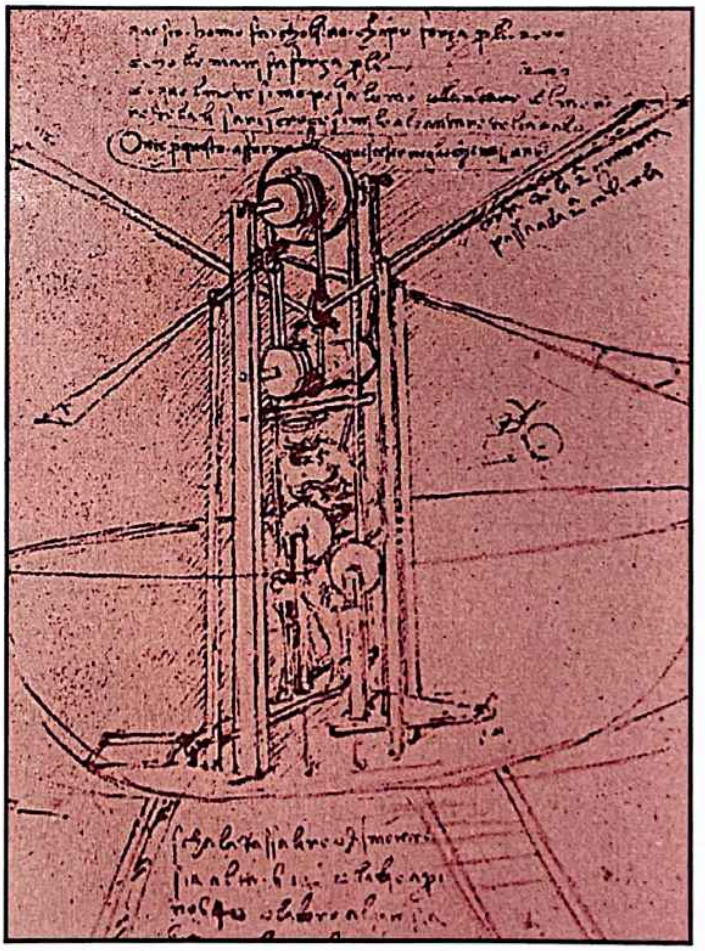
PRENTICE HALL PTR WEB INFRASTRUCTURE SERIES

WEB PROXY SERVERS

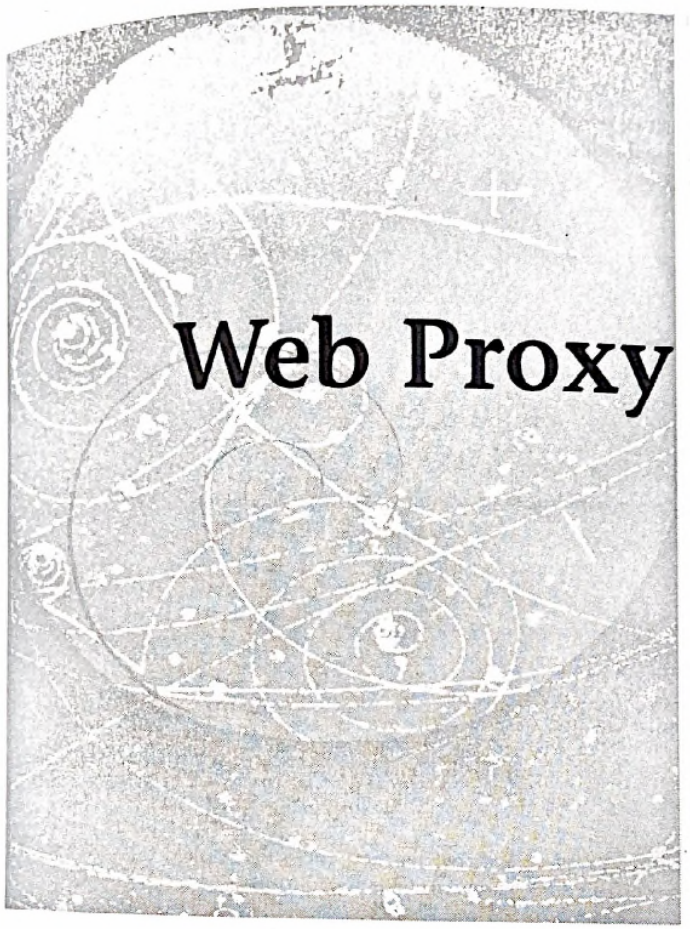
From the Chief Architect of the Netscape Proxy Server and Co-Developer of the CERN Proxy Server

Thorough Coverage of Proxy Network Design, Management, and Optimization



ARI LUOTONEN

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 1



Web Proxy Servers

Ari Luotonen

Paul Mockapetris, ***Series Advisor***

To join a Prentice Hall PTR Internet mailing list, point to
http://www.prenhall.com/mail_lists/

Prentice Hall PTR, Upper Saddle River, NJ 07458

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 2

Library of Congress Cataloging-in-Publication Date

Luotonen, Ari.
Web proxy servers / Ari Luotonen.
p. cm. — (Web infrastructure series)
Includes bibliographical references and index.
ISBN: 0-13-680612-0 (alk. paper)
1. Web servers. 2. Title. II. Series.
TK5105.888.L86 1997
005.7'1376—dc21 97-40619
CIP

Editorial/Production Supervision: *Kathleen M. Caren*
Acquisitions Editor: *Mary Franz*
Editorial Assistant: *Noreen Regina*
Series Advisor: *Paul Mockapetris*
Marketing Manager: *Miles Williams*
Buyer: *Alexis Heydt*
Cover Design: *Anthony Gemmellaro*; Illustration: *Leonardo da Vinci*
Cover Design Direction: *Jerry Votta*
Series Design: *Meg Van Arsdale*
Art Director: *Gail Cocker-Bogusz*

© 1998 Netscape Communications Corporation
Published by Prentice Hall PTR
Prentice-Hall, Inc., A Simon & Schuster Company
Upper Saddle River, NJ 07458

Prentice Hall books are widley used by permission by corporations and government agencies for training, marketing, and resale. The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:

Corporate Sales Department,
Prentice Hall PTR
One Lake Street
Upper Saddle River, NJ 07458
Phone: 800-382-3419; FAX: 201-236-714
E-mail (Internet): corpsales@prenhall.com

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-680612-0

Prentice-Hall International (UK) Limited, London
Prentice-Hall of Australia Pty. Limited, Sydney
Prentice-Hall Canada Inc., Toronto
Prentice-Hall Hispanoamericana, S.A., Mexico
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Simon & Schuster Asia Pte. Ltd., Singapore
Editora Prentice-Hall do Brasil, Ltda., Rio de Janeiro

CHAPTER

4

The HTTP Protocol

This chapter provides an overview and reference of the HyperText Transfer Protocol, commonly referred to as HTTP. Aspects related to proxy servers are emphasized. Some features not related to proxy servers have been omitted. While the World Wide Web consists of, and is built on top of, a plethora of different protocols, HTTP is the primary protocol used for transferring Web documents. Other common Internet protocols related to the Web and proxying are discussed throughout the rest of this part of the book.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 4

The first sections of this chapter are an overview of the main aspects of the HTTP protocol and are recommended reading in order to understand how HTTP works. The last three sections are a complete list of official HTTP request methods (the section on HTTP request methods on page 64), headers (the section on HTTP headers on page 69, and response status codes (the section on HTTP response status codes on page 96). Those sections may be arduous reading. Don't feel guilty about skipping them—but they may become handy as a reference to HTTP at a later time.

OVERALL OPERATION OF HTTP

HTTP is a request/response protocol. The client sends a request to the server, and the server sends back a response. There are no multiple-step handshakes in the beginning as with some other protocols, such as FTP.

There may be intermediate proxy servers between the client and the server; the client may send a request to the proxy server, and the proxy will forward the request to the server, or another proxy. This is called a request chain. The response comes back through the same path, the response chain. Intermediate proxy servers may cache resources and may return a cached copy without forwarding the request to the origin server.

An HTTP request consists of a *method*, a target URL, protocol version identifier, and a set of *headers*. The method specifies the type of operation. The most common method is GET, which is used to retrieve documents. POST is used to perform HTML form submissions. The section on HTTP request methods on page 64 describes the methods in more detail. Headers contain additional information to the request; the section on HTTP headers on page 69 lists all currently defined HTTP headers.

An HTTP response consists of a protocol version identifier, status code, human-readable response status line, response headers, and the requested resource content. Status codes are listed and described in the section on HTTP response status codes on page 96.

Extensive examples of the actual protocol are shown in the following sections. But first, let's step back and look at the history and evolution of HTTP.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 5

DESIGN GOALS OF HTTP

The HTTP protocol was designed with simplicity, extensibility, compatibility, and speed in mind. The goal was to make it easy to implement and debug applications that use HTTP as their communication protocol. Another important factor was to make it suitable for a global hypermedia information system, which had some definite requirements on the protocol. The following sections provide more insight into these design goals.

Simplicity

HTTP is a simple ASCII text protocol. Unlike binary protocols which are hard to debug, HTTP can easily be debugged by simply using the `telnet` program to connect to the proxy server port and mimicing the HTTP protocol by hand (see Chapter 22 on troubleshooting).

The first version of HTTP, later given the version number 0.9, was very simple; the request simply contained the request method—in practice always GET—and the URL of the document that was being requested, followed by a <cr><lf> [1]:

GET *URL* <cr><lf>

HTTP/0.9 supports only plain retrieval of documents, with no access control, or any other fancy features. The response simply contains the requested document, with no other information. The connection is closed immediately after the document transfer is complete.

Extensibility

The HTTP/1.0 version introduced request and response *headers*. Headers allowed the HTTP protocol to be extended in a flexible manner. It was now possible to send authentication credentials, state information, content negotiation directives, and other data that could be used in the context of request processing. The response would no longer depend solely on the requested URL, but possibly on the other information passed in the headers, such as the user's identity, or the client software version. Users could now get customized pages tailored specifically for them and their client software.

Compatibility

When designing a global application used by millions of people, compatibility with different protocol versions is of utmost importance. Packaged

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 6

applications targeted toward single companies may get away with changing their protocol between releases in an incompatible manner because the upgrade can be done for all users at once. However, in a global scale deployment with multiple client and server software vendors, interoperability between different protocol versions is required.

The HTTP specification has provisions for supporting older versions of the protocol, as well as ignoring new features that are not known by the current protocol.

Lightweight

A protocol used for transferring hypertext documents faces certain requirements. Hypertext documents consist of text that contains pointers, or hypertext links, to other documents. The nature of the system is such that a fast document transfer is followed by a fairly long period of inactivity—while the user is reading the document. After a while, the user will click on a link, which initiates a new retrieval.

This nature imposes the following requirements:

- The overhead of the protocol must be minimal in order to provide fast interactive performance.
- It must be fast to establish a connection to a hypertext server.
- Connection should not remain open while the user is reading the document.

These criteria were met fairly well by HTTP/0.9 and HTTP/1.0. However, as the Web developed, more embedded data appeared on pages, such as inlined images and Java applets. Each of these cause their own HTTP request, and the page load no longer mapped to a single HTTP request; instead, there would be several. This gave rise to the need for more efficient transport, in the form of *persistent connections* (see the section on HTTP persistent connections (keep-alive) on page 47), and later possibly as a multiplexing protocol (see the section on Multiplexed Sessions on page 50).

HTTP/0.9

As previously stated, the first version of HTTP, referred to as HTTP/0.9, was very simple: it supported only a single method, GET. This first version was sufficient for retrieving documents, but it provided no authenti-

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 7

cation or access control features other than those based on the IP address and the DNS host and domain names of the requesting client. The HTTP/0.9 response contained only the requested document, with no additional information.

Document Typing

The typing of documents in HTTP/0.9 was based on the filename extension present in the URL. This task was performed by HTTP/0.9 clients—the server had no way of communicating the document type to the client. Later, the introduction of headers in HTTP/1.0 made it possible to have the document type be determined by the server and passed to the client in the `Content-Type:` header (see page 94).

In HTTP/0.9, the extension `.html` in the URL indicated an HTML document [2], `.txt` a plain text document, `.gif` an image in GIF format, and `.jpg` an image in JPEG format. The client had to inspect the URL to find out this information. This method was also prone to error since some files would not have any extension. Clients would have to "guess" the type, and even though it was successful most of the time, there would be cases where the client guessed wrong, and the result was garbled.

HTTP/1.0 changed the document typing paradigm; it was moved to the server, and the clients were not allowed to look at the URL filename extension. This enabled the servers to have full control over the document typing and freed the naming scheme from the Web requirements. That is, servers could freely implement their own typing system—often it is based on the filename extensions, but nothing prevents other mechanisms.

HTTP/1.0

The HTTP/1.0 protocol is documented in the Informational RFC 1945 titled "Hypertext Transfer Protocol—HTTP/1.0."

HTTP/1.0 introduced a new, extended format for requests and responses allowing more data to be passed in both directions. After the actual request, a set of *header fields* follow. These are simple name-value pairs which allow additional information, such as authentication credentials, to be passed to the server. The HTTP header section is similar to Multipurpose Internet Mail Extensions (MIME).

Similarly, in addition to the document content, the response also includes a status line and its own header section. The response header sec-

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 8

tion can contain information such as the type of the document (the `Content-Length:` header) and its length (`Content-Length:`).

Another addition was the introduction of two new methods to complement `GET`: `HEAD` for querying only the header information of the document and `POST` which enabled HTML form submissions [3].

An HTTP/1.0 style request for a URL, say, `http://www.`*somesite*`.com/`*somedir/page.html* looks like this:

```
GET /somedir/page.html HTTP/1.0<cr><lf>
User-agent: Mozilla/4.0<cr><lf>
Accept: text/html, image/gif, image/jpeg<cr><lf>
<cr><lf>
```

The response looks something like this:

```
HTTP/1.0 200 Ok<cr><lf>
Server: Netscape-Enterprise/3.0<cr><lf>
Date: Sat, 26 Apr 1997 06:03:24 GMT<cr><lf>
Content-type: text/html<cr><lf>
Content-length: 5361<cr><lf>
<cr><lf>
... document content...
```

HTTP/1.0 Proxy Servers

The HTTP specification [RFC 1945] defines the term "proxy" as follows:

> An intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them, with possible translation, on to other servers. A proxy must interpret and, if necessary, rewrite a request message before forwarding it.
>
> Proxies are often used as client-side portals through network firewalls and as helper applications for handling requests via protocols not implemented by the user agent.

The simplest case of HTTP is when the connection is established directly from the client to the origin server. However, in practice this is often not the case. Instead, the request is made through one or more intermediary proxy servers. The general case is that there are numerous proxies in between the client and the origin server.

Furthermore, a proxy server may have a *cache* where it stores commonly accessed documents so that a given request might not actually

even go all the way to the origin server but instead be satisfied by a proxy server's cache, by reusing an earlier response to an equivalent request.

The HTTP specification [RFC 1945] defines the term "cache" as follows:

> A program's local store of 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 include a cache, though a cache cannot be used by a server while it is acting as a tunnel.

In other words, caching is not merely a feature of a proxy server. Client programs often have their own caches. Surprisingly, even origin servers may use caching for their internal purposes. For example, busy origin servers may cache the generated header section and reuse the same headers for equivalent requests over and over again, instead of reconstructing them every time.

Differences in the Use of HTTP

In addition to client/server communication, HTTP is used between clients and proxy servers, as well as between proxies and other proxies. But there is an important difference between a request made directly to an origin server, and one made through a proxy: the requested URL is used in its full form, including the protocol prefix, hostname, and the optional port number. They are omitted when the request is made directly to the origin server [4]. For example, a request for the URL `http://home.netscape.com/people/ari/` from a client to a proxy server would look like this (the <cr><lf> are not shown in this example anymore—however, there *is* always an empty line terminating the header section):

```
GET http://home.netscape.com/people/ari/ HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
```

but when forwarded to the origin server by the proxy, the request is rewritten to include only the URL path part:

```
GET /people/ari/ HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Forwarded: by http://proxy.mycompany.com:8080
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 10

Note also that HTTP/1.0 proxy servers may add the `Forwarded:` header indicating that the request passed through a proxy, although the HTTP/1.0 specification does not mention that; it was not included in the final specification, and HTTP/1.1 deprecates it by introducing the `Via:` header for that purpose.

Of course, if a proxy server forwards the request to another proxy server, the full URL is retained. Only the last proxy in the proxy chain, connecting directly to the origin server, will drop the protocol, host, and optional port part of the URL.

Sidebar

In the case of transparent proxy servers (see the section on Transparent Proxying on page 14), it is clearly not possible to know if there are any more proxy servers downstream. The request may be redirected to a transparent proxy server by routers. However, in this scenario many things do not work according to the HTTP/1.1 specification (which makes no provisions for transparent proxies).

Backward Compatibility

The HTTP/1.0 specification requires backward compatibility by clients and server to interoperate with HTTP/0.9. HTTP/1.0 servers must recognize both HTTP/0.9 and HTTP/1.0 requests and respond with the same protocol version. HTTP/1.0 clients must recognize both HTTP/0.9 and HTTP/1.0 responses [5].

Protocol Upgrading and Downgrading

Proxy servers that have a lower protocol version than the requesting client must downgrade the request to match the version of the proxy. This is because the request protocol version number indicates the level of capabilities of the sender (in this case the proxy), so the origin server must be notified about what version the proxy server it is talking to.

Proxy servers with a higher protocol version than the requesting client may upgrade the request before forwarding it, but, naturally, they need to downgrade the received response to the level of the client.

THE HTTP/1.1 PROTOCOL

The HTTP/1.1 specification is the Standards Track RFC 2068 titled "Hypertext Transfer Protocol—HTTP/1.1." The major improvements in HTTP/1.1 over version 1.0 are

Persistent connections. (see the section on HTTP persistent connections on page 47) HTTP/1.1 allows connections to remain open over several requests [6].

Request pipelining. (see page 53) Together with persistent connections, request pipelining reduces latency between requests and responses and delivers better perceived performance.

Cache control. (the sections on HTTP/1.1 Cache Control Terminology on page 173 and HTTP/1.1 Cache Control on page 176) One of the biggest missing features in HTTP/1.0 is the absence of an explicit cache control mechanism. HTTP/1.1 introduces a variety of directives that can be used to control caching on proxies and in clients.

Formalized validation model (conditional requests). (see page 158) HTTP/1.0 mentioned only the conditional GET feature as a mechanism to perform up-to-date checks. HTTP/1.1 formalizes the HTTP validation model and provides opaque validators (see page 95), instead of just the last modification date and time used by HTTP/1.0.

Content variants. (see page 88) HTTP/1.1 provides the basic utilities for associating multiple representations of a resource under a single URL. This is useful when providing a resource in multiple languages or different data formats.

Protocol tracing. (see the section on HTTP tracing on page 370) HTTP/1.1 specifies a new method, TRACE, which is useful in debugging proxy chains (see the section on proxy chaining on page 22).

HTTP PERSISTENT CONNECTIONS (KEEP-ALIVE)

HTTP is a simple request-response protocol. In basic HTTP/1.0, each request is made over a new connection. After the data transfer is complete, the connection is torn down, and a new one is established to get another resource from the same server.

Due to TCP's three-way handshake [7] there is a fair amount of latency in establishing each new connection. Therefore it would be beneficial to *reuse* the TCP connections to make multiple HTTP requests over a single connection. For this reason, an extension to the HTTP/1.0 pro-

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 12

tocol was made, the so-called *keep-alive* feature. Figure 4.1 illustrates one-shot requests compared to persistent connections used to perform several requests.

The keep-alive (persistent connection) feature allows the same connection to remain open for multiple requests. Obviously, the drawback is that the next request processing cannot start before the previous response has been sent by the server.

TCP also has a feature called *slow-start*. Basically, it is a flow control mechanism that slows down the speed at which data is sent until the protocol determines how fast the network link between the sender and receiver is. This mechanism is designed to prevent bogging down the network by shoving down as much data as possible, only to realize that the pipe is not big enough and depriving other applications of the network bandwidth that they need. In other words, due to slow-start, a TCP connection takes a fair amount of time to reach full throughput. In the case of HTTP with its short-lived connections, the slow-start makes the HTTP transactions take longer for small files, and they speed up for larger files. Using the persistent connection feature alleviates this problem as well.

Persistent Connections in HTTP/1.0

The keep-alive feature in HTTP/1.0 is invoked by the client first sending the following header in the request:

```
Connection: keep-alive
```

and if the server supports this feature, the server will respond with the same header in its response:

```
Connection: keep-alive
```

After that, the server will not close the connection which is the default behavior upon completion of data transfer, but instead it leaves the connection open and waits for the next request to come over the same connection.

Persistent Connections in HTTP/1.1

In HTTP/1.1, the persistent connection feature changed so that it became the default. The client and server must now explicitly specify if they do not want persistent connections by sending the header

```
Connection: close
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 13

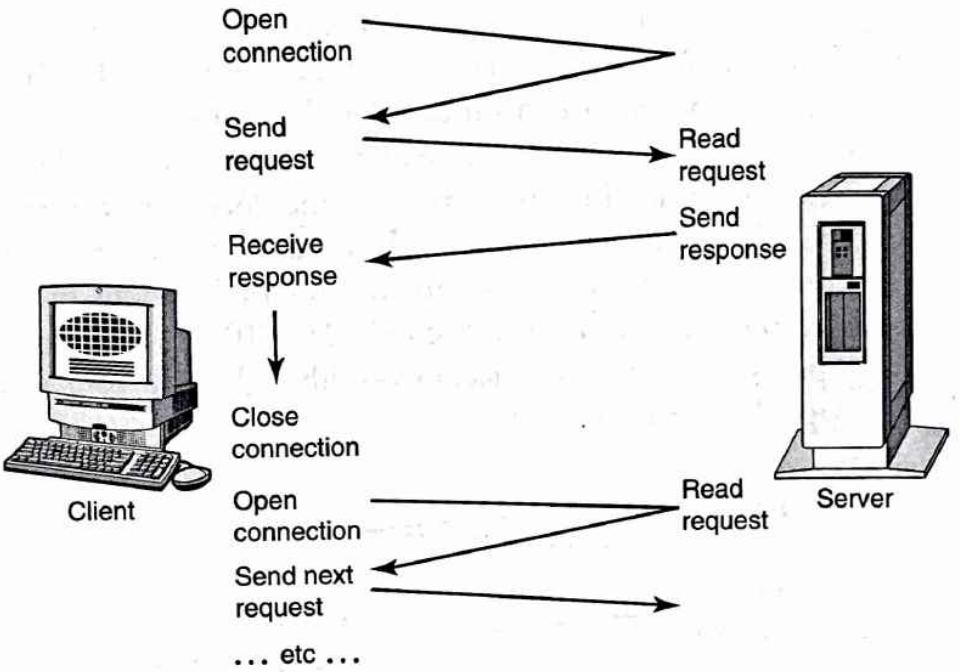
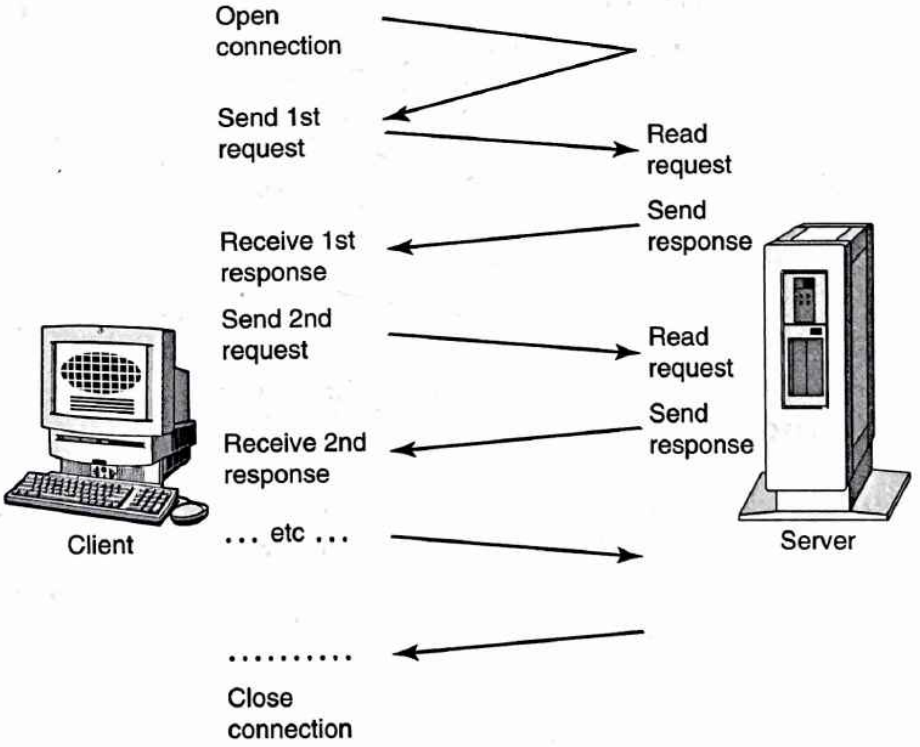


Figure 4.1 a) HTTP without persistent connections. Each request is made over a new connection.



b) HTTP with persistent connections. A connection is reused for performing several requests in a row.

Multiple Simultaneous Connections

Many Web clients, such as Netscape Navigator [8], open up several connections simultaneously, so that the document text (HTML) and its inlined images can be loaded simultaneously to improve the (perceived) response time (Figure 4.2). The speed improvement is caused by two reasons. First, the retrieval for the images in the document can start before the entire HTML file has been loaded. Multiple images can be rendered simultaneously, instead of having to wait for the previous one to finish before starting the next one. Second, TCP/IP tends to give better throughput with multiple connections—although such "connection hogging" might be considered rude.

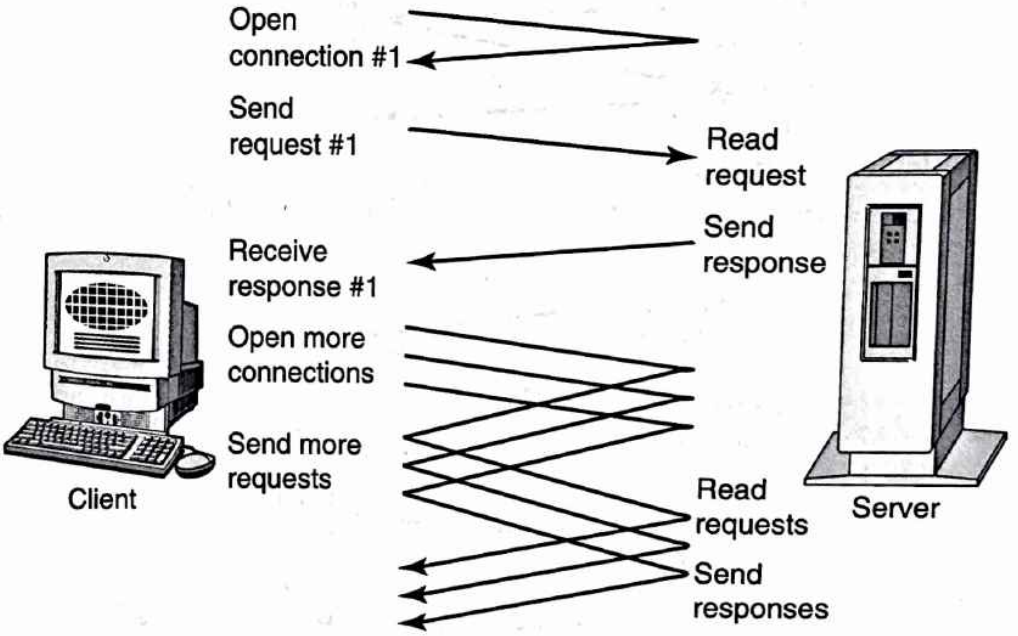


Figure 4.2 Multiple simultaneous connections to increase speed and provide better perceived performance.

Images in the current document view receive priority. They are laid out first, and images that are currently hidden outside the document view will be retrieved only after everything in the current view has been retrieved and rendered. All of this contributes to the speed perceived by the user.

Multiplexed Sessions

A widely suggested and generally agreed-upon next step is to allow multiple *sessions* to be multiplexed (interleaved) over a single connection. This

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 15

new architecture is commonly referred to as *HTTP-NG* (HTTP—Next Generation), or HTTP/2.0.

The idea is simple: add a session layer on top of the connection, and each one of the sessions is an individual HTTP request-response transaction. The session layer interleaves the chunks from each session into a single connection (Figure 4.3). At the receiving end, chunks are read, and based on their session ID they get passed to the handler of that session (request). The next section illustrates a session layer protocol by a simple example.

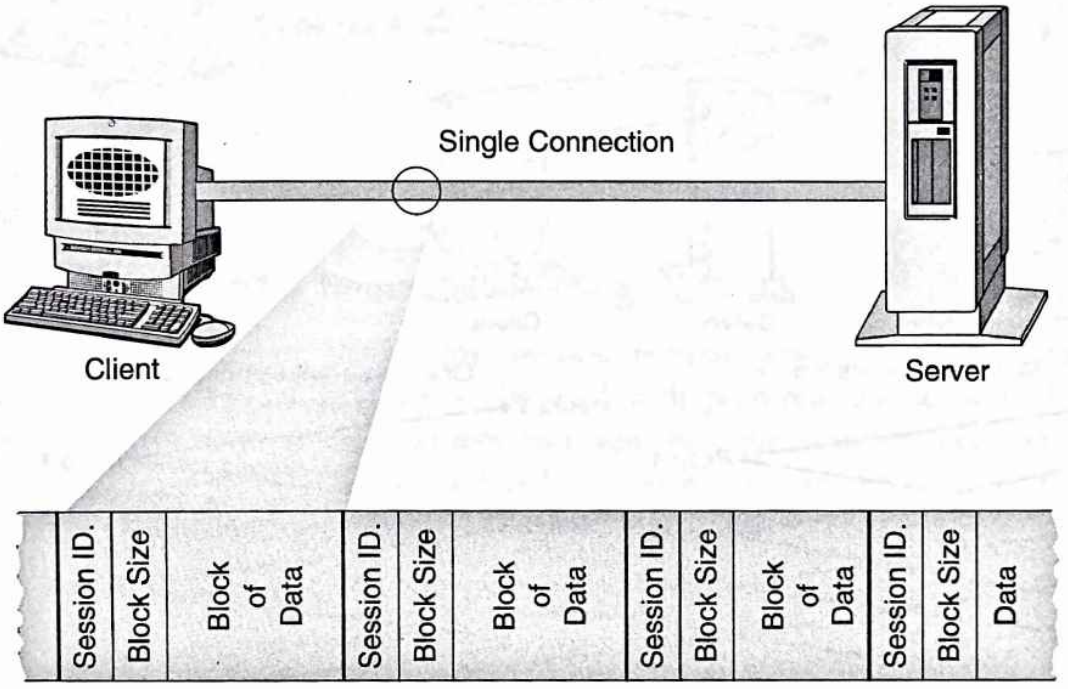


Figure 4.3 A session layer protocol.

Simple Session Layer Protocol Example

Each chunk of data is prefixed with a session identifier (e.g., 2 bytes), and the length of the chunk (2 bytes). The actual data within each chunk belongs to the session identified by the session identifier.

This way multiple HTTP requests and responses can be sent and received over a single TCP connection.

Comparing Multiple Connections vs. Single Persistent Connection

A single persistent connection is more network friendly. A long-lived connection handling several requests starts to benefit from decongestion algorithms used in the network layer of the operating system (or the TCP

stack). On the other hand, a single one-shot connection for each request doesn't benefit from them, because by the time the algorithms start to figure out the state of the network (whether it's fast or congested), the connection is already torn down.

However, by comparison, multiple simultaneous connections yield a much better perceived speed than a single persistent connection (Figure 4.4). This is obvious, because each single transfer must complete before the next one can start in the persistent connection scenario.

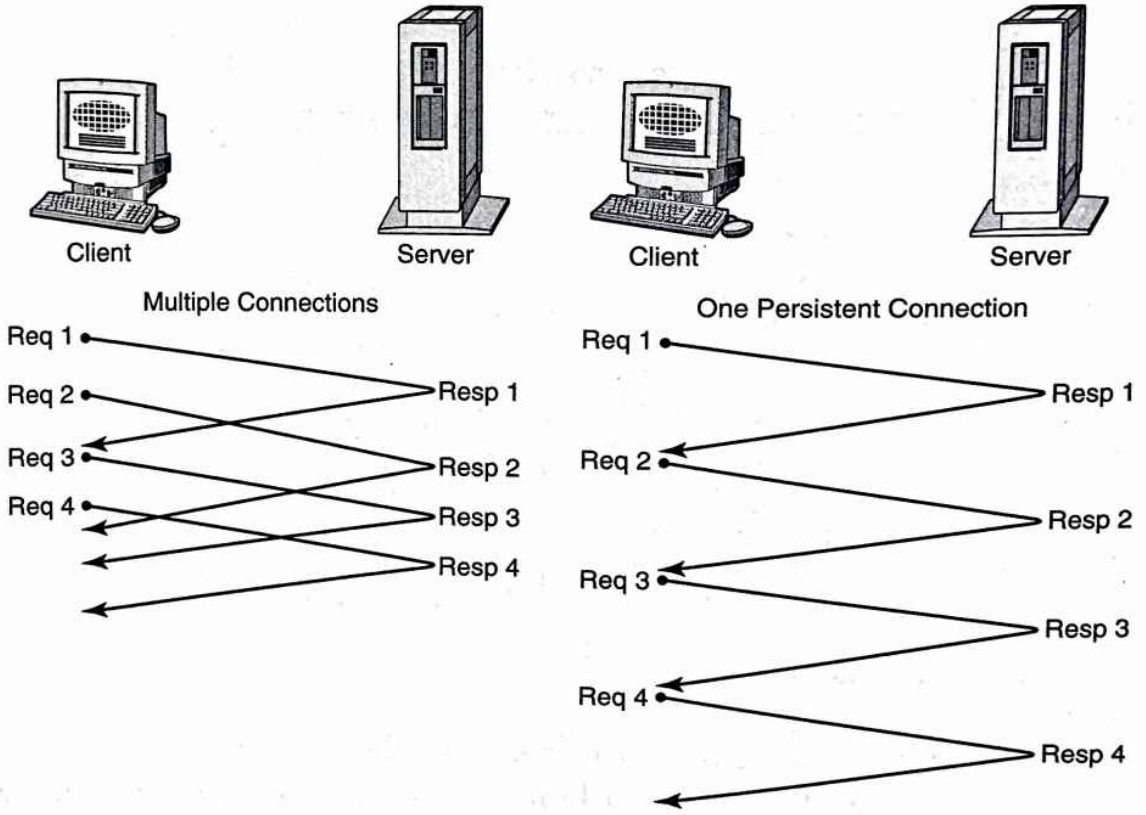


Figure 4.4 Comparison between multiple simultaneous connections and a single persistent connection.

Combining the two—that is, having several persistent connections—works somewhat better, as you get the best of both worlds (Figure 4.5). However, the benefit of persistent connections is reduced by the fact that

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 17

there will be fewer requests made over each persistent connection (since there are more connections among which the requests are divided).

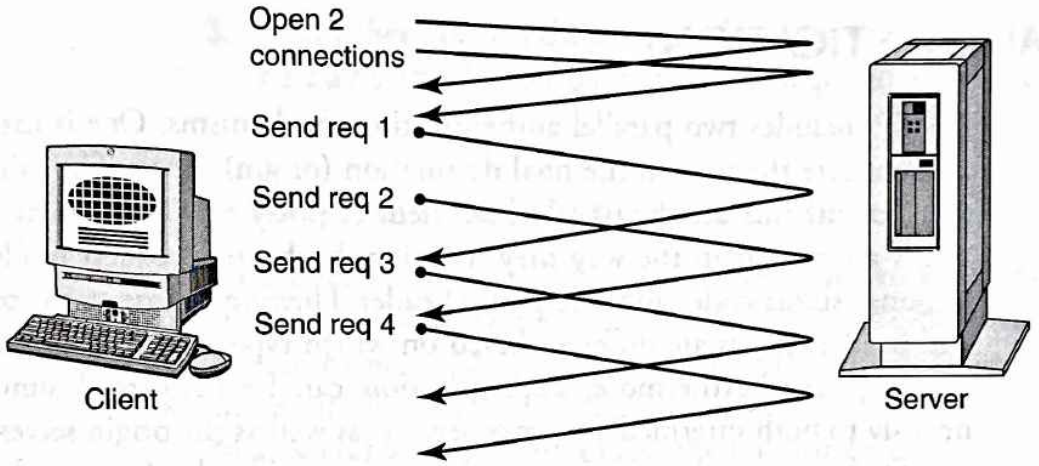


Figure 4.5 Two simultaneous persistent connections.

HTTP/1.1 Request Pipelining

To further reduce the latency, request pipelining was added to the HTTP/1.1 protocol to be used with persistent connections. Pipelining means that the next request is sent over the persistent connection *before* the previous response has been entirely (or at all) received. Figure 4.6 illustrates pipelining.

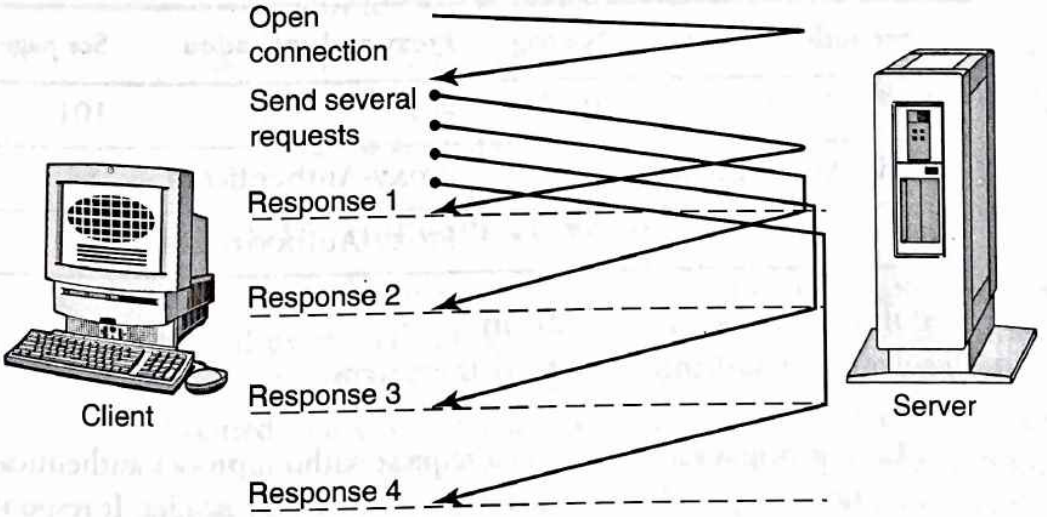


Figure 4.6 HTTP/1.1 request pipelining with persistent connections.

Persistent connections and request pipelining have been studied extensively, and it has been found that persistent connections alone per-

form poorly—that is, if the next request is sent only after the previous one finishes. Pipelining considerably improves performance.

HTTP AUTHENTICATION

HTTP includes two parallel authentication mechanisms. One is used to authenticate the user to the final destination (origin) server. The other is used to authenticate the user to intermediate proxy servers. Both mechanisms are similar in the way they function: both use a request header, a response status code, and a response header. However, those status codes and header names are different based on which type of authentication is taking place. Furthermore, authentication can be performed simultaneously to both intermediate proxy servers, as well as the origin server.

These two authentication mechanisms provide the framework for HTTP authentication. Different *authentication schemes* can be implemented on top of this framework. The HTTP specification defines the "`Basic`" authentication scheme and also mentions another one, "`Digest`" authentication. These are discussed later in this section.

Table 4-1 The correspondence between status codes, request headers, and response headers used by origin server and proxy server authentication.

Server authentication	See page	Proxy authentication	See page
401	101	407	101
WWW-Authenticate:	88	Proxy-Authenticate	88
Authorization:	77	Proxy-Authorization	78

Regular Server Authentication

Regular server authentication takes these steps:

1. The origin server receives a request without proper authentication credentials in the `Authorization:` header. It responds with `401` (Authentication Required) status code, and sends a `WWW-Authenticate:` header to the client, specifying details about how to perform authentication.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 19

2. The client receives the `401` status code, inspects the `WWW-Authenticate:` header, and prompts for the username and password from the user.
3. Client then resends the request, this time with a proper `Authorization:` header containing the required authentication credentials.

Proxy Server Authentication

Proxy server authentication follows the same framework as regular server authentication (challenge-response); however, it uses different status code and header names. Table 4-1 shows the correspondence between these two authentication mechanisms.

Proxy server authentication takes the following steps:

1. Proxy server receives a request without proper proxy authentication credentials in the `Proxy-Authorization:` header field. It responds with the `407` (Proxy Authentication Required) status code, accompanied by a `Proxy-Authenticate:` response header, specifying details about how to perform proxy authentication.
2. The client receives the `407` status code, inspects the `Proxy-Authenticate:` header and prompts for the username and password from the user.
3. Client then resends the request to the proxy, this time with proper `Proxy-Authorization:` header containing the required authentication credentials.

Basic HTTP Authentication (to Server)

The "`Basic`" authentication scheme is the only authentication scheme defined by the HTTP specification. It does not involve encryption but passes the username and password printable-encoded—which is obscured, but effectively in the clear [9]. With HTTP server authentication, the server will respond with the `401` status code, accompanied by the `WWW-Authenticate:` header, which specifies the authentication scheme "`Basic,`" and so-called "realm," which is simply a string that gets displayed to the user as an indication of *who* is asking for authentication:

```
HTTP/1.0 401 Unauthorized
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 20

```
Server: Netscape-Enterprise/3.0
Date: Sat, 26 Apr 1997 06:03:24 GMT
WWW-Authenticate: Basic realm="Demo Server"
```

The client will prompt for the username and password and use base-64 printable encoding (RFC 1421) to encode the string with username and password, separated by a semicolon:

username:*password*$_{printable_encoded}$

This encoded string will be specified in the `Authorization:` header, along with the authentication scheme specifier "`Basic`":

`Authorization: Basic` *username*:*password*$_{printable_encoded}$

The request will be reissued to the server, this time with the authentication credentials in the `Authorization:` header:

```
GET /somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Authorization: Basic ZWFzdGVyOmVnZw==
```

It is a common misunderstanding that the authentication occurs only in the first request and is then "magically" remembered by the server. This is not the case. The client software will cache, or "remember," the username and password and automatically send the `Authorization:` header to the server on subsequent requests. In other words, authentication occurs with *every request.*

Basic HTTP Authentication (to Proxy)

The "`Basic`" authentication scheme can be used for authenticating the user to an intermediate proxy server as well. The authentication challenge will come back from the proxy, with the `407` status and the `Proxy-Authenticate:` header:

```
HTTP/1.0 407 Proxy authentication required
Server: Demo-Proxy/4.0
Date: Wed, 31 Dec 1997 12:00:00 GMT
Proxy-Authenticate: Basic realm="Firewall Proxy"
```

The client will reissue the request, attaching the printable-encoded username and password in the `Proxy-Authorization:` header:

```
GET http://www.somesite.com/somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 21

```
Proxy-authorization: Basic eG1hczpjb29raWU=
```

Note that the origin server may issue an authentication challenge after the proxy authentication. In this case, both authentication credentials will be attached to the request:

```
GET http://www.somesite.com/somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Authorization: Basic ZWFzdGVyOmVnZw==
Proxy-authorization: Basic eG1hczpjb29raWU=
```

The proxy server will use credentials in the `Proxy-Authorization:` header and remove the header from the request. The origin server will use the `Authorization:` header to get its credentials.

Proxy Authentication Limited to One Proxy

The proxy authentication design in HTTP/1.0 and HTTP/1.1 is lacking in one respect: in a chain of proxies the authentication can occur at a single proxy only (see the section on proxy chaining on page 22 for a description). There cannot be multiple instances of the `Proxy-Authorization:` header targeted at different proxies in the chain. There are several remedies for this problem:

- Share the authentication database among all proxy servers, or synchronize the databases periodically. Then configure the inner proxies to propagate authentication credentials to all outer proxies and use a shared authentication database or at least synchronized passwords by all proxy servers.
- Require authentication only at the first proxy; protect upstream proxy servers by other means, such as IP address-based access control; accept requests only from the internal proxy server IP addresses.
- Require authentication only at the outermost proxy servers. With this solution, the inner proxy server will not log the usernames. Also, all requests satisfied from the inner proxy's cache can be received without any authorization. Only when the request is forwarded to the outer proxy will the user have to enter his or her username and password.

These alternatives are discussed in more detail below.

Propagating Proxy Authentication Credentials

Normally, once authentication has occurred in a proxy, the proxy will strip out the `Proxy-Authorization:` header from the request when it forwards it along to the next proxy in the proxy chain. This is to prevent the disclosure of the user's authentication credentials to a proxy (or server) that does not need to have them.

However, the specification leaves it open for proxy servers to, at their discretion, forward the authentication credentials to the next proxy in the chain even if the current proxy has already used them for authentication. If the proxies share the same user database, or at least have the same password for each given user, authentication can be performed by all proxies in the chain.

Some proxy servers support this feature of forwarding proxy authentication credentials that have already been *consumed* by that proxy. This allows the same authentication credentials to be available to all proxies in the proxy chain; however, the requirement is that user databases be shared, or at least username/password pairs be synchronized.

However, all proxies should remove the `Proxy-Authorization:` header when the request is forwarded to an origin server. An origin server, even if it is in actuality a reverse proxy server, never has any legitimate use for proxy authentication credentials. See the section on capturing authentication credentials on page 271 for a description of potential security holes that this could open.

Require Authentication at the First Proxy Only

In this scenario, authentication is set up only on the first proxy. Proxies further upstream from those front-line proxies are configured so that they only accept requests from those proxies, never directly from client programs. This can be done by a combination of the following means:

- router setup; only allow connections to the outer proxy host and port from the authorized inner proxy server hosts
- proxy access control by hostname/IP address; allow the proxy to only accept requests from the authorized inner proxy server hosts
- proxy header validation; verify that the `Via:` header indicates that the request was really forwarded by an authorized inner proxy server
- use secure communication between proxies; some proxy servers support secure communication channels between proxy servers;

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 23

using this feature it's possible to authenticate the connecting proxy server, as well as encrypt the actual data being transferred between chained proxies. See the section on other load balancing solutions on page 324 for a description of secure communication between proxy servers.

Require Authentication at the Outermost Proxy Only

In this scenario, none of the internal proxy servers require authentication. They may enforce IP address- or hostname-based access control, but they do not authenticate the users (at least not via the `Proxy-Authorization:` mechanism).

Authentication is required by only the outermost proxies in the proxy chain. This would typically be at the firewall proxy, that is, upon making a request that requires a connection that leaves the intranet and goes out to the Internet.

Note that there may be external pages from the Internet already cached on the inner proxy servers so that a given user may browse some Internet pages even without first authenticating him or herself.

Proposed Protocol Modification

At the time of writing this book (summer 1997), there is an initiative to get the proxy authentication limitation resolved in the HTTP protocol level. This change would allow multiple authentication credentials to be passed to all intermediate proxy servers, if required. Once standardized, this feature will surely be supported by various proxy servers and client software.

VIRTUAL SERVERS

In HTTP/1.0, there is a slight difference in the protocol between clients and origin servers, as opposed to clients and proxy servers (or between proxies and other proxies). The requested URL is a *full URL* when requested from a proxy server, but only a *partial URL* when requested from an HTTP/1.0 server.

A typical request sent to a proxy server looks like this:

```
GET http://home.netscape.com/people/ari/ HTTP/1.0
User-agent: Mozilla/3.0
Accept: text/html, image/gif, image/jpeg
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 24

while an origin server would see only the *path part* of the URL:

```
GET /people/ari/ HTTP/1.0
User-agent: Mozilla/3.0
Accept:text/html, image/gif, image/jpeg
```

Originally, leaving out the protocol, host, and the optional port part of the URL from the request seemed like a harmless thing to, while saving a few bytes. After all, the origin server implicitly knows its hostname, port number, and the protocol that it speaks (the `http://` prefix).

However, this did not remain true when the Web technology became famous. A lot of companies and individuals wanted to have a presence in the Internet with a form of a dedicated Web server address (`http://www.`*sitename*`.com` instead of `http://www.`*hostsite*`.com/`*sitename*), but they couldn't afford, or didn't want, to spend money on a dedicated Web server hardware [10]. Many entities actually preferred that this host be maintained by a third party *service provider*.

Service providers, on the other hand, didn't want to dedicate an entire server host for each home page URL, either. Instead, they would give multiple DNS aliases for the same server machine. However, the problem is that regardless of the DNS alias used to resolve the IP address, the IP address would always be the same. Because the HTTP request didn't include the hostname used to yield the IP address, the server software could not determine which DNS alias was used, and which home page to send to a client requesting a URL such as `http://www.`*somesite*`.com`.

Multiple Network Interfaces

The short-term solution, although not a viable one for large-scale deployment, was to actually allocate different IP addresses for each DNS name and then run multiple instances of the Web server, one for each IP address. However, this is wasteful of machine resources, and especially of the IP address space, which is already running out due to the explosive growth of the Internet.

The "Host:" Header

As a longer-term solution, the `Host:` request header was added to the HTTP protocol as an extension to HTTP/1.0. This header carries the hostname used to yield the IP address [11]. For example, to retrieve the URL

`http://www.`*somesite*`.com/test.html`

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 25

the following HTTP/1.0 request would be sent to the server www.*somesite*.com:

```
GET /test.html HTTP/1.0
Host: www.somesite.com
User-agent: Mozilla/3.0
Accept: text/html, image/gif, image/jpeg
Accept-language: en
```

Now the server can look up the Host: header, and based on that serve a completely different document tree that is specific to the server http://www.*somesite*.com/.

Example. Let's illustrate the purpose of the Host: header with an example where two Web servers for different companies are hosted on the same physical server machine. The machine has aliases www.company-a.com and www.company-b.com. Each of these companies would like to advertise their Web home page with the URLs

```
http://www.company-a.com/
```

and

```
http://www.company-b.com/
```

However, in HTTP/1.0, the request received by the Web server will look the same regardless of which one of the URLs was used to access the server:

```
GET / HTTP/1.0 ...
```

The introduction of the Host: header removes this problem by reporting which hostname (alias) was used.

```
GET / HTTP/1.0
Host: www.company-a.com
...
```

The Web server software will inspect the contents of the Host: header and determine which document tree to use for this request.

Full URLs in Requests

The truly best long-term solution is to start using the full URL in all requests, including those sent to the origin server. However, this change is not backward compatible, so this transition must be done gradually. HTTP/1.1 already requires origin servers to also understand full URLs. The clients may make HTTP/1.1 requests with the full URL; however, this is not a requirement, since it would not work if the server is an old HTTP/1.0 server. In the future, when HTTP/1.1 (and HTTP/2.0) serv-

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 26

ers and clients become the majority, clients might transition to using full URLs for all requests. However, this is still a far away dream—it's a fact that HTTP/1.0 servers will stay around for a long time, even when newer versions become available. Not everybody will get around to upgrading their servers if they keep working well using the older protocol—and there seems to be no reason why they wouldn't.

META HTTP-EQUIV

HTML (HyperText Markup Language) provides a feature to set HTTP response headers from within the HTML document. This is an easy way for the document author to set, for example, the expiration time of the document, or to utilize the "automatic refresh feature" of some client programs which can cause a different URL to be loaded automatically after a specified amount of time.

This feature normally works via the HTTP response header. For example,

```
Refresh: 5; http://home.netscape.com/people/ari/nextpage.html
```

will cause the current document to be displayed for five seconds and then cause the URL specified in the `Refresh:` header to be automatically loaded.

The same effect can be accomplished by use of the `META` tag inside the actual HTML document. The above example could be expressed inside the HTML file as

```
<META HTTP-EQUIV="Refresh"
CONTENT="5; URL=http://home.netscape.com/people/ari/nextpage.html">
```

Note that the `<META>` tag must appear in the head section of the HTML document—that is, between `<HEAD>...</HEAD>`, as opposed to inside `<BODY>...</BODY>` where the actual HTML text of the document is.

However, the `META` tag is mainly a client-side feature. Most proxies, in fact, do not parse the HTML for `META` (or any other) tags. This would be an unnecessary performance hit. The `META` tag is mostly just a convenience feature for HTML authors to store metadata of the document that *should* really be sent as HTTP response headers to the client.

In this author's opinion, rather than requiring that the proxy take the performance hit of scanning for META tags, it should be performed by the origin server. After all, it has a greater ability to simply scan for those tags once, each time the document is modified and maybe cache the

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 27

found headers in a separate cache file. This way all the proxy servers don't have to repeat this step.

In any case, merely client-side features—such as the above automatic refresh feature—can still be embedded within the HTML document without any regard to proxies. However, headers that should affect proxy servers should be set in the HTTP header rather that embedded within the HTML document as `META` tags to guarantee that the proxy sees it. Web server software usually provides a mechanism for setting these headers.

MIME MEDIA TYPES

The type of Web objects is indicated by a MIME media type. HTTP transmits the media type of the document—whether it is an HTML document, a text file, an image, audio, or video clip, or application-specific data—in the `Content-Type:` header (page 94).

The media type specification has two parts: the main type and subtype. The main type specifies the overall category of the object; the subtype gives a more accurate type specification. For example, HTML documents have the media type "`text/html`"—the main type is "`text`", and the subtype is "`html`."

The main media type categories are

`application/*` application-specific data that cannot be categorized under any other main type category; for example,

- `application/octet-stream` binary data that does not have any better, more specific type
- `application/pdf` PDF (Adobe's Portable Document Format)

`audio/*` audio data; for example,

- `audio/basic`
- `audio/wav`

`image/*` image data; for example,

- `image/gif` image in the GIF format
- `image/jpeg` image in the JPEG format

`message/*` for example,

- `message/http`, an HTTP message
- `message/rfc822`, RFC 822 style message

`model/*` for example,

- `model/vrml`, virtual reality modeling language format

`multipart/*` multipart MIME message, enclosing several separate MIME messages; for example,

- `multipart/mixed`, mixed data types
- `multipart/form-data`, an HTML form submission using a multipart MIME format

`text/*` text data; for example,

- `text/html`, HTML document
- `text/plain`, plain text document

`video/*` video data; for example,

- `video/mpeg`
- `video/quicktime`

MIME types may also have parameters attached to them in the `Content-Type:` header. These parameters give further details about the format, such as its format version, or the character set used by the document:

```
text/html; charset=ISO-8859-4
```

HTTP REQUEST METHODS

The first word in the HTTP request is the *method*, which indicates the action of the request. Table 4-2 lists the common methods defined by the HTTP/1.1 specification. However, new experimental methods may be freely added, and proxy servers should be able to deal with them (see the section on Unknown Method Tunneling on page 138). In fact, there are HTTP/1.0 servers that support the PUT method, as well as a number of other methods not in this list at all.

Table 4-2 Common methods defined by HTTP. Support in the different HTTP versions is indicated.

Method	0.9	1.0	1.1	Description
GET	•	•	•	Retrieve a resource
HEAD		•	•	Retrieve metadata
POST		•	•	Form submission
PUT			•	Upload file
DELETE			•	Delete a resource

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 29

Table 4-2 Common methods defined by HTTP. Support in the different HTTP versions is indicated. *(Continued)*

Method	0.9	1.0	1.1	Description
TRACE			•	Trace a proxy chain
OPTIONS			•	Query server options

The GET Method

The GET method is the most commonly used HTTP method: it is used to retrieve a single resource—whether an HTML file, image file, Java applet, or any other type of object, or part of it. The GET method can be *conditional*, that is, an additional condition may be attached to the request, and the response is determined on the condition. This allows efficient cache up-to-date checks. Conditional GET is discussed in the section on conditional requests on page 158.

The URL may point to a static file on the origin server. In this case, the contents of the file are simply sent to the requesting client. Objects that originate from a static file are ideal for caching—basically, the thinking is that if the object can be stored in a file on the server, it can be stored in a cache file as well.

The URL may also point to dynamic content, such as server-parsed HTML, a CGI script, or other kind of dynamic server application. In this case, the origin server will execute the necessary instructions in order to produce the dynamic content. Contrary to common misconception, CGI scripts cannot be transferred in source code form to a proxy server and be executed there; the origin server *always* executes the dynamic application code and produces the resulting document, which is then passed to the proxy and cached there if the type of dynamic application allows for that.

The GET method is expected to have *no side effects*, that is, it should be safe to reissue the GET request without changing the state of the server. In practice, some GET requests do cause side effects that are not recommended by the HTTP specification but for historical reasons they are not illegal. For example, HTML form field values originally used to be submitted with the GET method before the POST method was widely supported by server software. Even though POST is currently the predominant way of submitting forms, use of GET is still legal, and in some cases, appropriate. Namely, in cases where the query is short, and the

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 30

execution of it will not have any side effects, it is justified to use the GET method. Furthermore, this allows the results to be cached by proxy servers.

Visitor Counters

As an example of side effects with the GET method, many sites have a "visitor counter" image URL which returns a GIF image representing how many visitors have accessed the site. Each access to that URL will cause the counter to be incremented on the origin server, and an image representing the current value of the counter to be passed back. Obviously, reretrieving this type of visitor counter URL will cause a side effect of incrementing the counter by one every time, unless of course the counter is sophisticated enough to use HTTP cookies, or check the requesting IP address first. However, relying on IP addresses to count individual visitors is an extremely inaccurate method because of proxy servers—a single proxy server (represented by a single IP address) can service hundreds or thousands of users. The issue of counting individual user accesses is discussed in depth in Chapter 9.

The HEAD Method

The HEAD method acts like the GET method, except that it returns only the HTTP response and entity headers, but not the actual content of the document. This method is useful for finding out the size, type, or other attributes of an object, without actually retrieving the object itself.

The POST Method

The POST method is for submitting HTML forms, annotating existing resources, posting messages and articles, and extending databases. Unlike the GET request, POST can cause permanent (side) effects on the server, and it is not allowed to re-perform a POST request automatically without user approval to refresh the content of the document view. Responses resulting from a POST operation are not cacheable, unless otherwise explicitly stated by `Cache-Control:` or `Expires:` headers.

The data being posted is sent in the entity section of the request. In the case of HTML forms, this means that the encoded name-value pairs are not appended to the URL as with the GET method but instead placed in the body part of the POST request.

The URL in POST requests refers to the data handling process of the posted data (for form submissions), or an association between that URL and the posted data (in case of annotations, messages to mailing lists, or

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 31

articles to bulletin boards, in which case the URL refers to the annotated document, the mailing list, or the bulletin board, respectively).

The PUT Method

The PUT method is used modify existing resources and to create new ones. Responses to PUT requests are not cacheable. However, if the URL itself has been cached by the proxy through which the PUT request is being made, the cached copy should be marked as stale by that proxy server since it's apparent that the resource was modified by the PUT operation.

The URL in PUT requests refers to the resource being created or modified. A PUT request can be made conditional like the GET method. This is useful when making sure that the modified version is derived from the version currently on the server. This help prevent overwriting someone else's changes.

The DELETE Method

The DELETE method is used to delete existing resources. The deletion of a resource does not necessarily take effect immediately. The server may respond with a 202 Accepted status code indicating that the request to delete has been received but not yet carried out.

The OPTIONS Method

The OPTIONS method allows requests for information about the proxy and origin server chain's capabilities and communication options available for a particular URL. The OPTIONS request has two alternative forms; with the URL specified as an asterisk

```
OPTIONS * HTTP/1.1
User-agent: Mozilla/4.0
```

the request applies to overall features and options available in the server. With a specific URL

```
OPTIONS URL HTTP/1.1
User-agent: Mozilla/4.0
```

the request applies to the specified URL only.

The response to an OPTIONS request includes applicable response headers describing available options, such as the Allow: and Public: headers.

The TRACE Method

The TRACE method allows for tracing the requests in proxy chains. The response to a TRACE response is generated by the last server in the server chain (either origin server, or an intermediate proxy server if the `Max-Forwards:` header is used) and is simply the request as received by that server. From the trace response it is possible to see the different hops the request made through the proxies by examining the `Via:` headers.

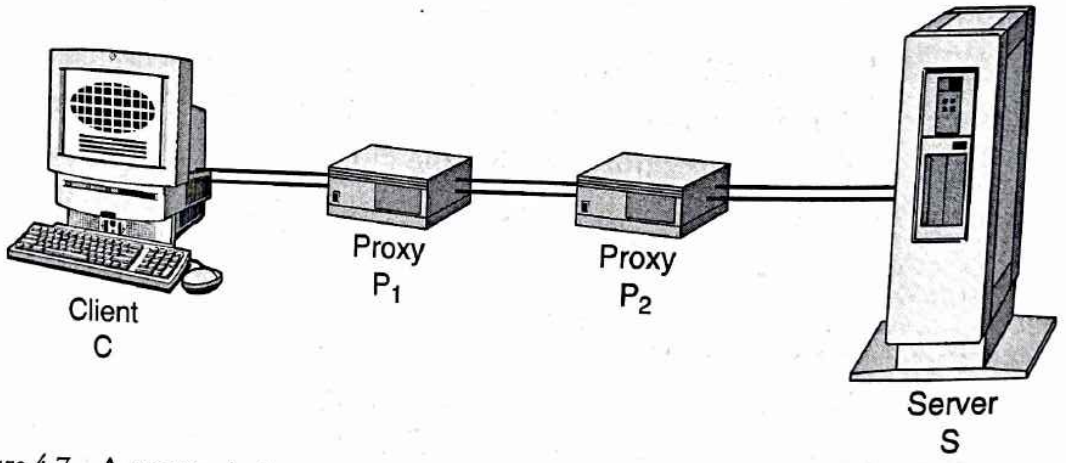


Figure 4.7 A proxy chain with two proxy servers.

As an example, let's assume we have a requesting client *C*, a chain of two proxy servers, P_1 and P_2, and an origin server *S* (see Figure 4.7). The client issues a TRACE request to the inner proxy server P_1:

```
TRACE http://S/ HTTP/1.1
User-Agent: Mozilla/4.0
```

The inner proxy server P_1 will forward the request to the outer proxy server P_2, adding the `Via:` header:

```
TRACE http://S/ HTTP/1.1
User-Agent: Mozilla/4.0
Via: 1.1 P1
```

which in its turn will forward the request to the destination origin server *S*:

```
TRACE / HTTP/1.1
User-Agent: Mozilla/4.0
Via: 1.1 P1, 1.1 P2
```

The origin server will generate the response, enclosing the received request as the entity in the response:

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 33

```
HTTP/1.1 200 Ok
Server: Netscape-Enterprise/4.0
Date: Sun, 11 May 1997 09:30:37 GMT
Content-type: message/http

TRACE / HTTP/1.1
User-Agent: Mozilla/4.0
Via: 1.1 P1, 1.1 P2
```

HTTP HEADERS

HTTP headers are used to include additional information to requests and responses. The HTTP/1.1 specification defines a total of 46 headers. They are divided into four categories: general headers, request headers, response headers, and entity headers. General headers may exist in both requests and responses. Request and response headers are specific to only requests and responses, respectively. Entity headers describe the content of the request body (if any), or the content of the response body (if any). Requests have the following overall structure:

```
METHOD URL HTTP/version
... General headers...
... Request headers headers...
... Entity headers (optional)...
                                ⇐ empty line
... Request entity (if any)...
```

Responses have a similar structure:

```
HTTP/version status-code reason-line
... General headers...
... Response headers headers...
... Entity headers (optional)...
                                ⇐ empty line
... Resource entity (if any)...
```

In addition to the standard headers defined by the specification, applications may add their own, experimental headers when necessary. This allows HTTP to be easily extensible for new applications.

Table 4-3 lists alphabetically all header fields defined by the HTTP/1.1 specification, their type, and the page on which they are described.

Table 4-3 HTTP headers. Headers are categorized to general, request, response, and entity headers. The page where the header is defined is included.

Header name	General	Request	Responce	Entity	Page
Accept:		•			76
Accept-Charset:		•			76
Accept-Encoding:		•			77
Accept-Language:		•			77
Accept-Ranges:			•		86
Age:			•		86
Allow:				•	91
Authorization:		•			77
Cache-Control:	•				72
Connection:	•				72
Content-Base:				•	91
Content-Encoding:				•	92
Content-Language:				•	93
Content-Length:				•	93
Content-Location:				•	93
Content-MD5:				•	94
Content-Range:				•	94
Content-Type				•	94
Date:	•			•	73

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 35

Table 4-3 HTTP headers. Headers are categorized to general, request, response, and entity headers. The page where the header is defined is included. *(Continued)*

Header name	General	Request	Responce	Entity	Page
ETag:				•	95
Expires:				•	95
From:		•			78
Host:		•			78
If-Modified-Since:		•			78
If-Match:		•			79
If-None-Match:		•			81
If-Range:		•			83
If-Unmodified-Since:		•			79
Last-Modified:				•	95
Location:			•		86
Max-Forwards:		•			83
Pragma:	*				74
Proxy-Authenticate:			•		88
Proxy-Authorization:		•			78
Public:			•		87
Range:		•			84
Referer:		•			85
Retry-After:			•		88
Server:			•		87

Table 4-3 HTTP headers. Headers are categorized to general, request, response, and entity headers. The page where the header is defined is included. *(Continued)*

Header name	General	Request	Responce	Entity	Page
Transfer-Encoding:	•				74
Upgrade:	•				74
User-Agent:		•			85
Vary:			•		88
Via:	•				75
Warning:			•		90
WWW-Authenticate:			•		88

General Headers

General headers are headers that can be present in either requests or responses. This section briefly describes each of the general headers defined by the HTTP/1.1 specification.

The Cache-Control: General Header

The `Cache-Control:` header is used to control various aspects of caching. This header plays such an important role in HTTP/1.1 caching that it is covered in its own section, HTTP/1.1 Cache Control on page 176.

The "Connection:" General Header

The `Connection:` header is used to specify communication options for the connection. As an example, in HTTP/1.1 *persistent connections* (see page 47) are the default; that is, the connection will remain open after the response has been sent so that the client can reuse the same connection for the next request. To override this default behavior and cause the connection to be closed after the response has been sent and received, the "`close`" specifier can be sent in the `Connection:` header:

```
Connection: close
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 37

In a more general sense, the `Connection:` header is intended to specify which headers are specific to the connection and must not be forwarded to the next server—unless the proxy is aware of the feature and would like to issue a similar header itself for the next connection.

As an example, let's say that at a later time an experimental feature called "connection multiplexing" is introduced [12]. The connection-specific parameters for this feature are passed in the `Multiplex-Config:` header. This header is intended to be connection specific and is not to be forwarded to the next proxy (or origin) server—therefore the `Connection:` header is used to specify this header name:

```
GET http://www.somesite.com/ HTTP/1.0
Multiplex-config: some options
Connection: multiplex-config
...
```

As another example, let's take the HTTP/1.0 compatibility header `Keep-Alive:` which was the older mechanism for establishing persistent connections. This header, if present, specifies the number of seconds that the connection will remain open waiting for the next request—after which time the connection will be closed if no request arrives. In HTTP/1.1, this header would be "protected" by the `Connection:` header, to make it *hop-by-hop* [13]:

```
GET http://www.somesite.com/ HTTP/1.0
Keep-alive: some options
Connection: keep-alive
...
```

The "Date:" General Header

The `Date:` header indicates the date and time at which the message was generated. In requests it is the time that the client generated the request. If the request is *initiated* by a proxy server without a client request (e.g., with pull mode caching), the proxy will set the `Date:` header. Otherwise, the proxy will forward the `Date:` header generated by the client. In responses the `Date:` header represents the time that the origin server generated the response—a cached copy will carry the `Date:` header generated by the origin server. Responses from proxy servers will therefore include the original `Date:` header, so it can be used to determine the cached copy's age [14].

The `Date:` header has several legal formats, but the preferred format is the RFC 822 [15] date format:

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 38

```
Date: Sun, 09 May 1997 19:40:17 GMT
```

Other legal formats are listed below; in practice older servers may send these date formats, but newer software should never generate them:

```
Date: Sunday, 09-May-97 19:40:17 GMT
Date: Sun May 9 19:40:17 1997 GMT
```

The "Pragma:" General Header

The `Pragma:` header passes special directives in requests and responses. However, this header is being phased out in favor of the `Cache-Control:` header. It should no longer be used by new implementations, but older implementations may still send it.

The HTTP/1.1 specification defines only one such pragma directive:

```
Pragma: no-cache
```

In requests, the `no-cache` pragma indicates that all intermediate proxy servers should forward the request all the way to the origin server, even if they have an apparently fresh copy in their cache. In practice, this directive is linked to the `Reload` button in clients—pressing this button will force the request to be passed all the way to the origin server and guarantees that the document is up-to-date. The preferred alternative for this pragma is

```
Cache-Control: no-cache
```

In responses, the `no-cache` pragma indicates that the response should not be cached. The aforementioned `Cache-Control: no-cache` header is a preferred alternative.

The "Transfer-Encoding:" General Header

The `Transfer-Encoding:` header indicates any transformations that have been performed on the message. The only transfer encoding specified by the HTTP/1.1 specification is chunked encoding:

```
Transfer-encoding: chunked
```

The "Upgrade:" General Header

The `Upgrade:` header is intended for switching the protocol, or the protocol version, on-the-fly. This allows smooth migration from the HTTP/1.1 protocol to future protocols, such as HTTP/2.0. The client may include this header in the request advertising protocols and protocol versions that it is capable of supporting and would prefer to switch to:

```
Upgrade: HTTP/2.0
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 39

The "Via:" General Header

The `Via:` header indicates the proxy chain that the request was passed through. The format is

```
Via: protocol pseudonym
```

where *protocol* is the protocol name (optional if HTTP) and version of the received request. The *pseudonym* value is the hostname, or a symbolic name of the proxy server. For security reasons, it may often be undesirable to disclose the intermediate proxy server hostnames—especially in firewall environments. As a remedy, a pseudonym can be used instead. A comment may be enclosed in parentheses at the end; oftentimes, the proxy server software name and version is included as a comment:

```
Via: HTTP/1.1 myproxy (Demo-Proxy/4.0)
```

The protocol name is optional if it is HTTP; in that case simply the HTTP protocol version number is used:

```
Via: 1.1 myproxy (Demo-Proxy/4.0)
```

In a proxy chain, each proxy adds its own entry in the end, separated by a comma:

```
Via: 1.1 first-proxy, 1.1 second-proxy
```

In HTTP/1.0 where no `Via:` header was defined, the `User-Agent:` field was used by some proxy servers to indicate intermediate proxy servers by appending the proxy server information after the client software string:

```
User-Agent: Mozilla/3.0 via proxy gateway CERN-HTTPD/3.0 libwww/2.17
```

Also, some intermediate proxy servers use the `Forwarded:` header which was an experimental feature but was never included in the specification in favor of the more compact `Via:` header:

```
Forwarded: by http://proxy-host:port (Demo-Proxy/2.5)
```

Request Headers

Request headers are headers that are meaningful only in requests. They include additional information about the request and may act as request modifiers. This section describes the request headers defined by the HTTP/1.1 specification.

The "Accept:" Request Header

The `Accept:` header specifies what media types are acceptable to the requesting client. For example, a text-mode-only client [16] might send a header specifying that only HTML and plain text are acceptable:

```
Accept: text/html, text/plain
```

whereas graphical user interface clients might send

```
Accept: text/html, text/plain, image/gif, image/jpeg
```

The asterisk wildcard character can be used to specify a whole group of media types, for example all text and image types:

```
Accept: text/*, image/*
```

Furthermore, to indicate that all formats are acceptable the client may specify

```
Accept: */*
```

The client may also use the *quality parameter* `q` to specify the preference of media types. The value of `q` is between 0 (not preferred) and 1 (preferred):

```
Accept: text/html; q=1, image/gif; q=1, text/*; q=0.5, */*; q=0.1
```

The above would give preference to HTML text files and GIF and JPEG images (quality 1, highest); intermediate preference to any other text files; and low preference for all others ("use only if no other more preferable format is available").

The default value for the `q` parameter is 1, so `q=1` can be left out from the above example:

```
Accept: text/html, image/gif, image/jpeg, text/*; q=0.5, */*; q=0.1
```

Note that quality parameters are separated by a semicolon "`;`" from the media type, and media types are separated by a comma "`,`" from each other.

The actual media type of the object sent in the response is specified by the `Content-Type:` header (see page 94).

The "Accept-Charset:" Request Header

The `Accept-Charset:` header is used to specify acceptable character sets. By default, all available character sets are considered acceptable; specifying this header will narrow down the acceptable character sets:

```
Accept-charset: iso-8859-5
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 41

The `Accept-Charset:` header utilizes the quality parameter `q` in the same way as the `Accept:` header (see previous section). The actual character set used is specified using the `charset` parameter (if present) in the `Content-Type:` header of the response:

```
Content-type: text/html; charset=ISO-8859-4
```

The "Accept-Encoding:" Request Header

The `Accept-Encoding:` header specifies the acceptable encodings that the server may use. Example:

```
Accept-encoding: compress, gzip
```

The actual encoding used, if any, is returned in the `Content-Encoding:` response header (see page 92).

The "Accept-Language:" Request Header

The `Accept-Language:` request header is used to specify language preferences of the user. The quality parameter `q` can be used just as with the `Accept:` header. Example:

```
Accept-Language: en, fr=0.5
```

This would give preference to English but also accept French.

The "Authorization:" Request Header

The `Authorization:` header is used to pass user's authentication credentials to the origin server. The `401 Unauthorized` status code, together with the `WWW-Authenticate:` response header, is used to challenge user's authentication credentials. The section on page 54 discusses HTTP server authentication.

The existence of the `Authorization:` request header suggests to intermediate proxy servers that the content is protected and should not be cached. Otherwise, proxy servers might serve a cached document to an unauthorized user. However, proxy servers may cache the document if they force an up-to-date check on every request, which forces the user to authenticate to the origin server. This feature is enabled by either one of the following response headers from the origin server:

```
Cache-control: proxy-revalidate
Cache-control: must-revalidate
```

Also, if the document is explicitly marked public, it may be cached by the proxy and served directly without forcing an up-to-date check every time to force authentication:

```
Cache-control: public
```

See the section on HTTP/1.1 Cache Control on page 176 for a full description of the `Cache-Control:` header.

The "Proxy-Authorization:" Request Header

The `Proxy-Authorization:` header is used to pass the user's authentication credentials to a proxy server. The `407 Proxy authorization required` status code, together with the `Proxy-Authenticate:` response header, is used to challenge the user's proxy authentication credentials. The section on Proxy Server Authentication on page 55 discusses proxy authentication.

The "From:" Request Header

The `From:` request header contains the requesting user's E-mail address. However, for privacy reasons, this header is rarely present in requests.

```
From: ari@netscape.com
```

In the past the automatic generation of this header by client software caused an uproar among champions of privacy—and, as a result, client software no longer automatically sends this header field.

The "Host:" Request Header

The `Host:` header specifies the hostname and port number present in the URL being requested. This addresses the problem with virtual multihosting in HTTP/1.0 where it was not possible to distinguish which hostname alias was used to determine the IP address of the HTTP server. Virtual multihosting is discussed in the section on virtual servers on page 59.

The "If-Modified-Since:" Request Header

The `If-Modified-Since:` request header is used with cache up-to-date checks to perform conditional `GET` requests. See the section on conditional requests on page 158 for an in-depth description of the use of this header.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 43

The "If-Unmodified-Since:" Request Header

The `If-Unmodified-Since:` header is used to make the request conditional, so that the operation is carried only if the resource has *not* been modified since the specified date and time. Example:

```
If-unmodified-since: Sun, 11 May 1997 09:30:37 GMT
```

This feature can be used when using the PUT method to update a new version of a resource. The operation succeeds only if no one else has modified the resource since it was retrieved by the client that modified it. Otherwise there would be a risk of overwriting someone else's changes.

Another use for this header is with byte range requests where the requesting client or proxy server wants to make sure that the new received byte ranges are derived from the same base version as other byte ranges already held. Otherwise, mixed byte ranges would be present and results would be inconsistent and possibly corrupt. The `If-Range:` header can be used for similar purposes; see the section on Request Headers on page 75.

The "If-Match:" Request Header

The `If-Match:` header can be used to perform conditional requests. It is an alternative for the `If-Modified-Since:` header that can be used to quote the value of the `Last-Modified:` header of the cached object. The `If-Match:` header quotes the entity tag of the object as specified by the `ETag:` entity header. For example, if an earlier received object had a header

```
ETag: "doc-id-2441"
```

a request could be issued with the request header

```
If-match: "doc-id-2441"
```

If this *precondition* is not met, the server will respond with the status code

```
412 Precondition failed
```

The `If-Match:` header may list any number of entity tags, in which case the precondition is true if any of the entity tags matches that of the request target object:

```
If-match: "etag-1", "etag-2", "etag-3"
```

This feature is especially useful when performing an update on the server, that is, using method PUT to store a new, modified version of the document. By enclosing the entity tag in the `If-Match:` precondition,

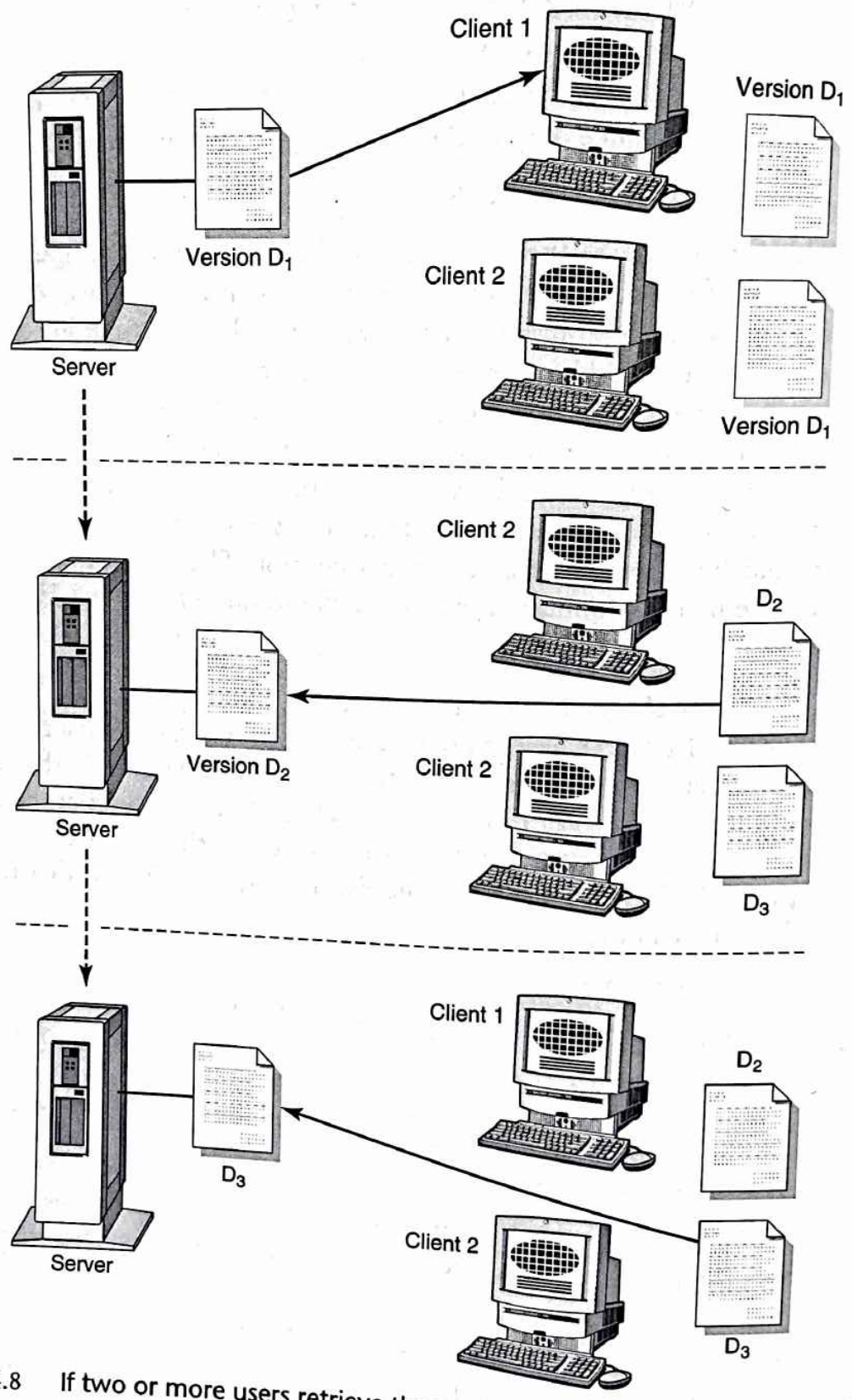


Figure 4.8 If two or more users retrieve the version 1 of document D, denoted by D_1, and both make changes, there is a risk of D_3 overwriting changes made in D_2.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 45

the PUT operation will succeed only if the document has not already been modified by somebody else (which would have caused the entity tag to be changed as well). This way other people's changes do not get discarded accidentally by overwriting the object with a version that is derived from an older version than the one currently available on the server. Figure 4.8 depicts this problem, and Figure 4.9 shows how entity tags help prevent it.

The `If-Match:` header has a special form

```
If-match: *
```

which means "if any representation exists"; that is, including this in the request headers makes the precondition such that the operation will succeed only if the object representation already exists. If the target object representation does not exist, the precondition will fail and the error status code `412 Precondition failed` will be returned.

The "If-None-Match:" Request Header

The `If-None-Match:` header is the inverse of the `If-Match:` header. While an `If-Match:` precondition is true if any of the entity tags matches, an `If-None-Match:` precondition is true if none of the listed entity tags match. The syntax is the same as for `If-Match:`, and any number of entity tags may be listed:

```
If-none-match: "etag-1", "etag-2", "etag-3"
```

This feature is useful for performing up-to-date checks on cached objects. As opposed to `If-Match:` which causes the error code `412 Precondition failed` on failure, the `If-None-Match:` header returns status code `304 Not modified` instead. The typical use is similar to the way the `If-Modified-Since:` header is used in conditional GET requests. Let's say the cache has an object with the entity header

```
ETag: "doc-id-2441"
```

When performing an up-to-date check, the header

```
If-none-match: "doc-id-2441"
```

is included in the request. If the document has not changed, the response will have status code:

```
304 Not modified
```

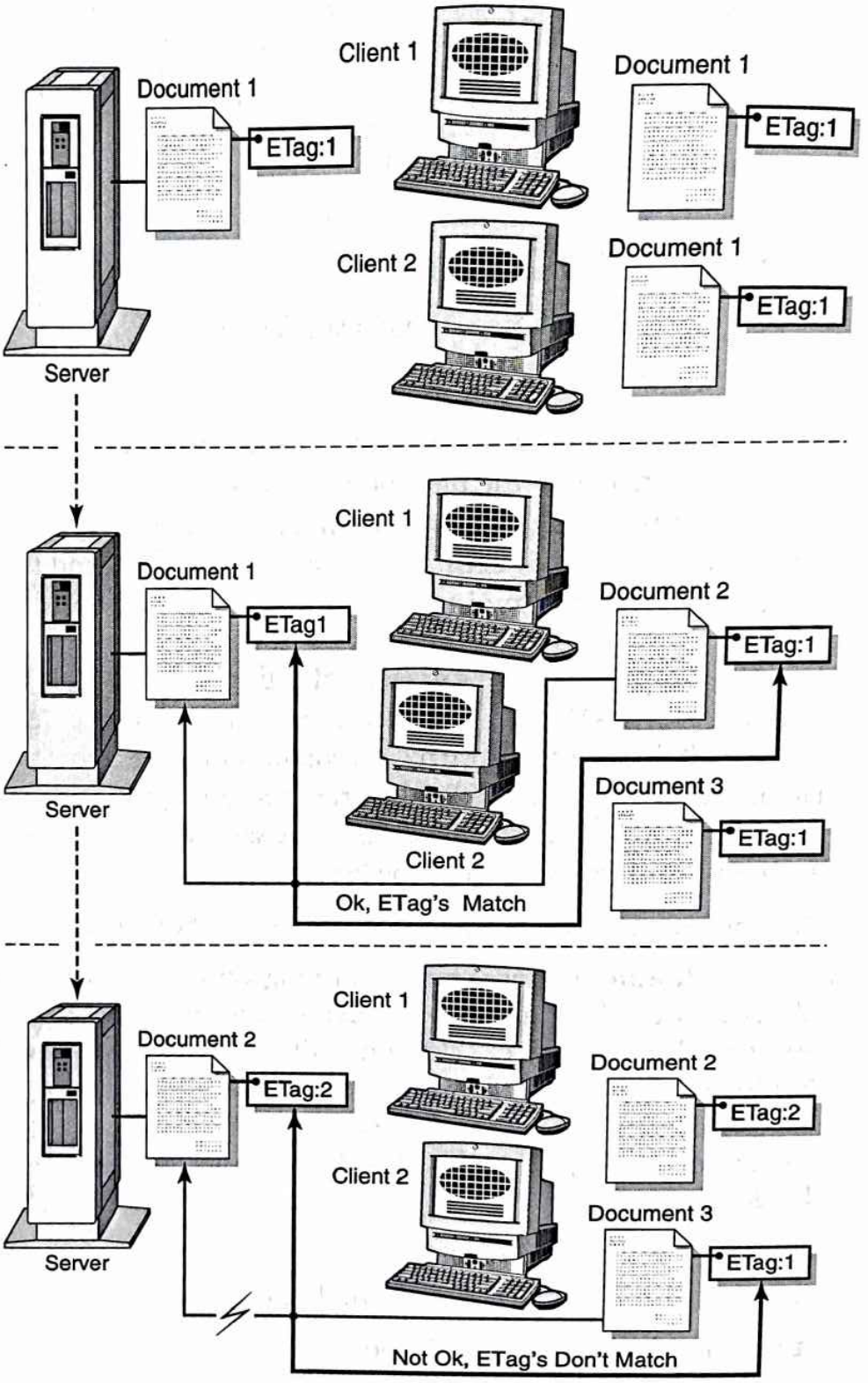


Figure 4.9 Entity tags and conditional requests used to prevent accidental loss of updates.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 47

without object entity, saving bandwidth. However, if the document has changed, the new version is returned as a regular response, with a new entity tag enclosed:

```
HTTP/1.1 200 Ok
Server: Netscape-Enterprise/4.0
Date: Sun, 11 May 1997 09:30:37 GMT
ETag: "doc-id-2442"
Content-type: text/html

... newer version of document here...
```

The "If-Range:" Request Header

The `If-Range:` header is used with byte range requests to guarantee that any new byte range responses are generated from the same source object as previous byte ranges. Otherwise, the client might have corrupt data, as some of the ranges could be from an older version, some from a newer one. The `If-Range:` request header quotes the entity tag of the object for which the client already has received one or more range responses:

```
If-Range: "entity-tag"
```

If the entity tag matches the entity tag of the object on the origin server, the requested byte range(s) are returned in a normal `206 Partial content` response. However, the semantics for the `If-Range:` header differ from the `If-Match:` and `If-Unmodified-Since:` headers when this precondition fails. Instead of the `412 Precondition failed` status code, *the entire object* is returned instead, with the `200 Ok` status. This allows clients to ask efficiently for a missing part of the object if their earlier part is still up-to-date. Otherwise, the client gets the entire object. If `If-Match:`, `If-Unmodified-Since:`, or both are used, another request would have to be made in order to get the object document in the case that the object has changed and the client's current byte ranges have become stale. In other words, the `If-Range:` header is a specialized version of the `If-Match:` header to optimize this case.

The "Max-Forwards:" Request Header

The `Max-Forwards:` header can be used with the `TRACE` method (see the section on the trace method on page 68) to limit the number of hops that the request can make. This is useful if the proxy chain appears

to have a loop in it. The `Max-Forwards:` header specifies the maximum number of hops that the `TRACE` request can make until it gets bounced back (the response is generated); for example,

```
Max-forwards: 12
```

Each proxy in a chain decrements the value of this header by one, and once the value reaches zero the request will no longer be forwarded but immediately responded to by the proxy. If the request reaches the destination origin server before `Max-Forwards:` reaches zero, the origin server will generate the response instead.

The "Range:" Request Header

The `Range:` header is used to make range retrieval requests. The most common type of range request is a byte range request. A byte range request does not request the entire resource, but instead a certain portion of it, addressed by a range of bytes. Byte numbering starts from zero. For example, to get the first 500 bytes of a resource:

```
GET http://www.somesite.com/somefile
Range: bytes=0-499
```

If the second number in the range specification is left out, it refers to the end of document. For example, the following means "bytes starting from offset 4096 all the way to the end of file":

```
Range: bytes=4096-
```

If the first number is missing, the request means "the specified number of bytes from the end of file." For example, to get the last 500 bytes of the file, the following `Range:` header would be used:

```
Range: bytes=-500
```

While this last special case may be confusing [17], it is useful: the trailing bytes of the file can be requested without knowing exactly how long the file is. This is historically useful with file formats that have useful information in the end, such as an earlier version of the Adobe PDF format.

The response to a range request is sent with the `206 Partial content` response status code instead of `200 Ok`. The `Content-Range:` header (page 94) is included in the response specifying the actual range returned and the size of the entire resource.

Multiple byte ranges may be requested by specifying a comma-separated list of ranges in the `Range:` header:

```
Range: bytes=0-1024,2048-4096
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 49

would return the first 1 KB of the file, skip the second kilobyte, and return the next two kilobytes (bytes 2048...4096). Multiple byte ranges are returned as a multipart MIME message with the `Content-Type: multipart/byteranges`.

The "Referer:" Request Header

The `Referer:` request header contains the URL of the document that contained the reference to the requested URL:

```
Referer: http://www.somesite.com/index.html
```

This way servers can track which document the user is coming from (what document the hypertext link was in) and produce logs based on that information. Also, tracking bad or mistyped links becomes easier as it is possible to find out which document they are in [18].

The "User-Agent:" Request Header

The `User-Agent:` header reports the client software name and version number that generated the request. The basic format is

```
User-agent: software-name/version
```

For example,

```
User-agent: Mozilla/4.0
```

In practice, the field is followed by a comment enclosed in parentheses giving more specific information about the client software, for example,

```
User-agent: Mozilla/4.0 (X11; U; IRIX 6.2 IP22)
```

The format of this comment is not defined by the specification. It is specific to the client software; the above format is used by the Netscape Navigator.

Sidebar

Mozilla User-Agent. "`Mozilla`" is the code name of Netscape Navigator and it is used in the `User-Agent:` field to identify it. Historically, in the absence of a more sophisticated format negotiation scheme, the `User-Agent:` field was commonly used to determine the feature level of the client software. For example, new features introduced by the Navigator, such as HTML tables and frames, would be triggered by the origin server software if the `User-Agent:` field indicated that the client software was Netscape Navigator.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 50

Unfortunately, this mechanism hindered the deployment of these features on other client software, such as Microsoft Internet Explorer. For this reason, some client software other than Netscape Navigator also uses the magic word "Mozilla" as an indication of their software, and further identify the software in the comment section. For example, the Microsoft Internet Explorer uses

```
User-agent: Mozilla/4.0 (compatible; MSIE 4.0; Windows 95)
```

Response Headers

Response headers are headers that are applicable only for responses. They convey information related to the server and the requested resource.

The "Accept-Ranges:" Response Header

The `Accept-Ranges:` header indicates that the *server* is capable of responding to range requests. Unlike other accept headers, this is a *response header*, and it indicates the server's capabilities, not the client's. For example, to notify the client that the server supports byte range requests, the server would issue the header

```
Accept-ranges: bytes
```

The "Age:" Response Header

The `Age:` header specifies the age of the response entity since the time the response was generated by the origin server. Age is specified in seconds; for example,

```
Age: 3612
```

The section on Age of Objects on page 173 describes HTTP/1.1 object age calculations in detail.

The "Location:" Response Header

The `Location:` header is used with 3xx redirection status (see Table 4-8 on page 99) codes to indicate the redirection destination location. The content of the `Location:` field is the destination URL; for example,

```
Location: http://home.netscape.com/people/ari/
```

The "Public:" Response Header

The `Public:` header indicates methods that are supported by the origin or proxy server software:

```
Public: GET, HEAD, POST, OPTIONS, PUT
```

This is most useful in the case of unusual, new methods. Let's say a new, experimental method gets added to HTTP which allows several URLs to be requested at once; let's call this method `MGET`. Now, the server can specify that in the `Public:` header:

```
Public: GET, MGET, HEAD, POST, OPTIONS, PUT
```

The `Public:` header doesn't imply that a given method is applicable to a particular (requested) URL; the `Allow:` header described in the section on Entity Headers on page 91 is intended for that. The `Public:` header is intended for informing the client about the server's general capabilities.

Intermediate proxy servers have to remove or modify the `Public:` header such that it reflects the capabilities of the proxy server as well. Let's say the request format for the `MGET` method is so complicated that a plain vanilla proxy server without explicit `MGET` support cannot handle it. This means that the proxy server will have to either remove the `Public:` header altogether, or at least remove the `MGET` method name from that field.

See the section on Unknown Method Tunneling on page 138 for a discussion about how and why unknown methods can often work through plain vanilla proxy servers which don't really know anything about those methods.

The "Server:" Response Header

The `Server:` response header identifies the server software that generated the response. For example,

```
Server: Apache/1.2
```

Some server software includes separate entries for its different components. As an example, the CERN server identifies its own version, as well as the WWW library that is a component of it:

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

The "WWW-Authenticate:" Response Header

The `WWW-Authenticate:` header is an authentication challenge header sent with the `401 Unauthorized` status code. This header contains authentication parameters that the client should use when preparing the authentication challenge response to an origin server. See the section on HTTP Authentication on page 54 for more details on HTTP server authentication.

The "Proxy-Authenticate:" Response Header

The `Proxy-Authenticate:` header is a proxy authentication challenge header sent with the `407 Proxy Authentication Required` status code. It specifies authentication parameters for the client to use when constructing the authentication credentials to the proxy server. See the section on Proxy Server Authentication on page 55 for more details on proxy authentication.

The "Retry-After:" Response Header

The `Retry-After:` header can be used in connection with the status code `503 Service unavailable`. It indicates that the requesting client or proxy server may retry the request after the specified interval or at the specified time. If the value is an integer number it indicates the number of seconds to wait. For example, to indicate that the service should be back up in two minutes:

```
Retry-after: 120
```

The header may also have a full date and time specification which indicates the absolute time at which the request may be retried:

```
Retry-after: Tue, 20 May 1997 09:00:00 GMT
```

The "Vary:" Response Header

A single URL may point to a document with multiple different representations. Examples of different variants might be

- language
- document format (HTML, PDF)
- different HTML features used based on the *user-agent* (client software)

This poses a problem with caching. A French speaking user may have requested a certain document through a proxy server, and the language

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 53

negotiation mechanism on the origin server has chosen to send the French version of the document. The French document gets cached by the proxy. At a later time, an English speaker requests the same URL but gets the cached French version from the cache, even though there would have been an English version on the origin server.

The HTTP/1.1 protocol attempts to solve this problem by using the `Vary:` header. This header lists the request headers on which the document content may vary. For example, a document available in several languages would have the following `Vary:` header in the response:

```
Vary: Accept-language
```

The above is currently the most common use of the `Vary:` header. However, a document available for multiple languages, with different representations for different client programs, could have the following header:

```
Vary: Accept-language, User-agent
```

An HTTP/1.1 proxy server must support the `Vary:` mechanism. This means storing multiple different variations of the document pointed to by a single URL. The server notifies the proxy via the `Vary:` header which request headers affect the resulting document.

As an example, say the proxy receives the following request:

```
GET http://home.netscape.com/ HTTP/1.0
User-agent: Mozilla/3.0
Accept-language: en
```

and the server responds with

```
HTTP/1.0 200 Ok
Vary: Accept-language
Content-type: text/html
Content-language: en
Content-length: 4242
...
```

The proxy will have to store not only the URL, but also the fact that this document has variants based on the `Accept-language:` request header, and the value of that header in the request that yielded this cached response.

For future requests, the proxy may return the said document from the cache *only* if the request specifies English language (code "en") to be one

of the acceptable alternatives to the user. That is, if the proxy receives the request

```
GET http://home.netscape.com/ HTTP/1.0
User-agent: Mozilla/3.0
Accept-language: fr
```

the proxy will have to forward the request to the origin server to check if a French language version of the document exists (code `fr`).

The special form of the `Vary:` header

```
Vary: *
```

means that the response varied on factors other than the request headers, such as the IP address where the request originated. If this form is returned by the origin server, the proxy server is required to perform a check with the origin server every time, even if a cached copy might still be fresh, in order to allow the origin server to perform the correct variant selection.

The "Warning:" Response Header

The `Warning:` header allows the origin server or intermediate proxy servers to attach warning messages indicating additional status information of the resource in a human-readable form. A warning header contains a warning code, warning agent identifier (either the proxy address, or a pseudonym), and a human-readable warning message:

```
Warning: 10 proxy-id "Revalidation failed"
```

Table 4-4 lists warning codes defined by the HTTP/1.1 specification. Warnings are attached to the message if there is a risk that the response is no longer fresh, or it is known that the response is stale, but the cached copy was still used because revalidation failed, or some other reason. Such warnings may be displayed to the user by the client software.

Table 4-4 HTTP/1.1 warning codes.

Code	Meaning
10	Response is stale
11	Revalidation failed
12	Disconnected operation

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 55

Table 4-4 HTTP/1.1 warning codes. *(Continued)*

Code	Meaning
13	Heuristic expiration
14	Transformation applied
99	Miscellaneous warning

Entity Headers

Entity headers carry meta information about the requested resource. This section briefly describes the entity headers defined by the HTTP/1.1 specification.

The "Allow:" Entity Header

The `Allow:` header lists the HTTP methods that are supported by the requested URL. For example,

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

The "Content-Base:" Entity Header

The `Content-Base:` header defines the URL which the relative URLs within the returned document are relative to. Let's use an example to illustrate the purpose and use of this header. Let's say we have a document pointed to by the URL

```
http://www.somesite.com/dir/file.html
```

and that document has a relative link

```
file2.html
```

It would normally be considered as a reference to a full URL:

```
http://www.somesite.com/dir/file2.html
```

that is, to be in the same "directory" as the document that pointed to it. Similarly, a reference such as

```
../file3.html
```

would be expanded to

```
http://www.somesite.com/file3.html
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 56

Now, with the `Content-Base:` header it is possible to change the URL to which the references are considered relative. If the following header field were present in the response

```
Content-base: http://www.somesite.com/somedir/xyzzy.html
```

then the abovementioned relative references

```
file2.html
../file3.html
```

would be considered to be references to these full URLs instead, respectively:

```
http://www.somesite.com/somedir/file2.html
http://www.somesite.com/file3.html
```

It is also possible to set the base URL from within an HTML document by using the <BASE> tag:

```
<BASE HREF="http://www.somesite.com/somedir/xyzzy.html">
```

If both the `Content-Base:` HTTP header and <BASE> HTML tag are present, the base URL set by the <BASE> HTML tag overrides the `Content-Base:` header.

The "Content-Encoding:" Entity Header

The `Content-Encoding:` header indicates the encoding of the entity body of the response. For example, if the content is compressed using "`gzip`," the following header is used:

```
Content-encoding: gzip [19]
```

The media type indicated by the `Content-Type:` header signifies the media type of the entity body *after* the decodings have been applied. For example, the header pair

```
Content-encoding: gzip
Content-type: text/html
```

indicates that the entity is an HTML document that has been compressed using "`gzip`." Another common encoding type is

```
Content-encoding: compress
```

referring to the compression performed by the "`compress`" program. The HTTP/1.1 specification notes that it is not recommended that future compression algorithms be identified by the name of the program performing the compression. In actuality, "`gzip`" refers to the Lempel-

Ziv coding (LZ77) and "`compress`" to the Lempel-Ziv-Welch coding (LZW).

If multiple encodings have been applied, they are listed in the order they were applied. For example,

```
Content-encoding: compress, uuencode
```

would mean that the object was first compressed, then uuencoded [20]. To decode the object it will be first uudecoded, then uncompressed.

The "Content-Language:" Entity Header

The `Content-Language:` header identifies the language of the returned resource entity. For example, an English language document could have

```
Content-Language: en
```

The "Content-Length:" Entity Header

The `Content-Length:` header specifies the length of the entity object in bytes. Example:

```
Content-length: 4580
```

The "Content-Location:" Entity Header

The `Content-Location:` header can be used to specify the URL or the accessed resource. This is useful when the requested URL points to a resource with multiple representations (different media types or languages, for example). The `Content-Location:` specifies the URL of the actual resource version returned and can be used later to access that exact same representation without interference from content negotiation (which for a different user might yield a different resource and cause confusion if there were references made to the specific representation of it).

```
Content-Location: http://www.somesite.com/index-en.html
```

However, proxy caches are not allowed to treat the `Content-Location:` header as an indication that a request to that URL could be satisfied by returning it from the cache. The URL specified in the `Content-Location:` field should be used only to differentiate between multiple entities returned for the same request URL. See the section on Response Headers on page 86 on content negotiation for further discussion about multiple representations and how they are handled by proxy servers.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 58

If the `Content-Base:` field is not specified, the value of `Content-Location:` is treated as the base URL for rendering relative references within the document as well. Otherwise, the explicit URL specified in `Content-Base:` header will be used (see page 91).

The "Content-MD5:" Entity Header

The `Content-MD5:` header contains the MD5 signature [21] of the entity body. This signature can be used for detecting accidental message modification during transport but is not a cryptographic guarantee that the message has not been maliciously modified (because a malicious intermediary can replace the MD5 signature with a new MD5 signature that matches the modified content).

```
Content-MD5: base-64 encoded MD5 signature
```

The `Content-MD5:` header may be created by the origin servers only—intermediate proxy servers are not allowed to insert it. This is because the `Content-MD5:` header is intended to be an end-to-end message integrity check. All intermediate proxy servers and the end client are allowed to check the message integrity by calculating the MD5 signature for the content and matching it against the value of the `Content-MD5:` header field.

The "Content-Range:" Entity Header

The `Content-Range:` header is returned in responses to byte range requests specifying the actual byte range returned, as well as the total number of bytes in the entire object:

Content-range: idx_{first}-idx_{last}/$bytes_{total}$

For example, the first 500 bytes of a 4000 byte document would be indicated with

```
Content-range: 0-499/4000
```

Note that the first byte has index zero—so the last 1000 bytes of the same document would be

```
Content-range: 3000-3999/4000
```

The "Content-Type:" Entity Header

The `Content-Type:` header specifies the media type of the object. For example, an HTML object would have

```
Content-type: text/html
```

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 59

and a GIF image

```
Content-type: image/gif
```

The "ETag:" Entity Header

The `ETag:` header specifies the *entity tag* for the returned object. Entity tags can be used in object revalidation with the `If-Match:` and `If-None-Match:` headers, and with object comparison with multiple content variants (the `Vary:` header).

```
ETag: entity-tag
```

Entity tags are unique identifiers for a specific version or representation of the object. The entity tag changes if the object is changed. The entity tag is an opaque quoted string; its internal format is up to the Web server software implementor:

```
ETag: "doc-id-2441"
```

The "Expires:" Entity Header

The `Expires:` header specifies the expiration date and time of the object. A cached copy of the object should not be used after this time without revalidation. See the section on HTTP/1.1 Cache Control on page 176 for a full discussion about the HTTP header fields pertaining to caching. Usually, the `Expires:` header contains the exact date and time at which the object expires:

```
Expires: Sat, 26 Apr 1997 06:03:24 GMT
```

However, there are implementations that will send any of the following to indicate immediate expiration:

```
Expires: 0                          ⇐ That's a zero.

Expires: now
```

Any such "invalid" `Expires:` field value should be treated as "expires immediately".

The "Last-Modified:" Entity Header

The `Last-Modified:` header specifies the creation or last modification time of the object on the origin server:

```
Last-modified: Sun, 11 May 1997 09:30:37 GMT
```

This time stamp is used in conditional GET requests in the `If-Modified-Since:` request header to perform cache up-to-date checks. See

the sections on HTTP/1.1 Cache Control Terminology on page 173 and on HTTP/1.1 Cache Control on page 176.

HTTP RESPONSE STATUS CODES

HTTP status codes are divided to five categories; status codes are in the range 100–599. The first digit determines the overall meaning of the status code; the two remaining digits specify the condition in more detail. The main categories are presented in Table 4-5. Individual status codes are listed in Table 4-6 through Table 4-10. The following sections will briefly describe each status code defined by the HTTP/1.1 specification [22].

Table 4-5 HTTP status code categories.

Code	Category	Description
1xx	Informational	Provisional, informational status code; actual status code will be sent once processing progresses further. This class of status codes indicated that the request has not been refused, but the actual result is not yet known.
2xx	Successful	Request received and processed successfully.
3xx	Redirection	Further action required by the client program. This class of status codes is used to redirect the client to a different location (URL), and with up-to-date checks to notify that a cached copy is still up-to-date.
4xx	Client Error	Indicates an error in the request due to the client.
5xx	Server Error	Indicates an error on the server side.

Table 4-6 HTTP/1.0 and HTTP/1.1 1xx Informational status codes. Informational status codes were a new addition to HTTP/1.1, so they didn't exist in HTTP/1.0.

Code	1.0	Meaning
100		Continue
101		Switching Protocols

1xx Informational Status Codes

100 `Continue` Indicates that the client may continue with its request. This status code is useful when performing PUT, especially on a large document. The client will first send only the request header, and upon receipt of the `100` response it continues with the actual document content.

This way, the server has a chance of rejecting the request before the actual data transfer occurs. This is important especially considering the HTTP authentication model, allowing the authentication credentials to be challenged without having the client start the data transfer when the first request is doomed to fail due to authentication challenge.

101 `Switching Protocols` Indicates a protocol change; switching protocol versions, or the protocol itself.

Table 4-7 HTTP/1.0 and HTTP/1.1 2xx Successful status codes. Status codes already present in HTTP/1.0 are marked with a bullet; others were new addition to HTTP/1.1.

Code	1.0	Meaning
200	•	OK
201	•	Created
202	•	Accepted
203		Non-Authoritative Information
204	•	No Content
205		Reset Content
206		Partial Content

2xx Successful Status Codes

200 `OK` Request succeeded and the information is returned in the response. This is the most common status code.

201 `Created` The request resulted in a newly created resource. The new resource is actually created before this response status is be sent. If the resource will be created at some time in the future (as a result of batch processing), the status code `202` is used instead.

202 `Accepted` Indicates that the request has been accepted for further processing. As an example, at a later time, a new resource may get created as a result of an accepted request.

203 `Non-Authoritative Information` The returned metainformation is returned from a cached copy, not the origin server, and may thus be incorrect or incomplete.

204 `No Content` The response is intentionally blank; the request has been successfully processed, but the client should not change the current document view but instead remain on the same page.

205 `Reset Content` Intended to notify the client to reset (not replace) the current document in the document view. This is useful for clearing form field values after submitting a form, so that the next set of values can be input on the same form.

206 `Partial Content` The response is a partial document, as requested by a partial `GET` request. Byte range responses are returned with this status code.

3xx Redirection Status Codes

Table 4-8 HTTP/1.0 and HTTP/1.1 3xx Redirection status codes.

Code	1.0	Meaning
300		Multiple Choices
301	•	Moved Permanently
302	•	Moved Temporarily
303		See Other
304	•	Not Modified
305		Use Proxy
306		Proxy Redirection [23]

`300` `Multiple Choices` The requested document has multiple representations, and the preferred version can be chosen, either manually by the user or automatically by the client software. The HTTP/1.1 specification does not specify exactly how automatic selection should be done.

`301` `Moved Permanently` The requested resource has been permanently moved to a different location; this new location URL is specified in the `Location:` header. The client software will automatically retrieve the new URL instead.

`302` `Moved Temporarily` Similar to code `301`, but the move is considered "temporary." In practice, this is usually used to "fix" URLs to contain the trailing slash so that relative links work correctly in directory `index.html` files.

`303` `See Other` Used to automatically redirect the client to a different URL as a result of a `POST` method.

`304` `Not Modified` A response indicating that the client's or proxy server's cached copy is still up-to-date. The document content is not transferred.

305 `Use Proxy` The request should be performed through the specified proxy server instead. This is currently not fully specified by the HTTP/1.1 specification.

306 `Proxy Redirection` A proposed extension to HTTP/1.1; not fully specified.

4xx Client Error Status Codes

Table 4-9 HTTP/1.0 and HTTP/1.1 4xx Client Error status codes.

Code	1.0	Meaning
400	•	Bad Request
401	•	Unauthorized
402		Payment Required
403	•	Forbidden
404	•	Not Found
405		Method Not Allowed
406		Not Acceptable
407		Proxy Authentication Required
408		Request Timeout
409		Conflict
410		Gone
411		Length Required
412		Precondition Failed
413		Request Entity Too Large
414		Request-URI Too Large
415		Unsupported Media Type

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 65

400 `Bad Request` A generic error code indicating that the request could not be understood by the server. Used when no other error code is applicable, or if the exact error is not known or does not have its own error code.

401 `Unauthorized` Authentication challenge (page 54).

402 `Payment Required` Reserved for future use.

403 `Forbidden` The server refuses to fulfill the request. For example, the document may be protected so that it is not accessible from the requesting IP address.

Some servers, wishing to protect even the fact whether or not the document exists, use the `404 Not Found` status code instead.

404 `Not Found` The requested document does not exist. Some servers may use this status code even when the document does exist, but the user is unauthorized to access it. This way, even the existence of the document is not disclosed to an unauthorized user.

Other servers may use the opposite approach: always use the `403 Forbidden` response, even when the requested document truly doesn't exist. The end result is the same: it is impossible to determine which URLs really exist, and which are invalid.

405 `Method Not Allowed` The request method is not allowed for the specified URL.

406 `Not Acceptable` None of the available representations are acceptable to the client, according to the accept headers [24]

407 `Proxy Authentication Required` Proxy authentication challenge (page 55).

408 `Request Time-out` The client did not send a request within the time that the server was prepared to wait.

409 `Conflict` The requested action could not be performed due to a conflict. A typical case is with the PUT request, when there have been modifications made to the same base version by two or more users.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 66

410 `Gone` The requested resource is no longer available, and new location is not known or does not exist. This is an alternative status code for `404 Not Found` that can be used when the server has knowledge that the resource *used to* exist on this server, but now it has been deleted.

411 `Length Required` Request is missing the `Content-Length:` header when it is required.

412 `Precondition Failed` A precondition set for the request failed, and therefore the request cannot be carried out. This status code is used with conditional requests (`If-Match:`, `If-None-Match:` and `If-Unmodified-Since:` headers) to indicate that the condition was not met.

413 `Request Entity Too Large` The request entity is too large for the server to process. This can be used to prevent denial-of-service attacks where a malicious client sends an excessive amount of data to the server in order to cause it to waste its resources.

414 `Request-URI Too Large` The request URL is too long.

415 `Unsupported Media Type` The request entity is of unsupported type.

5xx Server Error Status Codes

Table 4-10 HTTP/1.0 and HTTP/1.1 5xx Server Error status codes.

Code	1.0	Meaning
500	•	Internal Server Error
501	•	Not Implemented
502	•	Bad Gateway
503	•	Service Unavailable
504		Gateway Timeout
505		HTTP Version Not Supported

`500` `Internal Server Error` A generic error code indicating an unexpected error on the server.

`501` `Not Implemented` The request could not be serviced because the server software does not support the functionality that would be required to fulfill the request.

`502` `Bad Gateway` An intermediate proxy server received a bad response from another (cascaded) proxy server or the destination origin server.

`503` `Service Unavailable` Service is temporarily unavailable, due to high load or maintenance being performed on the server.

`504` `Gateway Time-out` An intermediate proxy server timed out waiting for a response from another (cascaded) proxy server or the destination origin server.

`505` `HTTP Version Not Supported` The request HTTP protocol version is not supported by the server.

SUMMARY

Congratulations—you survived the HTTP chapter! I promise the following chapters on other Web-related protocols are not going to be as heavy.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 68

We went to this much detail with HTTP since it is in such a central position in the Web, and especially in the proxy servers. Many of the features discussed in this book may be hard to grasp without understanding how the HTTP protocol works. You are now ready to read the rest of the chapters in the order of your interest. You have the basic knowledge necessary to understand the rest of this book.

Endnotes

1.<cr><lf> stands for carriage return and linefeed characters, respectively. These abbreviations are used in the first couple examples; later, they are omitted, and the reader is assumed to keep in mind that they are always present in the request and response headers.

2. HTML stands for HyperText Markup Language, which is the language used for creating hypertext documents.

3. At first HTML forms were submitted as queries with the GET method, but POST is a cleaner mechanism for doing that.

4. Because the server implicitly knows its hostname, port number, and the protocol it uses.

5. Remember, HTTP/1.0 request can be understood by an HTTP/0.9 server, as it will simply ignore the extra information present in an HTTP/1.0 request, and the front portion of a HTTP/1.0 request looks exactly like an HTTP/0.9 request. In practice, however, there are other problems that require clients to re-do the request using HTTP/0.9. Namely, the TCP stack tends to reset the connection because the client sends more data than the server is willing to read.

6. HTTP/1.0 had persistent connections as an extension. However, it had limitations and problems with proxies that did not support it—causing connections to remain open and leaving proxies to think that there was still data to retrieve.

7. TCP performs a 3-way handshake to establish the connection, which introduces three extra round trips in the beginning, before actual data can be transferred. The latency induced by these round trips may be noticeable.

8. Netscape Navigator is a registered trademark of Netscape Communications Corporation.

9. "In the clear" means "not encrypted", or "in cleartext".

10. Note that this problem is not unique to the Web technology; other network applications have had similar problems.

11. Some client software also appends the port number, separated by a colon.

12. This is totally fictitious—there is currently no connection multiplexing extension to HTTP/1.1.

13. "Hop-by-hop" means that the header is not propagated to further connections, but affects a single connection only, and is then stripped (and, possibly re-introduced).

14. See page 173 for a description of HTTP object age calculations.

15. RFC 822 is the Standard for the Format of ARPA Internet Text Messa es, revised by David H. Crocker.

16. Such as Lynx.

17. At a glance it might look like it means the *first* 500 bytes.

18. It's rather ironic that the header field assisting in finding mistyped links is itself mistyped...:-)

19. For historical reasons, the string "x-gzip" is used by some server software instead of "gzip".

20. The term "`uuencode`" comes from "Unix to Unix encoding", an encoding that represents binary data in a printable form so that it can be transferred over a 7-bit (non-binary) protocol, such as e-mail.

21. See the section on The MD5 Algorithm on page 255 for a discussion about MD5 signatures and message integrity.

22. HTTP/1.1 only.

23. Proposed.

24. The term "accept headers" is used collectively for all of the various accept headers: `Accept:`, `Accept-Charset:`, `Accept-Encoding:` and `Accept-Language:`.

LENNOX EXHIBIT 1014
Lennox Industries Inc. v. Rosen Technologies LLC, IPR2023-00715, Page 70