HTTP/1.1, part 1: Message Routing and Syntax"
draft-ietf-httpbis-p1-messaging-20

Abstract

   The Hypertext Transfer Protocol (HTTP) is an application-level
   protocol for distributed, collaborative, hypertext information
   systems. HTTP has been in use by the World Wide Web global
   information initiative since 1990. 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 general security concerns for implementations.

Editorial Note (To be removed by RFC Editor)

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

   The current issues list is at
   <http://tools.ietf.org/wg/httpbis/trac/report/3> and related
   documents (including fancy diffs) can be found at
   <http://tools.ietf.org/wg/httpbis/>.

   The changes in this draft are summarized in Appendix D.21.

Status of This Memo

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

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

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any

time. It is inappropriate to use Internet-Drafts as reference
material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 17, 2013.

Copyright Notice

Table of Contents

1. Introduction

The Hypertext Transfer Protocol (HTTP) is an application-level
request/response protocol that uses extensible semantics and MIME-
like 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:

    RFC xxx1: Message Routing and Syntax

    RFC xxx2: Semantics and Payloads

    RFC xxx3: Conditional Requests

    RFC xxx4: Range Requests

    RFC xxx5: Caching

    RFC xxx6: Authentication

This HTTP/1.1 specification obsoletes and moves to historic status
RFC 2616, its predecessor RFC 2068, RFC 2145 (on HTTP versioning),
and RFC 2817 (on using CONNECT for TLS upgrades).

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 HTTP 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
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. Requirement 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].

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF)
notation of [RFC5234] with the list rule extension defined in
Appendix B. Appendix C shows the collected ABNF with the list rule
expanded.

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 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.

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". 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. We use
the term "user agent" to refer to the program that initiates a
request, such as a WWW browser, editor, or spider (web-traversing
robot), and the term "origin server" to refer to the program that can
originate authoritative responses to a request. For general
requirements, we use the term "sender" to refer to whichever
component sent a given message and the term "recipient" to refer to
any component that receives the message.

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 [Part2]
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).

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).

The following example illustrates a typical message exchange for a
GET request 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: 14
Vary: Accept-Encoding
Content-Type: text/plain

Hello World!
```

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, software/firmware updaters, command-line programs,
mobile apps, and communication devices in a multitude of shapes and
sizes. Likewise, common HTTP origin servers include home automation
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 we cannot assume the
user agent can make interactive suggestions to a user or provide
adequate warning for security or privacy options. In the few cases
where this specification requires reporting of errors to the user, it
is acceptable for such reporting to only be visible in an error
console or log file. Likewise, requirements that an automated action
be confirmed by the user before proceeding can me met via advance
configuration choices, run-time options, or simply not proceeding
with the unsafe action.

2.3. Connections and Transport Independence

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 the underlying transport
protocol is outside the scope of this specification.

The specific connection protocols to be used for an interaction are
determined by client configuration and the target URI (Section 5.1).
For example, the "http" URI scheme (Section 2.8.1) 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 instead of using the defaults.

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

2.4. 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
               <              <              <              <
```
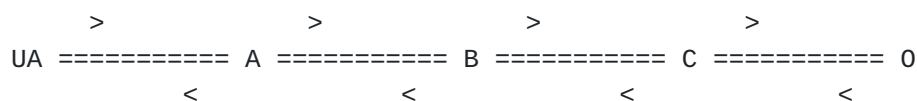
The figure above shows three intermediaries (A, B, and C) between the
user agent and origin server. A request or response message that
travels the whole chain will pass through four separate connections.
Some HTTP communication options might apply only to the connection
with the nearest, non-tunnel neighbor, only to the end-points 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
request.

We use the terms "upstream" and "downstream" to describe various
requirements in relation to the directional flow of a message: all
messages flow from upstream to downstream. Likewise, we use the
terms inbound and outbound to refer to directions in relation to the
request path: "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-layer 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.

An HTTP-to-HTTP proxy is called a "transforming proxy" if it is
designed or configured to modify request or response 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 an intranet-to-Internet
privacy filter. Such transformations are presumed to be desired by
the client (or client organization) that selected the proxy and are
beyond the scope of this specification. However, when a proxy is not
intended to transform a given message, we use the term "non-
transforming proxy" to target requirements that preserve HTTP message
semantics. See Section 4.4.4 of [Part2] and Section 7.6 of [Part6]
for status and warning codes related to transformations.

A "gateway" (a.k.a., "reverse proxy") is a receiving agent that acts
as a layer above some other server(s) and translates the received
requests to the underlying server's protocol. 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.

A gateway behaves as an origin server on its outbound connection and

as a user agent on its inbound connection. 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 MUST conform to HTTP user agent requirements
on the gateway's inbound connection and MUST implement the Connection
(Section 6.1) and Via (Section 6.2) header fields for both
connections.

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 is used to establish private communication
through a shared firewall proxy.

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 often
introduce security flaws or interoperability problems by 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. They are
indistinguishable from a man-in-the-middle attack.

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,
servers 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.

2.5. 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 which has not been cached by UA or A.

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

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
[Part6].

There are a wide variety of architectures and configurations of
caches and proxies deployed across the World Wide Web and inside
large organizations. These systems include national hierarchies of
proxy caches to save transoceanic bandwidth, systems that broadcast
or multicast cache entries, organizations that distribute subsets of
cached data via optical media, and so on.

2.6. 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.

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. Note

   that SHOULD-level requirements are relevant here, unless one of the
   documented exceptions is applicable.

   In addition to the prose requirements placed upon them, senders MUST
   NOT generate protocol elements that do not match the grammar defined
   by the ABNF rules for those protocol elements that are applicable to
   the sender's role. If a received protocol element is processed, the
   recipient MUST be able to parse any value that would match the ABNF
   rules for that protocol element, excluding only those rules not
   applicable to the recipient's role.

   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.

2.7. 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.

   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 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 defined such that, when they are understood
by a recipient, they might override or enhance the interpretation of
previously defined header fields. When an implementation receives an
unrecognized header field, the recipient MUST ignore that header
field for local processing regardless of the message's HTTP version.
An unrecognized header field received by a proxy MUST be forwarded
downstream unless the header field's field-name is listed in the
message's Connection header field (see Section 6.1). These
requirements allow HTTP's functionality to be enhanced without
requiring prior update of deployed intermediaries.

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 MUST NOT 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 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.

An HTTP 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. An HTTP client MUST NOT send a version to which it is not
conformant.

An HTTP 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 or header fields (e.g., Server)
that the server improperly handles higher request versions.

   An HTTP server SHOULD send a response version equal to the highest
   version to which the server is conformant and whose major version is
   less than or equal to the one received in the request. An HTTP
   server MUST NOT send a version to which it is not conformant. A
   server MAY send a 505 (HTTP Version Not Supported) response if it
   cannot send a response using the major version used in the client's
   request.

   An HTTP server MAY send an HTTP/1.0 response to an HTTP/1.0 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 is
   specifically avoiding any such changes to the protocol.

2.8. Uniform Resource Identifiers

   Uniform Resource Identifiers (URIs) [RFC3986] are used throughout
   HTTP as the means for identifying resources. URI references are used
   to target requests, indicate redirects, and define relationships.
   HTTP does not limit what a resource might be; it merely defines an
   interface that can be used to interact with a resource via HTTP.
   More information on the scope of URIs and resources can be found in
   [RFC3986].

   This specification adopts the definitions of "URI-reference",
   "absolute-URI", "relative-part", "port", "host", "path-abempty",
   "path-absolute", "query", and "authority" from the URI generic syntax
   [RFC3986]. In addition, we define a partial-URI rule for protocol
   elements that allow a relative URI but not a fragment.

```
   URI-reference = <URI-reference, defined in [RFC3986], Section 4.1>
   absolute-URI  = <absolute-URI, defined in [RFC3986], Section 4.3>
   relative-part = <relative-part, defined in [RFC3986], Section 4.2>
   authority     = <authority, defined in [RFC3986], Section 3.2>
   path-abempty  = <path-abempty, defined in [RFC3986], Section 3.3>
   path-absolute = <path-absolute, defined in [RFC3986], Section 3.3>
   port          = <port, defined in [RFC3986], Section 3.2.3>
   query         = <query, defined in [RFC3986], Section 3.4>
   uri-host      = <host, defined in [RFC3986], Section 3.2.2>

   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).

2.8.1. 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 connections on a given port.

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

The HTTP origin server is identified by the generic syntax's
authority component, which includes a host identifier and optional
TCP port ([RFC3986], Section 3.2.2). The remainder of the URI,
consisting of both the hierarchical path component and optional query
component, serves as an identifier for a potential resource within
that origin server's name space.

If the host identifier is provided as an IP literal or IPv4 address,
then the origin server is any listener on the indicated TCP port at
that IP address. If host is a registered name, then that name is
considered an indirect identifier and the recipient might use a name
resolution service, such as DNS, to find the address of a listener
for that host. The host MUST NOT be empty; if an "http" URI is
received with an empty host, then it MUST be rejected as invalid. If
the port subcomponent is empty or not given, then TCP port 80 is
assumed (the default reserved port for WWW services).

Regardless of the form of host identifier, access to that host is not
implied by the mere presence of its name or address. The host might
or might not exist and, even when it does exist, might or might not

be running an HTTP server or listening to the indicated port. The
"http" URI scheme makes use of the delegated nature of Internet names
and addresses to establish a naming authority (whatever entity has
the ability to place an HTTP server at that Internet name or address)
and allows that authority to determine which names are valid and how
they might be used.

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
HTTP response message, as described in Section 4 of [Part2], 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 servers that require an SSL/TLS transport
layer on a connection. Other protocols might also be used to provide
access to "http" identified resources -- it is only the authoritative
interface used for mapping the namespace 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. Senders MUST NOT
include a userinfo subcomponent (and its "@" delimiter) when
transmitting an "http" URI in a message. Recipients of HTTP messages
that contain a URI reference SHOULD parse for the existence of
userinfo and treat its presence as an error, likely indicating that
the deprecated subcomponent is being used to obscure the authority
for the sake of phishing attacks.

2.8.2. 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 for
SSL/TLS-secured connections on a given TCP port.

   All of the requirements listed above for the "http" scheme are also
   requirements for the "https" scheme, except that a default TCP port
   of 443 is assumed if the port subcomponent is empty or not given, and
   the TCP connection MUST be secured for privacy through the use of
   strong encryption prior to sending the first HTTP request.

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

   Unlike the "http" scheme, responses to "https" identified requests
   are never "public" and thus MUST NOT be reused for shared caching.
   They can, however, be reused in a private cache if the message is
   cacheable by default in HTTP or specifically indicated as such by the
   Cache-Control header field (Section 7.2 of [Part6]).

   Resources made available via the "https" scheme have no shared
   identity with the "http" scheme even if their resource identifiers
   indicate the same authority (the same host listening to the same TCP
   port). They are distinct name spaces 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].

2.8.3. 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 [RFC3986], Section 6, 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 elide the port subcomponent. Likewise, 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 (see [RFC3986], Section 2.1): the normal form is to
   not encode them.

   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 ]
```

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.

Recipients 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.

3.1. Start Line

An HTTP message can either be 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). 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

Implementations MUST NOT send whitespace between the start-line and
the first header field. 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.

3.1.1. 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 ending with CRLF.

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

A server MUST be able to parse any received message that begins with
a request-line and matches the ABNF rule for HTTP-message.

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

      method   = token

The methods defined by this specification can be found in Section 2
of [Part2], 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.

No whitespace is allowed inside the method, request-target, and
protocol version. Hence, recipients typically parse the request-line
into its component parts by splitting on the SP characters.

Unfortunately, some user agents fail to properly encode hypertext
references that have embedded whitespace, sending the characters
directly instead of properly percent-encoding the disallowed
characters. 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. Recipients 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 pre-defined limit on the length of a request-
   line. A server that receives a method longer than any that it
   implements SHOULD respond with either a 405 (Method Not Allowed), if
   it is an origin server, or a 501 (Not Implemented) status code. A
   server MUST be prepared to receive URIs of unbounded length and
   respond with the 414 (URI Too Long) status code if the received
   request-target would be longer than the server wishes to handle (see
   Section 4.6.12 of [Part2]).

   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 up to 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

   A client MUST be able to parse any received message that begins with
   a status-line and matches the ABNF rule for HTTP-message.

   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 4 of [Part2] 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 HTTP header field consists of a case-insensitive field name
followed by a colon (":"), optional whitespace, and the field value.

```
header-field   = field-name ":" OWS field-value BWS
field-name     = token
field-value    = *( field-content / obs-fold )
field-content  = *( HTAB / SP / VCHAR / obs-text )
obs-fold       = CRLF ( SP / HTAB )
                 ; obsolete line folding
                 ; see Section 3.2.2
```

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 9.10 of [Part2] as containing the
origination timestamp for the message in which it appears.

HTTP header fields are fully extensible: there is no limit on the
introduction of new field names, each presumably defining new
semantics, or 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 the standards process. 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.

New HTTP header fields SHOULD be registered with IANA according to
the procedures in Section 3.1 of [Part2]. Unrecognized header fields
MUST be forwarded by a proxy 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. Unrecognized
header fields SHOULD be ignored by other recipients.

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
wait until the entire header section is received before interpreting
a request message, since later header fields might include
conditionals, authentication credentials, or deliberately misleading
duplicate header fields that would impact request processing.

Multiple header fields with the same field name MUST NOT be sent in a
message unless the entire field value for that header field is
defined as a comma-separated list [i.e., #(values)]. Multiple header
fields with the same field name can be combined 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: The "Set-Cookie" header field as implemented in practice can
   occur multiple times, but does not use the list syntax, and thus
   cannot be combined into a single line ([RFC6265]). (See Appendix
   A.2.3 of [Kri2001] for details.) Also note that the Set-Cookie2
   header field specified in [RFC2965] does not share this problem.

3.2.1. 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. OWS SHOULD either not be produced or be produced as a
single SP. Multiple OWS octets that occur within field-content
SHOULD either be replaced with a single SP or transformed to all SP
octets (each octet other than SP replaced with SP) before
interpreting the field value or forwarding the message downstream.

RWS is used when at least one linear whitespace octet is required to
separate field tokens. RWS SHOULD be produced as a single SP.
Multiple RWS octets that occur within field-content SHOULD either be
replaced with a single SP or transformed to all SP octets before
interpreting the field value or forwarding the message downstream.

BWS is used where the grammar allows optional whitespace for
historical reasons but senders SHOULD NOT produce it in messages.
HTTP/1.1 recipients MUST accept such bad optional whitespace and
remove it before interpreting the field value or forwarding the
message downstream.

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

3.2.2. Field Parsing

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.
Any received request message that contains whitespace between a
header field-name and colon MUST be rejected 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 MAY be preceded by optional whitespace (OWS); a single
SP is preferred. The field value does not include any leading or
trailing white space: OWS occurring before the first non-whitespace
octet of the field value or after the last non-whitespace octet of
the field value is ignored and SHOULD be removed before further
processing (as this does not change the meaning of the 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 7.3.1). HTTP senders MUST NOT produce messages that include
line folding (i.e., that contain any field-value that matches the
obs-fold rule) unless the message is intended for packaging within
the message/http media type. HTTP recipients SHOULD accept line
folding and replace any embedded obs-fold whitespace with either a
single SP or a matching number of SP octets (to avoid buffer copying)
prior to interpreting the field value or forwarding the message
downstream.

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

3.2.3. Field Length

HTTP does not place a pre-defined limit on the length of header
fields, either in isolation or as a set. A server MUST be prepared
to receive request header fields of unbounded length and respond with
a 4xx (Client Error) status code if the received header field(s)
would be longer than the server wishes to handle.

A client that receives response header fields that are longer than it
wishes to handle can only treat it as a server error.

   Various ad-hoc limitations on header field length are found in
   practice. It is RECOMMENDED that all HTTP senders and recipients
   support messages whose combined header fields have 4000 or more
   octets.

3.2.4. Field value components

   Many HTTP/1.1 header field values consist of words (token or quoted-
   string) separated by whitespace or special characters. These special
   characters MUST be in a quoted string to be used within a parameter
   value (as defined in Section 4).

     word   = token / quoted-string

     token   = 1*tchar

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

     special   = "(" / ")" / "<" / ">" / "@" / ","
                     / ";" / ":" / "\" / DQUOTE / "/" / "["
                     / "]" / "?" / "=" / "{" / "}"

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

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

   The backslash octet ("\") can be used as a single-octet quoting
   mechanism within quoted-string constructs:

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

   Recipients that process the value of the quoted-string MUST handle a
   quoted-pair as if it were replaced by the octet following the
   backslash.

   Senders SHOULD NOT escape octets in quoted-strings that do not
   require escaping (i.e., other than DQUOTE and the backslash octet).

   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-cpair / comment ) ")"
   ctext            = OWS / %x21-27 / %x2A-5B / %x5D-7E / obs-text
```

The backslash octet ("\") can be used as a single-octet quoting
mechanism within comment constructs:

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

Senders SHOULD NOT escape octets in comments that do not require
escaping (i.e., other than the backslash octet "\" and the
parentheses "(" and ")").

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 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 never include
a message body because the associated response header fields (e.g.,
Transfer-Encoding, Content-Length, etc.) only indicate what their
values would have been if the request method had been GET. 2xx
(Successful) responses to CONNECT switch to tunnel mode instead of
having a message body. All 1xx (Informational), 204 (No Content),
and 304 (Not Modified) responses MUST NOT include a message body.
All other responses do include a message body, although the body MAY
be of zero length.

3.3.1. Transfer-Encoding

When one or more transfer codings are applied to a payload body in
order to form the message body, a Transfer-Encoding header field MUST
be sent in the message and MUST contain the list of corresponding
transfer-coding names in the same order that they were applied.
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 target resource.

   The "chunked" transfer-coding (Section 4.1) MUST be implemented by
   all HTTP/1.1 recipients because it plays a crucial role in delimiting
   messages when the payload body size is not known in advance. When
   the "chunked" transfer-coding is used, it MUST be the last transfer-
   coding applied to form the message body and MUST NOT be applied more
   than once in a message body. If any transfer-coding is applied to a
   request payload body, the final transfer-coding applied MUST be
   "chunked". If any transfer-coding is applied to a response payload
   body, then either the final transfer-coding applied MUST be "chunked"
   or the message MUST be terminated 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.

   If more than one Transfer-Encoding header field is present in a
   message, the multiple field-values MUST be combined into one field-
   value, according to the algorithm defined in Section 3.2, before
   determining the message body length.

   Unlike Content-Encoding (Section 5.4 of [Part2]), Transfer-Encoding
   is a property of the message, not of the payload, and thus MAY be
   added or removed by any implementation along the request/response
   chain. Additional information about the encoding parameters MAY 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 [Part4]) 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.

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

A server that receives a request message with a transfer-coding it
does not understand SHOULD respond with 501 (Not Implemented) and
then close the connection.

3.3.2. Content-Length

When a message does not have a Transfer-Encoding header field and the
payload body length can be determined prior to being transferred, a
Content-Length header field SHOULD be sent to indicate the length of
the payload body that is either present as the message body, for
requests and non-HEAD responses other than 304 (Not Modified), or
would have been present had the request been an unconditional GET.
The length is expressed as a decimal number of octets.

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

An example is

```
Content-Length: 3495
```

In the case of a response to a HEAD request, Content-Length indicates
the size of the payload body (without any potential transfer-coding)
that would have been sent had the request been a GET. In the case of
a 304 (Not Modified) response (Section 4.1 of [Part4]) to a GET
request, Content-Length indicates the size of the payload body
(without any potential transfer-coding) that would have been sent in
a 200 (OK) response.

Any Content-Length field value greater than or equal to zero is
valid. Since there is no predefined limit to the length of an HTTP
payload, recipients SHOULD anticipate potentially large decimal
numerals and prevent parsing errors due to integer conversion
overflows (Section 8.6).

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.

   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.

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 or response smuggling (bypass of security-related
checks on message routing or content) and thus ought to be
handled as an error. The provided Content-Length MUST be
removed, prior to forwarding the message downstream, or replaced
with the real message body length after the transfer-coding is
decoded.

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 MUST be
    treated as an error to prevent request or response smuggling. 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 discard
    the received response, send a 502 (Bad Gateway) status code as
    its downstream response, and then close the connection. If this
    is a response message received by a user-agent, it MUST be
    treated as an error by discarding the message and closing the
    connection.

5. If a valid Content-Length header field is present without
    Transfer-Encoding, its decimal value defines the message body
    length in octets. If the actual number of octets sent in the
    message is less than the indicated Content-Length, the recipient
    MUST consider the message to be incomplete and treat the
    connection as no longer usable. If the actual number of octets
    sent in the message is more than the indicated Content-Length,
    the recipient MUST only process the message body up to the field
    value's number of octets; the remainder of the message MUST
    either be discarded or treated as the next message in a pipeline.
    For the sake of robustness, a user-agent MAY attempt to detect
    and correct such an error in message framing if it is parsing the
    response to the last request on a connection and the connection
    has been closed by the server.

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, implementations SHOULD use 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" encoding, since some existing services respond to "chunked" with a 411 (Length Required) status code even though they understand the chunked encoding. 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 client that sends a request containing a message body MUST include 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.

3.4. Handling Incomplete Messages

Request messages that are prematurely terminated, possibly due to a canceled connection or a server-imposed time-out exception, MUST result in closure of the connection; sending an HTTP/1.1 error response prior to closing the connection is OPTIONAL.

Response messages that are prematurely terminated, usually by closure of the connection prior to receiving the expected number of octets or by failure to decode a transfer-encoded message body, MUST be recorded as incomplete. A response that terminates in the middle of the header block (before the empty line is received) cannot be assumed to convey the full semantics of the response and MUST be treated as an error.

A message body that uses the chunked transfer encoding 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 encoding 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 block was received intact.

A user agent MUST NOT render an incomplete response message body as

if it were complete (i.e., some indication needs to be given to the
user that an error occurred). Cache requirements for incomplete
responses are defined in Section 3 of [Part6].

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. Pipelining multiple requests on a connection is described
in Section 6.3.2.2.

3.5. Message Parsing Robustness

Older HTTP/1.0 client implementations might send an extra CRLF after
a POST request as a lame 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 client 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 client MUST
include the terminating CRLF octets as part of the message body
length.

In the interest of robustness, servers SHOULD ignore at least one
empty line received where a request-line is expected. In other
words, if the server is reading the protocol stream at the beginning
of a message and receives a CRLF first, it SHOULD ignore the CRLF.
Likewise, although the line terminator for the start-line and header
fields is the sequence CRLF, we recommend that recipients recognize a
single LF as a line terminator and ignore any CR.

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
MUST respond with an HTTP/1.1 400 (Bad Request) response.

4. Transfer Codings

Transfer-coding values 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 attribute/value pairs.

```
   transfer-parameter = attribute BWS "=" BWS value
   attribute          = token
   value              = word
```

   All transfer-coding values are case-insensitive. The HTTP Transfer
   Coding registry is defined in Section 7.4. HTTP/1.1 uses transfer-
   coding values in the TE header field (Section 4.3) and in the
   Transfer-Encoding header field (Section 3.3.1).

4.1. Chunked Transfer Coding

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

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

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

   chunk-ext      = *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
   chunk-ext-name = token
   chunk-ext-val  = token / quoted-str-nf
   chunk-data     = 1*OCTET ; a sequence of chunk-size octets
   trailer-part   = *( header-field CRLF )

   quoted-str-nf  = DQUOTE *( qdtext-nf / quoted-pair ) DQUOTE
                    ; like quoted-string, but disallowing line folding
   qdtext-nf      = HTAB / SP / %x21 / %x23-5B / %x5D-7E / obs-text
```

   The chunk-size field is a string of hex digits indicating the size of

the chunk-data in octets. The chunked encoding is ended by any chunk
whose size is zero, followed by the trailer, which is terminated by
an empty line.

The trailer allows the sender to include additional HTTP header
fields at the end of the message. The Trailer header field can be
used to indicate which header fields are included in a trailer (see
Section 4.4).

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

1. the request included a TE header field that indicates "trailers"
    is acceptable in the transfer-coding of the response, as
    described in Section 4.3; or,

2. the trailer fields consist entirely of optional metadata, and the
    recipient could use the message (in a manner acceptable to the
    server where the field originated) without receiving it. In
    other words, the server that generated the header field (often
    but not always the origin server) is willing to accept the
    possibility that the trailer fields might be silently discarded
    along the path to the client.

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

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

```
length := 0
read chunk-size, chunk-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 and CRLF
}
read header-field
while (header-field not empty) {
   append header-field to existing header fields
   read header-field
}
Content-Length := length
```

   Remove "chunked" from Transfer-Encoding

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

Use of chunk-ext extensions by senders is deprecated; they SHOULD NOT
be sent and definition of new chunk-extensions is discouraged.

4.2. Compression Codings

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

   Note: Use of program names for the identification of encoding
   formats is not desirable and is discouraged for future encodings.
   Their use here is representative of historical practice, not good
   design.

   Note: For compatibility with previous implementations of HTTP,
   applications SHOULD consider "x-gzip" and "x-compress" to be
   equivalent to "gzip" and "compress" respectively.

4.2.1. Compress Coding

The "compress" format is produced by the common UNIX file compression
program "compress". This format is an adaptive Lempel-Ziv-Welch
coding (LZW).

4.2.2. Deflate Coding

The "deflate" format is defined as the "deflate" compression
mechanism (described in [RFC1951]) used inside the "zlib" data format
([RFC1950]).

   Note: Some incorrect implementations send the "deflate" compressed
   data without the zlib wrapper.

4.2.3. Gzip Coding

The "gzip" format is produced by the file compression program "gzip"
(GNU zip), as described in [RFC1952]. This format is a Lempel-Ziv
coding (LZ77) with a 32 bit CRC.

4.3. TE

The "TE" header field indicates what extension transfer-codings the
client is willing to accept in the response, and whether or not it is

   willing to accept trailer fields in a chunked transfer-coding.

   Its value consists of the keyword "trailers" and/or a comma-separated
   list of extension transfer-coding names with optional accept
   parameters (as described in Section 4).

```
     TE        = #t-codings
     t-codings = "trailers" / ( transfer-extension [ te-params ] )
     te-params = OWS ";" OWS "q=" qvalue *( te-ext )
     te-ext    = OWS ";" OWS token [ "=" word ]
```

   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. This keyword is reserved for use with
   transfer-coding values even though it does not itself represent a
   transfer-coding.

   Examples of its use are:

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

   The TE header field only applies to the immediate connection.
   Therefore, the keyword MUST be supplied within a Connection header
   field (Section 6.1) whenever TE is present in an HTTP/1.1 message.

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

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

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

   2. If the transfer-coding being tested is one of the transfer-
       codings listed in the TE field, then it is acceptable unless it
       is accompanied by a qvalue of 0. (As defined in Section 4.3.1, a
       qvalue of 0 means "not acceptable".)

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

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

4.3.1. Quality Values

Both transfer codings (TE request header field, Section 4.3) and
content negotiation (Section 8 of [Part2]) use short "floating point"
numbers to indicate the relative importance ("weight") of various
negotiable parameters. A weight is normalized to a real number in
the range 0 through 1, where 0 is the minimum and 1 the maximum
value. If a parameter has a quality value of 0, then content with
this parameter is "not acceptable" for the client. HTTP/1.1
applications MUST NOT generate more than three digits after the
decimal point. User configuration of these values SHOULD also be
limited in this fashion.

```
  qvalue        = ( "0" [ "." 0*3DIGIT ] )
                / ( "1" [ "." 0*3("0") ] )
```

   Note: "Quality values" is a misnomer, since these values merely
   represent relative degradation in desired quality.

4.4. Trailer

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

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

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

If no Trailer header field is present, the trailer SHOULD NOT include
any header fields. See Section 4.1 for restrictions on the use of
trailer fields in a "chunked" transfer-coding.

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

o Transfer-Encoding

o Content-Length

o Trailer

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 [Part2], and a target resource upon which to apply those
semantics. A URI reference (Section 2.8) is typically used as an
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 identifier
component, if any, since fragment identifiers are reserved for
client-side processing ([RFC3986], Section 3.5).

HTTP intermediaries obtain the request semantics and target URI from
the request-line of an incoming request message.

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 response cache and the request semantics can be
satisfied by a cache ([Part6]), then the request is usually directed
to the cache 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 2.8.1) and
"https" (Section 2.8.2) schemes.

5.3. Request Target

Once an inbound connection is obtained (Section 6), 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

   origin-form    = path-absolute [ "?" query ]
   absolute-form  = absolute-URI
   authority-form = authority
   asterisk-form  = "*"
```

The most common form of request-target is the origin-form. 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,
then 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, containing the target URI's authority component
(excluding any userinfo).

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.

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. 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 5.6.

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

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

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

The authority-form of request-target is only used for CONNECT
requests (Section 2.3.8 of [Part2]). 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) as
the request-target. For example,

    CONNECT www.example.com:80 HTTP/1.1

The asterisk-form of request-target is only used for a server-wide
OPTIONS request (Section 2.3.1 of [Part2]). 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".

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. Since the Host field-value is
critical information for handling a request, it SHOULD be sent as the
first header field following the request-line.

   Host = uri-host [":" port] ; Section 2.8.1

A client MUST send a Host header field in all HTTP/1.1 request
messages. If the target URI includes an authority component, then
the Host field-value MUST be identical to that authority component
after excluding any userinfo (Section 2.8.1). If the authority
component is missing or undefined for the target URI, then the Host
header field MUST be sent with an empty field-value.

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

The Host header field MUST be sent 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 an HTTP/1.1 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. If the proxy forwards the request, it 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.

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

A server that receives an HTTP request message MUST reconstruct the
user agent's original target URI, based on the pieces of information
learned from the request-target, Host header field, and connection
context, in order to identify the intended target resource and
properly service the request. The URI derived from this
reconstruction process is referred to as the "effective request URI".

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

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

If the request is received over an SSL/TLS-secured TCP connection,
then the effective request URI's scheme is "https"; otherwise, the
scheme is "http".

If the request-target is in authority-form, then the effective
request URI's authority component is the same as the request-target.
Otherwise, if a Host header field is supplied with a non-empty field-
value, then the authority component is the same as the Host field-
value. Otherwise, the authority component is the concatenation of
the default host name configured for the server, a colon (":"), and
the connection's incoming TCP port number in decimal form.

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

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

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

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

has an effective request URI of

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

Example 2: the following message received over an SSL/TLS-secured TCP
connection

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

has an effective request URI of

     https://www.example.org

An origin server that does not allow resources to differ by requested
host MAY ignore the Host field-value and instead replace it with a
configured server name when constructing the effective request URI.

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

5.6. Intermediary Forwarding

As described in Section 2.4, 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.

In order to avoid request loops, a proxy that forwards requests to
other proxies MUST be able to recognize and exclude all of its own
server names, including any aliases, local variations, or literal IP
addresses.

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

host name if it is a fully qualified domain name.

A non-transforming proxy MUST NOT rewrite the "path-absolute" 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 "*".

Intermediaries that forward a message MUST implement the Connection header field as specified in Section 6.1.

5.6.1. End-to-end and Hop-by-hop Header Fields

For the purpose of defining the behavior of caches and non-caching proxies, we divide HTTP header fields into two categories:

o End-to-end header fields, which are transmitted to the ultimate recipient of a request or response. End-to-end header fields in responses MUST be stored as part of a cache entry and MUST be transmitted in any response formed from a cache entry.

o Hop-by-hop header fields, which are meaningful only for a single transport-level connection, and are not stored by caches or forwarded by proxies.

The following HTTP/1.1 header fields are hop-by-hop header fields:

o Connection

o Keep-Alive (Section 19.7.1.1 of [RFC2068])

o Proxy-Authenticate (Section 4.2 of [Part7])

o Proxy-Authorization (Section 4.3 of [Part7])

o TE

o Trailer

o Transfer-Encoding

o Upgrade

All other header fields defined by HTTP/1.1 are end-to-end header fields.

Other hop-by-hop header fields MUST be listed in a Connection header field (Section 6.1).

5.6.2. Non-modifiable Header Fields

Some features of HTTP/1.1, such as Digest Authentication, depend on
the value of certain end-to-end header fields. A non-transforming
proxy SHOULD NOT modify an end-to-end header field unless the
definition of that header field requires or specifically allows that.

A non-transforming proxy MUST NOT modify any of the following fields
in a request or response, and it MUST NOT add any of these fields if
not already present:

o Allow (Section 9.5 of [Part2])

o Content-Location (Section 9.8 of [Part2])

o Content-MD5 (Section 14.15 of [RFC2616])

o ETag (Section 2.3 of [Part4])

o Last-Modified (Section 2.2 of [Part4])

o Server (Section 9.17 of [Part2])

A non-transforming proxy MUST NOT modify any of the following fields
in a response:

o Expires (Section 7.3 of [Part6])

but it MAY add any of these fields if not already present. If an
Expires header field is added, it MUST be given a field value
identical to that of the Date header field in that response.

A proxy MUST NOT modify or add any of the following fields in a
message that contains the no-transform cache-control directive, or in
any request:

o Content-Encoding (Section 9.6 of [Part2])

o Content-Range (Section 5.2 of [Part5])

o Content-Type (Section 9.9 of [Part2])

A transforming proxy MAY modify or add these fields to a message that
does not include no-transform, but if it does so, it MUST add a
Warning 214 (Transformation applied) if one does not already appear
in the message (see Section 7.6 of [Part6]).

Warning: Unnecessary modification of end-to-end header fields
might cause authentication failures if stronger authentication
mechanisms are introduced in later versions of HTTP. Such
authentication mechanisms MAY rely on the values of header fields
not listed here.

A non-transforming proxy MUST preserve the message payload ([Part2]),
though it MAY change the message body through application or removal
of a transfer-coding (Section 4).

5.7. 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 4.3 of
[Part2]) precede a final response to the same request.

A client that uses persistent connections and sends more than one
request per connection MUST maintain a list of outstanding requests
in the order sent on that connection and MUST associate each received
response message to the highest ordered request that has not yet
received a final (non-1xx) response.

6. Connection Management

6.1. Connection

The "Connection" header field allows the sender to specify options
that are desired only for that particular connection. Such
connection options MUST be removed or replaced before the message can
be forwarded downstream by a proxy or gateway. This mechanism also
allows the sender to indicate which HTTP header fields used in the
message are only intended for the immediate recipient ("hop-by-hop"),
as opposed to all recipients on the chain ("end-to-end"), enabling
the message to be self-descriptive and allowing future connection-
specific extensions to be deployed in HTTP without fear that they
will be blindly forwarded by previously deployed intermediaries.

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

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

Connection options are compared case-insensitively.

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 sender's own connection options for the
forwarded message.

A sender MUST NOT include field-names in the Connection header field-
value for fields that are defined as expressing constraints for all
recipients in the request or response chain, such as the Cache-
Control header field (Section 7.2 of [Part6]).

The connection options do not have to 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 that
connection option. Recipients that trigger certain connection
behavior based on the presence of connection options MUST do so based
on the presence of the connection-option rather than only the
presence of the optional header field. In other words, if the
connection option is received as a header field but not indicated
within the Connection field-value, then the recipient MUST ignore the
connection-specific header field because it has likely been forwarded
by an intermediary that is only partially conformant.

When defining new connection options, specifications ought to
carefully consider existing deployed header fields and ensure that
the new connection option does not share the same name as an
unrelated header field that might already be deployed. 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.

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

   Connection: close

in either the request or the response header fields indicates that
the connection SHOULD NOT be considered "persistent" (Section 6.3)
after the current request/response is complete.

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

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

does not have a 1xx (Informational) status code.

6.2. Via

The "Via" header field MUST be sent by a proxy or gateway to indicate
the intermediate protocols and recipients between the user agent and
the server on requests, and between the origin server and the client
on responses. It is analogous to the "Received" field used by email
systems (Section 3.6.7 of [RFC5322]) and is intended to be used for
tracking message forwards, avoiding request loops, and identifying
the protocol capabilities of all senders along the request/response
chain.

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

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

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

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

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

For example, a request message could be sent from an HTTP/1.0 user
agent to an internal proxy code-named "fred", which uses HTTP/1.1 to
forward the request to a public proxy at 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 (Apache/1.1)

A proxy or gateway 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, the
received-by host of any host behind the firewall SHOULD be replaced
by an appropriate pseudonym for that host.

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

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

could be collapsed to

    Via: 1.0 ricky, 1.1 mertz, 1.0 lucy

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

6.3. Persistent Connections

6.3.1. Purpose

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

Persistent HTTP connections have a number of advantages:

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

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

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

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

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

HTTP implementations SHOULD implement persistent connections.

6.3.2. Overall Operation

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

Persistent connections provide a mechanism by which a client and a
server can signal the close of a TCP connection. This signaling
takes place using the Connection header field (Section 6.1). Once a
close has been signaled, the client MUST NOT send any more requests
on that connection.

6.3.2.1. Negotiation

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

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

including the connection option "close".

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

Clients and servers SHOULD NOT assume that a persistent connection is
maintained for HTTP versions less than 1.1 unless it is explicitly
signaled. See Appendix A.1.2 for more information on backward
compatibility with HTTP/1.0 clients.

Each persistent connection applies to only one transport link.

A proxy server MUST NOT establish a HTTP/1.1 persistent connection
with an HTTP/1.0 client (but see 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).

In order to remain persistent, all messages on the connection MUST
have a self-defined message length (i.e., one not defined by closure
of the connection), as described in Section 3.3.

6.3.2.2. Pipelining

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

Clients which assume persistent connections and pipeline immediately
after connection establishment SHOULD be prepared to retry their
connection if the first pipelined attempt fails. If a client does
such a retry, it MUST NOT pipeline before it knows the connection is
persistent. Clients MUST also be prepared to resend their requests
if the server closes the connection before sending all of the
corresponding responses.

Clients SHOULD NOT pipeline requests using non-idempotent request
methods or non-idempotent sequences of request methods (see Section
2.1.2 of [Part2]). Otherwise, a premature termination of the
transport connection could lead to indeterminate results. A client
wishing to send a non-idempotent request SHOULD wait to send that
request until it has received the response status line for the
previous request.

6.3.3. Practical Considerations

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

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

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

Clients (including proxies) SHOULD limit the number of simultaneous
connections that they maintain to a given server (including proxies).

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.

In particular, while using multiple connections avoids the "head-of-
line blocking" problem (whereby a request that takes significant
server-side processing and/or has a large payload can block
subsequent requests on the same connection), each connection used
consumes server resources (sometimes significantly), and furthermore
using multiple connections can cause undesirable side effects in
congested networks.

Note that servers might reject traffic that they deem abusive,
including an excessive number of connections from a client.

6.3.4. Retrying Requests

Senders can close the transport connection at any time. Therefore,
clients, servers, and proxies MUST be able to recover from
asynchronous close events. Client software MAY reopen the transport

connection and retransmit the aborted sequence of requests without
user interaction so long as the request sequence is idempotent (see
Section 2.1.2 of [Part2]). Non-idempotent request methods or
sequences MUST NOT be automatically retried, although user agents MAY
offer a human operator the choice of retrying the request(s).
Confirmation by user-agent software with semantic understanding of
the application MAY substitute for user confirmation. The automatic
retry SHOULD NOT be repeated if the second sequence of requests
fails.

6.4. Message Transmission Requirements

6.4.1. Persistent Connections and Flow Control

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

6.4.2. Monitoring Connections for Error Status Messages

An HTTP/1.1 (or later) client sending a message body SHOULD monitor
the network connection for an error status code while it is
transmitting the request. If the client sees an error status code,
it SHOULD immediately cease transmitting the body. If the body is
being sent using a "chunked" encoding (Section 4), a zero length
chunk and empty trailer MAY be used to prematurely mark the end of
the message. If the body was preceded by a Content-Length header
field, the client MUST close the connection.

6.4.3. Use of the 100 (Continue) Status

The purpose of the 100 (Continue) status code (see Section 4.3.1 of
[Part2]) is to allow a client that is sending a request message with
a request body to determine if the origin server is willing to accept
the request (based on the request header fields) before the client
sends the request body. In some cases, it might either be
inappropriate or highly inefficient for the client to send the body
if the server will reject the message without looking at the body.

Requirements for HTTP/1.1 clients:

o If a client will wait for a 100 (Continue) response before sending
   the request body, it MUST send an Expect header field (Section
   9.11 of [Part2]) with the "100-continue" expectation.

o A client MUST NOT send an Expect header field with the "100-
   continue" expectation if it does not intend to send a request

    body.

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

   Requirements for HTTP/1.1 origin servers:

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

   o An origin server SHOULD NOT send a 100 (Continue) response if the
      request message does not include an Expect header field with the
      "100-continue" expectation, and MUST NOT send a 100 (Continue)
      response if such a request comes from an HTTP/1.0 (or earlier)
      client. There is an exception to this rule: for compatibility
      with [RFC2068], a server MAY send a 100 (Continue) status code in
      response to an HTTP/1.1 PUT or POST request that does not include
      an Expect header field with the "100-continue" expectation. This
      exception, the purpose of which is to minimize any client
      processing delays associated with an undeclared wait for 100
      (Continue) status code, applies only to HTTP/1.1 requests, and not
      to requests with any other HTTP-version value.

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

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

   o If an origin server receives a request that does not include an
      Expect header field with the "100-continue" expectation, the
      request includes a request body, and the server responds with a
      final status code before reading the entire request body from the

transport connection, then the server SHOULD NOT close the
transport connection until it has read the entire request, or
until the client closes the connection. Otherwise, the client
might not reliably receive the response message. However, this
requirement ought not be construed as preventing a server from
defending itself against denial-of-service attacks, or from badly
broken client implementations.

Requirements for HTTP/1.1 proxies:

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

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

o Proxies SHOULD maintain a record of the HTTP version numbers
   received from recently-referenced next-hop servers.

o A proxy MUST NOT forward a 100 (Continue) response if the request
   message was received from an HTTP/1.0 (or earlier) client and did
   not include an Expect header field with the "100-continue"
   expectation. This requirement overrides the general rule for
   forwarding of 1xx responses (see Section 4.3 of [Part2]).

6.4.4. Closing Connections on Error

If the client is sending data, a server implementation using TCP
SHOULD be careful to ensure that the client acknowledges receipt of
the packet(s) containing the response, before the server closes the
input connection. If the client continues sending data to the server
after the close, the server's TCP stack will send a reset packet to
the client, which might erase the client's unacknowledged input
buffers before they can be read and interpreted by the HTTP
application.

6.5. Upgrade

The "Upgrade" header field allows the client to specify what
additional communication protocols it would like to use, if the
server chooses to switch protocols. Servers can use it to indicate
what protocols they are willing to switch to.

```
   Upgrade            = 1#protocol

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

   For example,

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

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

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

   The Upgrade header field only applies to the immediate connection.
   Therefore, the upgrade keyword MUST be supplied within a Connection
   header field (Section 6.1) whenever Upgrade is present in an HTTP/1.1
   message.

   The Upgrade header field cannot be used to indicate a switch to a
   protocol on a different connection. For that purpose, it is more
   appropriate to use a 3xx (Redirection) response (Section 4.5 of
   [Part2]).

   Servers MUST include the "Upgrade" header field in 101 (Switching
   Protocols) responses to indicate which protocol(s) are being switched
   to, and MUST include it in 426 (Upgrade Required) responses to
   indicate acceptable protocols to upgrade to. Servers MAY include it
   in any other response to indicate that they are willing to upgrade to
   one of the specified protocols.

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 2.7 and future updates to this
specification. Additional tokens can be registered with IANA using
the registration procedure defined in Section 7.6.

7. IANA Considerations

7.1. Header Field Registration

HTTP header fields are registered within the Message Header Field
Registry [RFC3864] maintained by IANA at <http://www.iana.org/
assignments/message-headers/message-header-index.html>.

This document defines the following HTTP header fields, so their
associated registry entries shall be updated according to the
permanent registrations below:

```
+-------------------+----------+----------+---------------+
| 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 6.5   |
| Via               | http     | standard | Section 6.2   |
+-------------------+----------+----------+---------------+
```

Furthermore, the header field-name "Close" shall be 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 7.1 |
+-------------------+----------+----------+-------------+
```

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

7.2. URI Scheme Registration

IANA maintains the registry of URI Schemes [RFC4395] at
<http://www.iana.org/assignments/uri-schemes.html>.

This document defines the following URI schemes, so their associated
registry entries shall be updated according to the permanent
registrations below:

```
+------------+-----------------------------------+---------------+
| URI Scheme | Description                       | Reference     |
+------------+-----------------------------------+---------------+
| http       | Hypertext Transfer Protocol       | Section 2.8.1 |
| https      | Hypertext Transfer Protocol Secure | Section 2.8.2 |
+------------+-----------------------------------+---------------+
```

7.3. Internet Media Type Registrations

This document serves as the specification for the Internet media
types "message/http" and "application/http". The following is to be
registered with IANA (see [RFC4288]).

7.3.1. 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: none

Optional parameters: version, msgtype

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

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

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

Security considerations: none

Interoperability considerations: none

Published specification: This specification (see Section 7.3.1).

Applications that use this media type:

Additional information:

   Magic number(s): none

   File extension(s): none

   Macintosh file type code(s): none

Person and email address to contact for further information: See
   Authors Section.

Intended usage: COMMON

Restrictions on usage: none

Author/Change controller: IESG

7.3.2. 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: none

Optional parameters: version, msgtype

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

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

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

Security considerations: none

Interoperability considerations: none

Published specification: This specification (see Section 7.3.2).

Applications that use this media type:

Additional information:

   Magic number(s): none

   File extension(s): none

   Macintosh file type code(s): none

Person and email address to contact for further information: See
   Authors Section.

Intended usage: COMMON

Restrictions on usage: none

Author/Change controller: IESG

7.4. Transfer Coding Registry

The HTTP Transfer Coding Registry defines the name space for transfer
coding names.

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 5.4 of [Part2]) unless the encoding transformation

is identical, as it is the case for the compression codings defined
in Section 4.2.

Values to be added to this name space require IETF Review (see
Section 4.1 of [RFC5226]), and MUST conform to the purpose of
transfer coding defined in this section.

The registry itself is maintained at
<http://www.iana.org/assignments/http-parameters>.

7.5. Transfer Coding Registrations

The HTTP Transfer Coding Registry shall be updated with the
registrations below:

```
+----------+---------------------------------------+--------------+
| Name     | Description                           | Reference    |
+----------+---------------------------------------+--------------+
| chunked  | Transfer in a series of chunks        | Section 4.1  |
| compress | UNIX "compress" program method        | Section 4.2.1 |
| deflate  | "deflate" compression mechanism       | Section 4.2.2 |
|          | ([RFC1951]) used inside the "zlib"    |              |
|          | data format ([RFC1950])               |              |
| gzip     | Same as GNU zip [RFC1952]             | Section 4.2.3 |
+----------+---------------------------------------+--------------+
```

7.6. Upgrade Token Registry

The HTTP Upgrade Token Registry defines the name space for protocol-
name tokens used to identify protocols in the Upgrade header field.
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.

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].

7.7. Upgrade Token Registration

The HTTP Upgrade Token Registry shall be updated with the
registration below:

```
+-------+----------------------+---------------------+-------------+
| Value | Description          | Expected Version    | Reference   |
|       |                      | Tokens              |             |
+-------+----------------------+---------------------+-------------+
| HTTP  | Hypertext Transfer   | any DIGIT.DIGIT     | Section 2.7 |
|       | Protocol             | (e.g, "2.0")        |             |
+-------+----------------------+---------------------+-------------+
```

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

8. Security Considerations

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

8.1. Personal Information

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

problems and generate highly adverse publicity for the implementer's
company.

8.2. Abuse of Server Log Information

A server is in the position to save personal data about a user's
requests which might identify their reading patterns or subjects of
interest. 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 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 should not be published even if the key is pseudonymous.

To minimize the risk of theft or accidental publication, log
information should 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.

8.3. Attacks Based On File and Path Names

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

8.4. DNS-related Attacks

   HTTP clients rely heavily on the Domain Name Service (DNS), and are
   thus generally prone to security attacks based on the deliberate
   misassociation of IP addresses and DNS names not protected by DNSSec.
   Clients need to be cautious in assuming the validity of an IP number/
   DNS name association unless the response is protected by DNSSec
   ([RFC4033]).

8.5. Intermediaries and Caching

   By their very nature, HTTP intermediaries are men-in-the-middle, and
   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 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.

   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.

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

8.6. Protocol Element Size Overflows

   Because HTTP uses mostly textual, character-delimited fields,
   attackers can overflow buffers in implementations, and/or perform a
   Denial of Service against implementations that accept fields with
   unlimited lengths.

   To promote interoperability, this specification makes specific
   recommendations for minimum size limits on request-line
   (Section 3.1.1) and blocks of 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.

This specification also provides a way for servers to reject messages
that have request-targets that are too long (Section 4.6.12 of
[Part2]) or request entities that are too large (Section 4.6 of
[Part2]).

Other fields (including but not limited to request methods, response
status phrases, header field-names, and body chunks) SHOULD be
limited by implementations carefully, so as to not impede
interoperability.

9. Acknowledgments

This edition of HTTP 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, Paul
J. Leach, and Mark Nottingham. See Section 16 of [RFC2616] for
additional acknowledgements from prior revisions.

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, 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,
Anne van Kesteren, Anthony Bryan, Asbjorn Ulsberg, Balachander
Krishnamurthy, Barry Leiba, Ben Laurie, Benjamin Niven-Jenkins, Bil
Corry, Bill Burke, Bjoern Hoehrmann, Bob Scheifler, Boris Zbarsky,
Brett Slatkin, Brian Kell, Brian McBarron, Brian Pane, Brian Smith,
Bryce Nesbitt, Cameron Heavon-Jones, Carl Kugler, Carsten Bormann,
Charles Fry, Chris Newman, Cyrus Daboo, Dale Robert Anderson, Dan
Winship, Daniel Stenberg, Dave Cridland, Dave Crocker, Dave Kristol,
David Booth, David Singer, David W. Morris, Diwakar Shetty, Dmitry
Kurochkin, Drummond Reed, Duane Wessels, Edward Lee, Eliot Lear, Eran
Hammer-Lahav, Eric D. Williams, Eric J. Bowman, Eric Lawrence, Eric
Rescorla, Erik Aronesty, Florian Weimer, Frank Ellermann, Fred Bohle,
Geoffrey Sneddon, Gervase Markham, Greg Wilkins, Harald Tveit
Alvestrand, Harry Halpin, Helge Hess, Henrik Nordstrom, Henry S.
Thompson, Henry Story, Herbert van de Sompel, Howard Melman, Hugo
Haas, Ian Hickson, Ingo Struck, J. Ross Nicoll, James H. Manger,
James Lacey, James M. Snell, Jamie Lokier, Jan Algermissen, Jeff
Hodges (who came up with the term 'effective Request-URI'), Jeff

Walden, Jim Luther, Joe D. Williams, Joe Gregorio, Joe Orton, John C.
Klensin, John C. Mallery, John Cowan, John Kemp, John Panzer, John
Schneider, John Stracke, John Sullivan, Jonas Sicking, Jonathan
Billington, Jonathan Moore, Jonathan Rees, Jonathan Silvera, Jordi
Ros, Joris Dobbelsteen, Josh Cohen, Julien Pierre, Jungshik Shin,
Justin Chapweske, Justin Erenkrantz, Justin James, Kalvinder Singh,
Karl Dubost, Keith Hoffman, Keith Moore, Koen Holtman, Konstantin
Voronkov, Kris Zyp, Lisa Dusseault, Maciej Stachowiak, 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, Max Clark, Michael
Burrows, Michael Hausenblas, Mike Amundsen, Mike Belshe, Mike Kelly,
Mike Schinkel, Miles Sabin, Murray S. Kucherawy, Mykyta Yevstifeyev,
Nathan Rixham, Nicholas Shanks, Nico Williams, Nicolas Alvarez,
Nicolas Mailhot, Noah Slater, Pablo Castro, Pat Hayes, Patrick R.
McManus, Paul E. Jones, Paul Hoffman, Paul Marquess, Peter Lepeska,
Peter Saint-Andre, Peter Watkins, Phil Archer, Phillip Hallam-Baker,
Poul-Henning Kamp, Preethi Natarajan, Ray Polk, Reto Bachmann-Gmuer,
Richard Cyganiak, Robert Brewer, Robert Collins, Robert O'Callahan,
Robert Olofsson, Robert Sayre, Robert Siemer, Robert de Wilde,
Roberto Javier Godoy, Roberto Peon, Ronny Widjaja, S. Mike Dierken,
Salvatore Loreto, Sam Johnston, Sam Ruby, Scott Lawrence (who
maintained the original issues list), Sean B. Palmer, Shane McCarron,
Stefan Eissing, Stefan Tilkov, Stefanos Harhalakis, Stephane
Bortzmeyer, Stephen Farrell, Stephen Ludin, Stuart Williams, Subbu
Allamaraju, Sylvain Hellegouarch, Tapan Divekar, Tatsuya Hayashi, Ted
Hardie, Thomas Broyer, 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, Yutaka
Oiwa, Zed A. Shaw, and Zhong Yu.

10. References

10.1. Normative References

[Part2]   Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed.,
               "HTTP/1.1, part 2: Semantics and Payloads",
               draft-ietf-httpbis-p2-semantics-20 (work in progress),
               July 2012.

[Part4]   Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed.,
               "HTTP/1.1, part 4: Conditional Requests",
               draft-ietf-httpbis-p4-conditional-20 (work in
               progress), July 2012.

[Part5]   Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed.,

                    "HTTP/1.1, part 5: Range Requests",
                    draft-ietf-httpbis-p5-range-20 (work in progress),
                    July 2012.

   [Part6]   Fielding, R., Ed., Lafon, Y., Ed., Nottingham, M., Ed.,
                    and J. Reschke, Ed., "HTTP/1.1, part 6: Caching",
                    draft-ietf-httpbis-p6-cache-20 (work in progress),
                    July 2012.

   [Part7]   Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed.,
                    "HTTP/1.1, part 7: Authentication",
                    draft-ietf-httpbis-p7-auth-20 (work in progress),
                    July 2012.

   [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.

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

10.2. Informative References

   [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.

   [Kri2001]   Kristol, D., "HTTP Cookies: Standards, Privacy, and
                    Politics", ACM Transactions on Internet Technology Vol.

                   1, #2, November 2001,
                   <http://arxiv.org/abs/cs.SE/0105018>.

     [Nie1997]   Frystyk, H., Gettys, J., Prud'hommeaux, E., Lie, H.,
                   and C. Lilley, "Network Performance Effects of
                   HTTP/1.1, CSS1, and PNG", ACM Proceedings of the ACM
                   SIGCOMM '97 conference on Applications, technologies,
                   architectures, and protocols for computer communication
                   SIGCOMM '97, September 1997,
                   <http://doi.acm.org/10.1145/263105.263157>.

     [Pad1995]   Padmanabhan, V. and J. Mogul, "Improving HTTP Latency",
                   Computer Networks and ISDN Systems v. 28, pp. 25-35,
                   December 1995,
                   <http://portal.acm.org/citation.cfm?id=219094>.

     [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.

[RFC2965]   Kristol, D. and L. Montulli, "HTTP State Management
               Mechanism", RFC 2965, October 2000.

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

[RFC3864]   Klyne, G., Nottingham, M., and J. Mogul, "Registration
               Procedures for Message Header Fields", BCP 90,
               RFC 3864, September 2004.

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

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

[RFC4395]   Hansen, T., Hardie, T., and L. Masinter, "Guidelines
               and Registration Procedures for New URI Schemes",
               BCP 115, RFC 4395, February 2006.

[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.

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

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

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

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

               (original report dated Aug. 1996)

Appendix A.   HTTP Version History

   HTTP has been in use by the World-Wide Web global information
   initiative since 1990. The first version of HTTP, later referred to
   as HTTP/0.9, was a simple protocol for hypertext data transfer across
   the Internet with 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 that could include metadata about the data
   transferred and modifiers 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
   new features that will either be safely ignored by an HTTP/1.0
   recipient or only sent when communicating with a party advertising
   conformance with HTTP/1.1.

   It is beyond the scope of a protocol specification to mandate
   conformance with previous versions. HTTP/1.1 was deliberately
   designed, however, to make supporting previous versions easy. We
   would expect a general-purpose HTTP/1.1 server to understand any
   valid request in the format of HTTP/1.0 and respond appropriately
   with an HTTP/1.1 message that only uses features understood (or
   safely ignored) by HTTP/1.0 clients. Likewise, we would expect an
   HTTP/1.1 client 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 wherein a buggy client failed to
   properly encode linear whitespace found in a URI reference and placed
   in 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. Multi-homed 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.

   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 a 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 need 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). Proxies/gateways MUST remove any transfer-coding
   prior to forwarding a message via a MIME-compliant protocol.

A.2. Changes from RFC 2616

   Clarify that the string "HTTP" in the HTTP-version ABNF production is
   case sensitive. Restrict the version numbers to be single digits due
   to the fact that implementations are known to handle multi-digit
   version numbers incorrectly. (Section 2.7)

   Update use of abs_path production from RFC 1808 to the path-absolute
   + query components of RFC 3986. State that the asterisk form is
   allowed for the OPTIONS request method only. (Section 5.3)

   Require that invalid whitespace around field-names be rejected.
   (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 3.2.1)

   The NUL octet is no longer allowed in comment and quoted-string text.
   The quoted-pair rule no longer allows escaping control characters
   other than HTAB. Non-ASCII content in header fields and reason
   phrase has been obsoleted and made opaque (the TEXT rule was
   removed). (Section 3.2.4)

   Empty list elements in list productions have been deprecated.
   (Appendix B)

   Require recipients to handle bogus Content-Length header fields as
   errors. (Section 3.3)

   Remove reference to non-existent identity transfer-coding value
   tokens. (Sections 3.3 and 4)

   Clarification that the chunk length does not include the count of the
   octets in the chunk header and trailer. Furthermore disallowed line
   folding in chunk extensions, and deprecate their use. (Section 4.1)

   Registration of Transfer Codings now requires IETF Review
   (Section 7.4)

Remove hard limit of two connections per server. Remove requirement
to retry a sequence of requests as long it was idempotent. Remove
requirements about when servers are allowed to close connections
prematurely. (Section 6.3.3)

Remove requirement to retry requests under certain circumstances when
the server prematurely closes the connection. (Section 6.4)

Change ABNF productions for header fields to only define the field
value.

Clarify exactly when "close" connection options have to be sent.
(Section 6.1)

Define the semantics of the Upgrade header field in responses other
than 101 (this was incorporated from [RFC2817]). (Section 6.5)

Take over the Upgrade Token Registry, previously defined in Section
7.2 of [RFC2817]. (Section 7.6)

Appendix B. 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).

Thus,

     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, recipients SHOULD accept
empty list elements. In other words, consumers would follow the list
productions:

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

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

Note that empty elements do not contribute to the count of elements present, though.

For example, given these ABNF productions:

```
  example-list      = 1#example-list-elmt
  example-list-elmt = token ; see Section 3.2.4
```

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

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

But these values would be invalid, as at least one non-empty element is required:

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

Appendix C shows the collected ABNF, with the list rules expanded as explained above.

Appendix C. 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, defined in [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, defined in [RFC3986], Section 4.3>
absolute-form = absolute-URI
asterisk-form = "*"
attribute = token
authority = <authority, defined in [RFC3986], Section 3.2>
authority-form = authority

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-str-nf
chunk-size = 1*HEXDIG
chunked-body = *chunk last-chunk trailer-part CRLF
comment = "(" *( ctext / quoted-cpair / comment ) ")"
connection-option = token
ctext = OWS / %x21-27 ; '!'-'''
 / %x2A-5B ; '*'-'['
 / %x5D-7E ; ']'-'~'
 / obs-text

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

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

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

message-body = *OCTET
method = token

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

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

qdtext = OWS / "!" / %x23-5B ; '#'-'['
 / %x5D-7E ; ']'-'~'
 / obs-text
qdtext-nf = HTAB / SP / "!" / %x23-5B ; '#'-'['
 / %x5D-7E ; ']'-'~'
 / obs-text
query = <query, defined in [RFC3986], Section 3.4>
quoted-cpair = "\" ( HTAB / SP / VCHAR / obs-text )
quoted-pair = "\" ( HTAB / SP / VCHAR / obs-text )
quoted-str-nf = DQUOTE *( qdtext-nf / quoted-pair ) DQUOTE
quoted-string = DQUOTE *( qdtext / quoted-pair ) DQUOTE
qvalue = ( "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, defined in [RFC3986], Section 4.2>
request-line = method SP request-target SP HTTP-version CRLF
request-target = origin-form / absolute-form / authority-form /
 asterisk-form

special = "(" / ")" / "<" / ">" / "@" / "," / ";" / ":" / "\" /
 DQUOTE / "/" / "[" / "]" / "?" / "=" / "{" / "}"
start-line = request-line / status-line
status-code = 3DIGIT
status-line = HTTP-version SP status-code SP reason-phrase CRLF

t-codings = "trailers" / ( transfer-extension [ te-params ] )
tchar = "!" / "#" / "$" / "%" / "&" / "'" / "*" / "+" / "-" / "." /
 "^" / "_" / "`" / "|" / "~" / DIGIT / ALPHA
te-ext = OWS ";" OWS token [ "=" word ]
te-params = OWS ";" OWS "q=" qvalue *te-ext
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 = attribute BWS "=" BWS value
```

      uri-host = <host, defined in [RFC3986], Section 3.2.2>

      value = word

      word = token / quoted-string

Appendix D. **Change Log (to be removed by RFC Editor before publication)**

D.1. **Since RFC 2616**

      Extracted relevant partitions from [RFC2616].

D.2. **Since draft-ietf-httpbis-p1-messaging-00**

      Closed issues:

      o <http://tools.ietf.org/wg/httpbis/trac/ticket/1>: "HTTP Version
         should be case sensitive"
         (<http://purl.org/NET/http-errata#verscase>)

      o <http://tools.ietf.org/wg/httpbis/trac/ticket/2>: "'unsafe'
         characters" (<http://purl.org/NET/http-errata#unsafe-uri>)

      o <http://tools.ietf.org/wg/httpbis/trac/ticket/3>: "Chunk Size
         Definition" (<http://purl.org/NET/http-errata#chunk-size>)

      o <http://tools.ietf.org/wg/httpbis/trac/ticket/4>: "Message Length"
         (<http://purl.org/NET/http-errata#msg-len-chars>)

      o <http://tools.ietf.org/wg/httpbis/trac/ticket/8>: "Media Type
         Registrations" (<http://purl.org/NET/http-errata#media-reg>)

      o <http://tools.ietf.org/wg/httpbis/trac/ticket/11>: "URI includes
         query" (<http://purl.org/NET/http-errata#uriquery>)

      o <http://tools.ietf.org/wg/httpbis/trac/ticket/15>: "No close on
         1xx responses" (<http://purl.org/NET/http-errata#noclose1xx>)

      o <http://tools.ietf.org/wg/httpbis/trac/ticket/16>: "Remove
         'identity' token references"
         (<http://purl.org/NET/http-errata#identity>)

      o <http://tools.ietf.org/wg/httpbis/trac/ticket/26>: "Import query
         BNF"

      o <http://tools.ietf.org/wg/httpbis/trac/ticket/31>: "qdtext BNF"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/35>: "Normative and
   Informative references"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/42>: "RFC2606
   Compliance"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/45>: "RFC977
   reference"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/46>: "RFC1700
   references"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/47>: "inconsistency
   in date format explanation"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/48>: "Date reference
   typo"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/65>: "Informative
   references"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/66>: "ISO-8859-1
   Reference"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/86>: "Normative up-
   to-date references"

Other changes:

o Update media type registrations to use RFC4288 template.

o Use names of RFC4234 core rules DQUOTE and HTAB, fix broken ABNF
   for chunk-data (work in progress on
   <http://tools.ietf.org/wg/httpbis/trac/ticket/36>)

D.3. Since draft-ietf-httpbis-p1-messaging-01

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/19>: "Bodies on GET
   (and other) requests"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/55>: "Updating to
   RFC4288"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/57>: "Status Code
   and Reason Phrase"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/82>: "rel_path not
      used"

   Ongoing work on ABNF conversion
   (<http://tools.ietf.org/wg/httpbis/trac/ticket/36>):

   o Get rid of duplicate BNF rule names ("host" -> "uri-host",
      "trailer" -> "trailer-part").

   o Avoid underscore character in rule names ("http_URL" -> "http-
      URL", "abs_path" -> "path-absolute").

   o Add rules for terms imported from URI spec ("absoluteURI",
      "authority", "path-absolute", "port", "query", "relativeURI",
      "host) -- these will have to be updated when switching over to
      RFC3986.

   o Synchronize core rules with RFC5234.

   o Get rid of prose rules that span multiple lines.

   o Get rid of unused rules LOALPHA and UPALPHA.

   o Move "Product Tokens" section (back) into Part 1, as "token" is
      used in the definition of the Upgrade header field.

   o Add explicit references to BNF syntax and rules imported from
      other parts of the specification.

   o Rewrite prose rule "token" in terms of "tchar", rewrite prose rule
      "TEXT".

D.4. Since draft-ietf-httpbis-p1-messaging-02

   Closed issues:

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/51>: "HTTP-date vs.
      rfc1123-date"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/64>: "WS in quoted-
      pair"

   Ongoing work on IANA Message Header Field Registration
   (<http://tools.ietf.org/wg/httpbis/trac/ticket/40>):

   o Reference RFC 3984, and update header field registrations for
      header fields defined in this document.

   Ongoing work on ABNF conversion
   (<http://tools.ietf.org/wg/httpbis/trac/ticket/36>):

   o Replace string literals when the string really is case-sensitive
      (HTTP-version).

D.5. Since draft-ietf-httpbis-p1-messaging-03

   Closed issues:

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/28>: "Connection
      closing"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/97>: "Move
      registrations and registry information to IANA Considerations"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/120>: "need new URL
      for PAD1995 reference"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/127>: "IANA
      Considerations: update HTTP URI scheme registration"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/128>: "Cite HTTPS
      URI scheme definition"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/129>: "List-type
      header fields vs Set-Cookie"

   Ongoing work on ABNF conversion
   (<http://tools.ietf.org/wg/httpbis/trac/ticket/36>):

   o Replace string literals when the string really is case-sensitive
      (HTTP-Date).

   o Replace HEX by HEXDIG for future consistence with RFC 5234's core
      rules.

D.6. Since draft-ietf-httpbis-p1-messaging-04

   Closed issues:

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/34>: "Out-of-date
      reference for URIs"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/132>: "RFC 2822 is
      updated by RFC 5322"

   Ongoing work on ABNF conversion

   (<http://tools.ietf.org/wg/httpbis/trac/ticket/36>):

   o Use "/" instead of "|" for alternatives.

   o Get rid of RFC822 dependency; use RFC5234 plus extensions instead.

   o Only reference RFC 5234's core rules.

   o Introduce new ABNF rules for "bad" whitespace ("BWS"), optional
      whitespace ("OWS") and required whitespace ("RWS").

   o Rewrite ABNFs to spell out whitespace rules, factor out header
      field value format definitions.

D.7. Since draft-ietf-httpbis-p1-messaging-05

   Closed issues:

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/30>: "Header LWS"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/52>: "Sort 1.3
      Terminology"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/63>: "RFC2047
      encoded words"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/74>: "Character
      Encodings in TEXT"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/77>: "Line Folding"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/83>: "OPTIONS * and
      proxies"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/94>: "reason-phrase
      BNF"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/111>: "Use of TEXT"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/118>: "Join
      "Differences Between HTTP Entities and RFC 2045 Entities"?"

   o <http://tools.ietf.org/wg/httpbis/trac/ticket/134>: "RFC822
      reference left in discussion of date formats"

   Final work on ABNF conversion
   (<http://tools.ietf.org/wg/httpbis/trac/ticket/36>):

o Rewrite definition of list rules, deprecate empty list elements.

o Add appendix containing collected and expanded ABNF.

Other changes:

o Rewrite introduction; add mostly new Architecture Section.

o Move definition of quality values from Part 3 into Part 1; make TE
   request header field grammar independent of accept-params (defined
   in Part 3).

D.8. Since draft-ietf-httpbis-p1-messaging-06

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/161>: "base for
   numeric protocol elements"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/162>: "comment ABNF"

Partly resolved issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/88>: "205 Bodies"
   (took out language that implied that there might be methods for
   which a request body MUST NOT be included)

o <http://tools.ietf.org/wg/httpbis/trac/ticket/163>: "editorial
   improvements around HTTP-date"

D.9. Since draft-ietf-httpbis-p1-messaging-07

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/93>: "Repeating
   single-value header fields"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/131>: "increase
   connection limit"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/157>: "IP addresses
   in URLs"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/172>: "take over
   HTTP Upgrade Token Registry"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/173>: "CR and LF in
   chunk extension values"

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/184>: "HTTP/0.9
    support"

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/188>: "pick IANA
    policy (RFC5226) for Transfer Coding / Content Coding"

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/189>: "move
    definitions of gzip/deflate/compress to part 1"

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/194>: "disallow
    control characters in quoted-pair"

Partly resolved issues:

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/148>: "update IANA
    requirements wrt Transfer-Coding values" (add the IANA
    Considerations subsection)

D.10. Since draft-ietf-httpbis-p1-messaging-08

Closed issues:

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/201>: "header
    parsing, treatment of leading and trailing OWS"

Partly resolved issues:

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/60>: "Placement of
    13.5.1 and 13.5.2"

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/200>: "use of term
    "word" when talking about header field structure"

D.11. Since draft-ietf-httpbis-p1-messaging-09

Closed issues:

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/73>: "Clarification
    of the term 'deflate'"

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/83>: "OPTIONS * and
    proxies"

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/122>: "MIME-Version
    not listed in P1, general header fields"

o   <http://tools.ietf.org/wg/httpbis/trac/ticket/143>: "IANA registry
    for content/transfer encodings"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/165>: "Case-
   sensitivity of HTTP-date"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/200>: "use of term
   "word" when talking about header field structure"

Partly resolved issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/196>: "Term for the
   requested resource's URI"

D.12. Since draft-ietf-httpbis-p1-messaging-10

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/28>: "Connection
   Closing"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/90>: "Delimiting
   messages with multipart/byteranges"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/95>: "Handling
   multiple Content-Length header fields"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/109>: "Clarify
   entity / representation / variant terminology"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/220>: "consider
   removing the 'changes from 2068' sections"

Partly resolved issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/159>: "HTTP(s) URI
   scheme definitions"

D.13. Since draft-ietf-httpbis-p1-messaging-11

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/193>: "Trailer
   requirements"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/204>: "Text about
   clock requirement for caches belongs in p6"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/221>: "effective
   request URI: handling of missing host in HTTP/1.0"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/248>: "confusing
   Date requirements for clients"

Partly resolved issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/95>: "Handling
   multiple Content-Length header fields"

D.14. Since draft-ietf-httpbis-p1-messaging-12

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/75>: "RFC2145
   Normative"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/159>: "HTTP(s) URI
   scheme definitions" (tune the requirements on userinfo)

o <http://tools.ietf.org/wg/httpbis/trac/ticket/210>: "define
   'transparent' proxy"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/224>: "Header Field
   Classification"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/233>: "Is * usable
   as a request-uri for new methods?"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/240>: "Migrate
   Upgrade details from RFC2817"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/276>: "untangle
   ABNFs for header fields"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/279>: "update RFC
   2109 reference"

D.15. Since draft-ietf-httpbis-p1-messaging-13

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/53>: "Allow is not
   in 13.5.2"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/95>: "Handling
   multiple Content-Length header fields"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/276>: "untangle
   ABNFs for header fields"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/286>: "Content-
   Length ABNF broken"

D.16. Since draft-ietf-httpbis-p1-messaging-14

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/273>: "HTTP-version
   should be redefined as fixed length pair of DIGIT . DIGIT"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/282>: "Recommend
   minimum sizes for protocol elements"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/283>: "Set
   expectations around buffering"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/288>: "Considering
   messages in isolation"

D.17. Since draft-ietf-httpbis-p1-messaging-15

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/100>: "DNS Spoofing
   / DNS Binding advice"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/254>: "move RFCs
   2145, 2616, 2817 to Historic status"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/270>: "\-escaping in
   quoted strings"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/305>: "'Close'
   should be reserved in the HTTP header field registry"

D.18. Since draft-ietf-httpbis-p1-messaging-16

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/186>: "Document
   HTTP's error-handling philosophy"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/215>: "Explain
   header field registration"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/219>: "Revise
   Acknowledgements Sections"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/297>: "Retrying
   Requests"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/318>: "Closing the
   connection on server error"

D.19. Since draft-ietf-httpbis-p1-messaging-17

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/158>: "Proxy-
   Connection and Keep-Alive"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/166>: "Clarify 'User
   Agent'"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/300>: "Define non-
   final responses"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/323>: "intended
   maturity level vs normative references"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/324>: "Intermediary
   rewriting of queries"

D.20. Since draft-ietf-httpbis-p1-messaging-18

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/250>: "message-body
   in CONNECT response"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/302>: "Misplaced
   text on connection handling in p2"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/335>: "wording of
   line folding rule"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/343>: "chunk-
   extensions"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/346>: "make IANA
   policy definitions consistent"

D.21. **Since draft-ietf-httpbis-p1-messaging-19**

Closed issues:

o <http://tools.ietf.org/wg/httpbis/trac/ticket/346>: "make IANA
   policy definitions consistent"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/359>: "clarify
   connection header field values are case-insensitive"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/361>: "ABNF
   requirements for recipients"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/368>: "note
   introduction of new IANA registries as normative changes"

o <http://tools.ietf.org/wg/httpbis/trac/ticket/374>: "Reference to
   ISO-8859-1 is informative"

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/

    Yves Lafon (editor)
    World Wide Web Consortium
    W3C / ERCIM
    2004, rte des Lucioles
    Sophia-Antipolis, AM 06902
    France

    EMail: ylafon@w3.org
    URI: http://www.raubacapeu.net/people/yves/

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

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