, such as a shared cache, is often useful to
   improve performance and availability, but only to the extent that the
   source can be trusted or the distrusted response can be safely used.

   Unfortunately, establishing authority can be difficult. For example,
   phishing is an attack on the user's perception of authority, where
   that perception can be misled by presenting similar branding in

Fielding & Reschke   Standards Track   [Page 67]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

hypertext, possibly aided by userinfo obfuscating the authority
component (see Section 2.7.1). User agents can reduce the impact of
phishing attacks by enabling users to easily inspect a target URI
prior to making an action, by prominently distinguishing (or
rejecting) userinfo when present, and by not sending stored
credentials and cookies when the referring document is from an
unknown or untrusted source.

When a registered name is used in the authority component, the "http"
URI scheme (Section 2.7.1) relies on the user's local name resolution
service to determine where it can find authoritative responses. This
means that any attack on a user's network host table, cached names,
or name resolution libraries becomes an avenue for attack on
establishing authority. Likewise, the user's choice of server for
Domain Name Service (DNS), and the hierarchy of servers from which it
obtains resolution results, could impact the authenticity of address
mappings; DNS Security Extensions (DNSSEC, [RFC4033]) are one way to
improve authenticity.

Furthermore, after an IP address is obtained, establishing authority
for an "http" URI is vulnerable to attacks on Internet Protocol
routing.

The "https" scheme (Section 2.7.2) is intended to prevent (or at
least reveal) many of these potential attacks on establishing
authority, provided that the negotiated TLS connection is secured and
the client properly verifies that the communicating server's identity
matches the target URI's authority component (see [RFC2818]).
Correctly implementing such verification can be difficult (see
[Georgiev]).

9.2. Risks of Intermediaries

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

Intermediaries that contain a shared cache are especially vulnerable
to cache poisoning attacks, as described in Section 8 of [RFC7234].

Fielding & Reschke   Standards Track   [Page 68]

Implementers need to consider the privacy and security implications
of their design and coding decisions, and of the configuration
options they provide to operators (especially the default
configuration).

Users need to be aware that intermediaries are no more trustworthy
than the people who run them; HTTP itself cannot solve this problem.

9.3. Attacks via Protocol Element Length

Because HTTP uses mostly textual, character-delimited fields, parsers
are often vulnerable to attacks based on sending very long (or very
slow) streams of data, particularly where an implementation is
expecting a protocol element with no predefined length.

To promote interoperability, specific recommendations are made for
minimum size limits on request-line (Section 3.1.1) and header fields
(Section 3.2). These are minimum recommendations, chosen to be
supportable even by implementations with limited resources; it is
expected that most implementations will choose substantially higher
limits.

A server can reject a message that has a request-target that is too
long (Section 6.5.12 of [RFC7231]) or a request payload that is too
large (Section 6.5.11 of [RFC7231]). Additional status codes related
to capacity limits have been defined by extensions to HTTP [RFC6585].

Recipients ought to carefully limit the extent to which they process
other protocol elements, including (but not limited to) request
methods, response status phrases, header field-names, numeric values,
and body chunks. Failure to limit such processing can result in
buffer overflows, arithmetic overflows, or increased vulnerability to
denial-of-service attacks.

9.4. 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

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

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.

9.5. 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 3.3.3, to reduce the effectiveness of request smuggling.

9.6. 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

Fielding & Reschke   Standards Track   [Page 70]

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 3.4) when
such verification is desired.

9.7. 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 2.7.2.

9.8. Privacy of Server Log Information

A server is in the position to save personal data about a user's
requests over time, which might identify their reading patterns or
subjects of interest. In particular, log information gathered at an
intermediary often contains a history of user agent interaction,
across a multitude of sites, that can be traced to individual users.

HTTP log information is confidential in nature; its handling is often
constrained by laws and regulations. Log information needs to be
securely stored and appropriate guidelines followed for its analysis.
Anonymization of personal information within individual entries
helps, but it is generally not sufficient to prevent real log traces
from being re-identified based on correlation with other access
characteristics. As such, access traces that are keyed to a specific
client are unsafe to publish even if the key is pseudonymous.

To minimize the risk of theft or accidental publication, log
information ought to be purged of personally identifiable
information, including user identifiers, IP addresses, and
user-provided query parameters, as soon as that information is no
longer necessary to support operational needs for security, auditing,
or fraud control.

10. Acknowledgments

This edition of HTTP/1.1 builds on the many contributions that went
into RFC 1945, RFC 2068, RFC 2145, and RFC 2616, including
substantial contributions made by the previous authors, editors, and
Working Group Chairs: Tim Berners-Lee, Ari Luotonen, Roy T. Fielding,
Henrik Frystyk Nielsen, Jim Gettys, Jeffrey C. Mogul, Larry Masinter,
and Paul J. Leach. Mark Nottingham oversaw this effort as Working
Group Chair.

Since 1999, the following contributors have helped improve the HTTP
specification by reporting bugs, asking smart questions, drafting or
reviewing text, and evaluating open issues:

Adam Barth, Adam Roach, Addison Phillips, Adrian Chadd, Adrian Cole,
Adrien W. de Croy, Alan Ford, Alan Ruttenberg, Albert Lunde, Alek
Storm, Alex Rousskov, Alexandre Morgaut, Alexey Melnikov, Alisha
Smith, Amichai Rothman, Amit Klein, Amos Jeffries, Andreas Maier,
Andreas Petersson, Andrei Popov, Anil Sharma, Anne van Kesteren,
Anthony Bryan, Asbjorn Ulsberg, Ashok Kumar, Balachander
Krishnamurthy, Barry Leiba, Ben Laurie, Benjamin Carlyle, Benjamin
Niven-Jenkins, Benoit Claise, Bil Corry, Bill Burke, Bjoern
Hoehrmann, Bob Scheifler, Boris Zbarsky, Brett Slatkin, Brian Kell,
Brian McBarron, Brian Pane, Brian Raymor, Brian Smith, Bruce Perens,
Bryce Nesbitt, Cameron Heavon-Jones, Carl Kugler, Carsten Bormann,
Charles Fry, Chris Burdess, Chris Newman, Christian Huitema, Cyrus
Daboo, Dale Robert Anderson, Dan Wing, Dan Winship, Daniel Stenberg,
Darrel Miller, Dave Cridland, Dave Crocker, Dave Kristol, Dave
Thaler, David Booth, David Singer, David W. Morris, Diwakar Shetty,
Dmitry Kurochkin, Drummond Reed, Duane Wessels, Edward Lee, Eitan
Adler, Eliot Lear, Emile Stephan, Eran Hammer-Lahav, Eric D.
Williams, Eric J. Bowman, Eric Lawrence, Eric Rescorla, Erik
Aronesty, EungJun Yi, Evan Prodromou, Felix Geisendoerfer, Florian
Weimer, Frank Ellermann, Fred Akalin, Fred Bohle, Frederic Kayser,
Gabor Molnar, Gabriel Montenegro, Geoffrey Sneddon, Gervase Markham,
Gili Tzabari, Grahame Grieve, Greg Slepak, Greg Wilkins, Grzegorz
Calkowski, Harald Tveit Alvestrand, Harry Halpin, Helge Hess, Henrik
Nordstrom, Henry S. Thompson, Henry Story, Herbert van de Sompel,
Herve Ruellan, Howard Melman, Hugo Haas, Ian Fette, Ian Hickson, Ido
Safruti, Ilari Liusvaara, Ilya Grigorik, Ingo Struck, J. Ross Nicoll,
James Cloos, James H. Manger, James Lacey, James M. Snell, Jamie

Lokier, Jan Algermissen, Jari Arkko, Jeff Hodges (who came up with
the term 'effective Request-URI'), Jeff Pinner, Jeff Walden, Jim
Luther, Jitu Padhye, Joe D. Williams, Joe Gregorio, Joe Orton, Joel
Jaeggli, John C. Klensin, John C. Mallery, John Cowan, John Kemp,
John Panzer, John Schneider, John Stracke, John Sullivan, Jonas
Sicking, Jonathan A. Rees, Jonathan Billington, Jonathan Moore,
Jonathan Silvera, Jordi Ros, Joris Dobbelsteen, Josh Cohen, Julien
Pierre, Jungshik Shin, Justin Chapweske, Justin Erenkrantz, Justin
James, Kalvinder Singh, Karl Dubost, Kathleen Moriarty, Keith
Hoffman, Keith Moore, Ken Murchison, Koen Holtman, Konstantin
Voronkov, Kris Zyp, Leif Hedstrom, Lionel Morand, Lisa Dusseault,
Maciej Stachowiak, Manu Sporny, Marc Schneider, Marc Slemko, Mark
Baker, Mark Pauley, Mark Watson, Markus Isomaki, Markus Lanthaler,
Martin J. Duerst, Martin Musatov, Martin Nilsson, Martin Thomson,
Matt Lynch, Matthew Cox, Matthew Kerwin, Max Clark, Menachem Dodge,
Meral Shirazipour, Michael Burrows, Michael Hausenblas, Michael
Scharf, Michael Sweet, Michael Tuexen, Michael Welzl, Mike Amundsen,
Mike Belshe, Mike Bishop, Mike Kelly, Mike Schinkel, Miles Sabin,
Murray S. Kucherawy, Mykyta Yevstifeyev, Nathan Rixham, Nicholas
Shanks, Nico Williams, Nicolas Alvarez, Nicolas Mailhot, Noah Slater,
Osama Mazahir, Pablo Castro, Pat Hayes, Patrick R. McManus, Paul E.
Jones, Paul Hoffman, Paul Marquess, Pete Resnick, Peter Lepeska,
Peter Occil, Peter Saint-Andre, Peter Watkins, Phil Archer, Phil
Hunt, Philippe Mougin, Phillip Hallam-Baker, Piotr Dobrogost, Poul-
Henning Kamp, Preethi Natarajan, Rajeev Bector, Ray Polk, Reto
Bachmann-Gmuer, Richard Barnes, Richard Cyganiak, Rob Trace, Robby
Simpson, Robert Brewer, Robert Collins, Robert Mattson, Robert
O'Callahan, Robert Olofsson, Robert Sayre, Robert Siemer, Robert de
Wilde, Roberto Javier Godoy, Roberto Peon, Roland Zink, Ronny
Widjaja, Ryan Hamilton, S. Mike Dierken, Salvatore Loreto, Sam
Johnston, Sam Pullara, Sam Ruby, Saurabh Kulkarni, Scott Lawrence
(who maintained the original issues list), Sean B. Palmer, Sean
Turner, Sebastien Barnoud, Shane McCarron, Shigeki Ohtsu, Simon
Yarde, Stefan Eissing, Stefan Tilkov, Stefanos Harhalakis, Stephane
Bortzmeyer, Stephen Farrell, Stephen Kent, Stephen Ludin, Stuart
Williams, Subbu Allamaraju, Subramanian Moonesamy, Susan Hares,
Sylvain Hellegouarch, Tapan Divekar, Tatsuhiro Tsujikawa, Tatsuya
Hayashi, Ted Hardie, Ted Lemon, Thomas Broyer, Thomas Fossati, Thomas
Maslen, Thomas Nadeau, Thomas Nordin, Thomas Roessler, Tim Bray, Tim
Morgan, Tim Olsen, Tom Zhou, Travis Snoozy, Tyler Close, Vincent
Murphy, Wenbo Zhu, Werner Baumann, Wilbur Streett, Wilfredo Sanchez
Vega, William A. Rowe Jr., William Chan, Willy Tarreau, Xiaoshu Wang,
Yaron Goland, Yngve Nysaeter Pettersen, Yoav Nir, Yogesh Bang,
Yuchung Cheng, Yutaka Oiwa, Yves Lafon (long-time member of the
editor team), Zed A. Shaw, and Zhong Yu.

See Section 16 of [RFC2616] for additional acknowledgements from
prior revisions.

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

11. References

11.1. Normative References

   [RFC0793]   Postel, J., "Transmission Control Protocol", STD 7,
                RFC 793, September 1981.

   [RFC1950]   Deutsch, L. and J-L. Gailly, "ZLIB Compressed Data
                Format Specification version 3.3", RFC 1950, May 1996.

   [RFC1951]   Deutsch, P., "DEFLATE Compressed Data Format
                Specification version 1.3", RFC 1951, May 1996.

   [RFC1952]   Deutsch, P., Gailly, J-L., Adler, M., Deutsch, L., and
                G. Randers-Pehrson, "GZIP file format specification
                version 4.3", RFC 1952, May 1996.

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

   [RFC3986]   Berners-Lee, T., Fielding, R., and L. Masinter,
                "Uniform Resource Identifier (URI): Generic Syntax",
                STD 66, RFC 3986, January 2005.

   [RFC5234]   Crocker, D., Ed. and P. Overell, "Augmented BNF for
                Syntax Specifications: ABNF", STD 68, RFC 5234,
                January 2008.

   [RFC7231]   Fielding, R., Ed. and J. Reschke, Ed., "Hypertext
                Transfer Protocol (HTTP/1.1): Semantics and Content",
                RFC 7231, June 2014.

   [RFC7232]   Fielding, R., Ed. and J. Reschke, Ed., "Hypertext
                Transfer Protocol (HTTP/1.1): Conditional Requests",
                RFC 7232, June 2014.

   [RFC7233]   Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed.,
                "Hypertext Transfer Protocol (HTTP/1.1): Range
                Requests", RFC 7233, June 2014.

   [RFC7234]   Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke,
                Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching",
                RFC 7234, June 2014.

   [RFC7235]   Fielding, R., Ed. and J. Reschke, Ed., "Hypertext
                Transfer Protocol (HTTP/1.1): Authentication",
                RFC 7235, June 2014.

```
RFC 7230           HTTP/1.1 Message Syntax and Routing         June 2014
```

   [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.

11.2. Informative References

   [BCP115]   Hansen, T., Hardie, T., and L. Masinter, "Guidelines
                 and Registration Procedures for New URI Schemes",
                 BCP 115, RFC 4395, February 2006.

   [BCP13]   Freed, N., Klensin, J., and T. Hansen, "Media Type
                 Specifications and Registration Procedures", BCP 13,
                 RFC 6838, January 2013.

   [BCP90]   Klyne, G., Nottingham, M., and J. Mogul, "Registration
                 Procedures for Message Header Fields", BCP 90,
                 RFC 3864, September 2004.

   [Georgiev]   Georgiev, M., Iyengar, S., Jana, S., Anubhai, R.,
                 Boneh, D., and V. Shmatikov, "The Most Dangerous Code
                 in the World: Validating SSL Certificates in Non-
                 browser Software", In Proceedings of the 2012 ACM
                 Conference on Computer and Communications Security (CCS
                 '12), pp. 38-49, October 2012,
                 <http://doi.acm.org/10.1145/2382196.2382204>.

   [ISO-8859-1] International Organization for Standardization,
                 "Information technology -- 8-bit single-byte coded
                 graphic character sets -- Part 1: Latin alphabet No.
                 1", ISO/IEC 8859-1:1998, 1998.

   [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>.

   [Kri2001]   Kristol, D., "HTTP Cookies: Standards, Privacy, and
                 Politics", ACM Transactions on Internet
                 Technology 1(2), November 2001,
                 <http://arxiv.org/abs/cs.SE/0105018>.

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

```
Fielding & Reschke          Standards Track                  [Page 75]
```

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   [RFC1919]   Chatel, M., "Classical versus Transparent IP Proxies",
                RFC 1919, March 1996.

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

   [RFC2045]   Freed, N. and N. Borenstein, "Multipurpose Internet
                Mail Extensions (MIME) Part One: Format of Internet
                Message Bodies", RFC 2045, November 1996.

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

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

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

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

   [RFC2817]   Khare, R. and S. Lawrence, "Upgrading to TLS Within
                HTTP/1.1", RFC 2817, May 2000.

   [RFC2818]   Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

   [RFC3040]   Cooper, I., Melve, I., and G. Tomlinson, "Internet Web
                Replication and Caching Taxonomy", RFC 3040,
                January 2001.

   [RFC4033]   Arends, R., Austein, R., Larson, M., Massey, D., and S.
                Rose, "DNS Security Introduction and Requirements",
                RFC 4033, March 2005.

   [RFC4559]   Jaganathan, K., Zhu, L., and J. Brezak, "SPNEGO-based
                Kerberos and NTLM HTTP Authentication in Microsoft
                Windows", RFC 4559, June 2006.

   [RFC5226]   Narten, T. and H. Alvestrand, "Guidelines for Writing
                an IANA Considerations Section in RFCs", BCP 26,
                RFC 5226, May 2008.

Fielding & Reschke   Standards Track   [Page 76]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

    [RFC5246]   Dierks, T. and E. Rescorla, "The Transport Layer
                   Security (TLS) Protocol Version 1.2", RFC 5246,
                   August 2008.

    [RFC5322]   Resnick, P., "Internet Message Format", RFC 5322,
                   October 2008.

    [RFC6265]   Barth, A., "HTTP State Management Mechanism", RFC 6265,
                   April 2011.

    [RFC6585]   Nottingham, M. and R. Fielding, "Additional HTTP Status
                   Codes", RFC 6585, April 2012.

Fielding & Reschke   Standards Track   [Page 77]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

Appendix A. 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.

A.1. Changes from HTTP/1.0

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

A.1.1. Multihomed Web Servers

The requirements that clients and servers support the Host header
field (Section 5.4), report an error if it is missing from an
HTTP/1.1 request, and accept absolute URIs (Section 5.3) are among
the most important changes defined by HTTP/1.1.

Fielding & Reschke   Standards Track   [Page 78]

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.

A.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.

A.1.3. Introduction of Transfer-Encoding

HTTP/1.1 introduces the Transfer-Encoding header field
(Section 3.3.1). Transfer codings need to be decoded prior to
forwarding an HTTP message over a MIME-compliant protocol.

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

A.2. Changes from RFC 2616

   HTTP's approach to error handling has been explained. (Section 2.5)

   The HTTP-version ABNF production has been clarified to be case-
   sensitive. Additionally, version numbers have been restricted to
   single digits, due to the fact that implementations are known to
   handle multi-digit version numbers incorrectly. (Section 2.6)

   Userinfo (i.e., username and password) are now disallowed in HTTP and
   HTTPS URIs, because of security issues related to their transmission
   on the wire. (Section 2.7.1)

   The HTTPS URI scheme is now defined by this specification;
   previously, it was done in Section 2.4 of [RFC2818]. Furthermore, it
   implies end-to-end security. (Section 2.7.2)

   HTTP messages can be (and often are) buffered by implementations;
   despite it sometimes being available as a stream, HTTP is
   fundamentally a message-oriented protocol. Minimum supported sizes
   for various protocol elements have been suggested, to improve
   interoperability. (Section 3)

   Invalid whitespace around field-names is now required to be rejected,
   because accepting it represents a security vulnerability. The ABNF
   productions defining header fields now only list the field value.
   (Section 3.2)

   Rules about implicit linear whitespace between certain grammar
   productions have been removed; now whitespace is only allowed where
   specifically defined in the ABNF. (Section 3.2.3)

   Header fields that span multiple lines ("line folding") are
   deprecated. (Section 3.2.4)

   The NUL octet is no longer allowed in comment and quoted-string text,
   and handling of backslash-escaping in them has been clarified. The
   quoted-pair rule no longer allows escaping control characters other
   than HTAB. Non-US-ASCII content in header fields and the reason
   phrase has been obsoleted and made opaque (the TEXT rule was
   removed). (Section 3.2.6)

   Bogus Content-Length header fields are now required to be handled as
   errors by recipients. (Section 3.3.2)

   The algorithm for determining the message body length has been
   clarified to indicate all of the special cases (e.g., driven by
   methods or status codes) that affect it, and that new protocol

Fielding & Reschke   Standards Track   [Page 80]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   elements cannot define such special cases. CONNECT is a new, special
   case in determining message body length. "multipart/byteranges" is no
   longer a way of determining message body length detection.
   (Section 3.3.3)

   The "identity" transfer coding token has been removed. (Sections 3.3
   and 4)

   Chunk length does not include the count of the octets in the chunk
   header and trailer. Line folding in chunk extensions is disallowed.
   (Section 4.1)

   The meaning of the "deflate" content coding has been clarified.
   (Section 4.2.2)

   The segment + query components of RFC 3986 have been used to define
   the request-target, instead of abs_path from RFC 1808. The
   asterisk-form of the request-target is only allowed with the OPTIONS
   method. (Section 5.3)

   The term "Effective Request URI" has been introduced. (Section 5.5)

   Gateways do not need to generate Via header fields anymore.
   (Section 5.7.1)

   Exactly when "close" connection options have to be sent has been
   clarified. Also, "hop-by-hop" header fields are required to appear
   in the Connection header field; just because they're defined as hop-
   by-hop in this specification doesn't exempt them. (Section 6.1)

   The limit of two connections per server has been removed. An
   idempotent sequence of requests is no longer required to be retried.
   The requirement to retry requests under certain circumstances when
   the server prematurely closes the connection has been removed. Also,
   some extraneous requirements about when servers are allowed to close
   connections prematurely have been removed. (Section 6.3)

   The semantics of the Upgrade header field is now defined in responses
   other than 101 (this was incorporated from [RFC2817]). Furthermore,
   the ordering in the field value is now significant. (Section 6.7)

   Empty list elements in list productions (e.g., a list header field
   containing ", ,") have been deprecated. (Section 7)

   Registration of Transfer Codings now requires IETF Review
   (Section 8.4)

Fielding & Reschke   Standards Track   [Page 81]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

   This specification now defines the Upgrade Token Registry, previously
   defined in Section 7.2 of [RFC2817]. (Section 8.6)

   The expectation to support HTTP/0.9 requests has been removed.
   (Appendix A)

   Issues with the Keep-Alive and Proxy-Connection header fields in
   requests are pointed out, with use of the latter being discouraged
   altogether. (Appendix A.1.2)

Appendix B. Collected ABNF

   BWS = OWS

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

   Content-Length = 1*DIGIT

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

   OWS = *(SP / HTAB)

   RWS = 1*(SP / HTAB)

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

   URI-reference = <URI-reference, see [RFC3986], Section 4.1>
   Upgrade = *("," OWS) protocol *(OWS "," [OWS protocol])

   Via = *("," OWS) (received-protocol RWS received-by [RWS comment
    ]) *(OWS "," [OWS (received-protocol RWS received-by [RWS
    comment])])

   absolute-URI = <absolute-URI, see [RFC3986], Section 4.3>
   absolute-form = absolute-URI
   absolute-path = 1*("/" segment)
   asterisk-form = "*"
   authority = <authority, see [RFC3986], Section 3.2>
   authority-form = authority

Fielding & Reschke   Standards Track   [Page 82]

```
RFC 7230          HTTP/1.1 Message Syntax and Routing          June 2014


    chunk = chunk-size [ chunk-ext ] CRLF chunk-data CRLF
    chunk-data = 1*OCTET
    chunk-ext = *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
    chunk-ext-name = token
    chunk-ext-val = token / quoted-string
    chunk-size = 1*HEXDIG
    chunked-body = *chunk last-chunk trailer-part CRLF
    comment = "(" *( ctext / quoted-pair / comment ) ")"
    connection-option = token
    ctext = HTAB / SP / %x21-27 ; '!'-'''
     / %x2A-5B ; '*'-'['
     / %x5D-7E ; ']'-'~'
     / obs-text

    field-content = field-vchar [ 1*( SP / HTAB ) field-vchar ]
    field-name = token
    field-value = *( field-content / obs-fold )
    field-vchar = VCHAR / obs-text
    fragment = <fragment, see [RFC3986], Section 3.5>

    header-field = field-name ":" OWS field-value OWS
    http-URI = "http://" authority path-abempty [ "?" query ] [ "#"
     fragment ]
    https-URI = "https://" authority path-abempty [ "?" query ] [ "#"
     fragment ]

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

    message-body = *OCTET
    method = token

    obs-fold = CRLF 1*( SP / HTAB )
    obs-text = %x80-FF
    origin-form = absolute-path [ "?" query ]

    partial-URI = relative-part [ "?" query ]
    path-abempty = <path-abempty, see [RFC3986], Section 3.3>
    port = <port, see [RFC3986], Section 3.2.3>
    protocol = protocol-name [ "/" protocol-version ]
    protocol-name = token
    protocol-version = token
    pseudonym = token

    qdtext = HTAB / SP / "!" / %x23-5B ; '#'-'['
     / %x5D-7E ; ']'-'~'
     / obs-text
    query = <query, see [RFC3986], Section 3.4>
    quoted-pair = "\" ( HTAB / SP / VCHAR / obs-text )
```

Fielding & Reschke   Standards Track   [Page 83]

```
RFC 7230           HTTP/1.1 Message Syntax and Routing          June 2014


   quoted-string = DQUOTE *( qdtext / quoted-pair ) DQUOTE

   rank = ( "0" [ "." *3DIGIT ] ) / ( "1" [ "." *3"0" ] )
   reason-phrase = *( HTAB / SP / VCHAR / obs-text )
   received-by = ( uri-host [ ":" port ] ) / pseudonym
   received-protocol = [ protocol-name "/" ] protocol-version
   relative-part = <relative-part, see [RFC3986], Section 4.2>
   request-line = method SP request-target SP HTTP-version CRLF
   request-target = origin-form / absolute-form / authority-form /
    asterisk-form

   scheme = <scheme, see [RFC3986], Section 3.1>
   segment = <segment, see [RFC3986], Section 3.3>
   start-line = request-line / status-line
   status-code = 3DIGIT
   status-line = HTTP-version SP status-code SP reason-phrase CRLF

   t-codings = "trailers" / ( transfer-coding [ t-ranking ] )
   t-ranking = OWS ";" OWS "q=" rank
   tchar = "!" / "#" / "$" / "%" / "&" / "'" / "*" / "+" / "-" / "." /
    "^" / "_" / "`" / "|" / "~" / DIGIT / ALPHA
   token = 1*tchar
   trailer-part = *( header-field CRLF )
   transfer-coding = "chunked" / "compress" / "deflate" / "gzip" /
    transfer-extension
   transfer-extension = token *( OWS ";" OWS transfer-parameter )
   transfer-parameter = token BWS "=" BWS ( token / quoted-string )

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

```
Fielding & Reschke           Standards Track                   [Page 84]
```

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

Index

   A
      absolute-form (of request-target) 42
      accelerator 10
      application/http Media Type 63
      asterisk-form (of request-target) 43
      authoritative response 67
      authority-form (of request-target) 42-43

   B
      browser 7

   C
      cache 11
      cacheable 12
      captive portal 11
      chunked (Coding Format) 28, 32, 36
      client 7
      close 51, 56
      compress (Coding Format) 38
      connection 7
      Connection header field 51, 56
      Content-Length header field 30

   D
      deflate (Coding Format) 38
      Delimiters 27
      downstream 10

   E
      effective request URI 45

   G
      gateway 10
      Grammar
         absolute-form 42
         absolute-path 16
         absolute-URI 16
         ALPHA 6
         asterisk-form 41, 43
         authority 16
         authority-form 42-43
         BWS 25
         chunk 36
         chunk-data 36
         chunk-ext 36
         chunk-ext-name 36

Fielding & Reschke   Standards Track   [Page 85]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

Fielding & Reschke   Standards Track   [Page 86]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

Fielding & Reschke   Standards Track   [Page 87]

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

RFC 7230   HTTP/1.1 Message Syntax and Routing   June 2014

Authors' Addresses

   Roy T. Fielding (editor)
   Adobe Systems Incorporated
   345 Park Ave
   San Jose, CA 95110
   USA

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

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

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