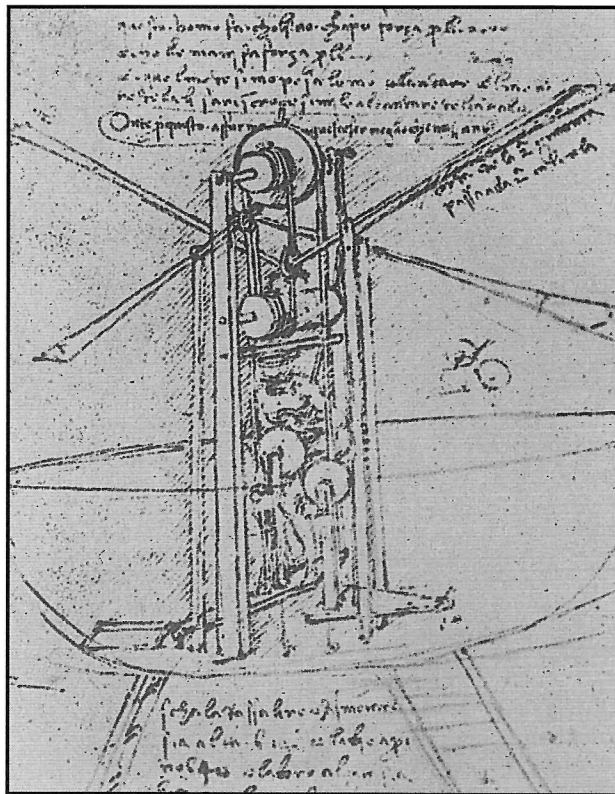


WEB PROXY SERVERS

▼
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

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. Caven*

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 widely 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

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-

protocol 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
```



Client

Figure 4.1

a) H
connb) H
perf

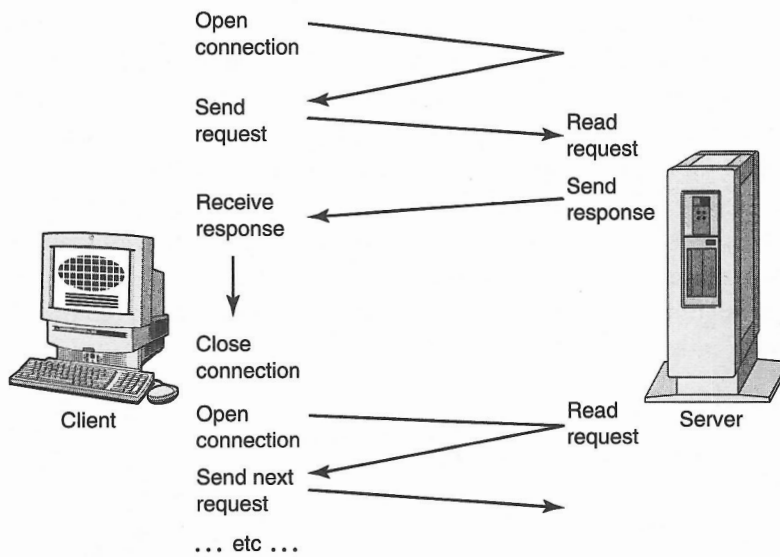
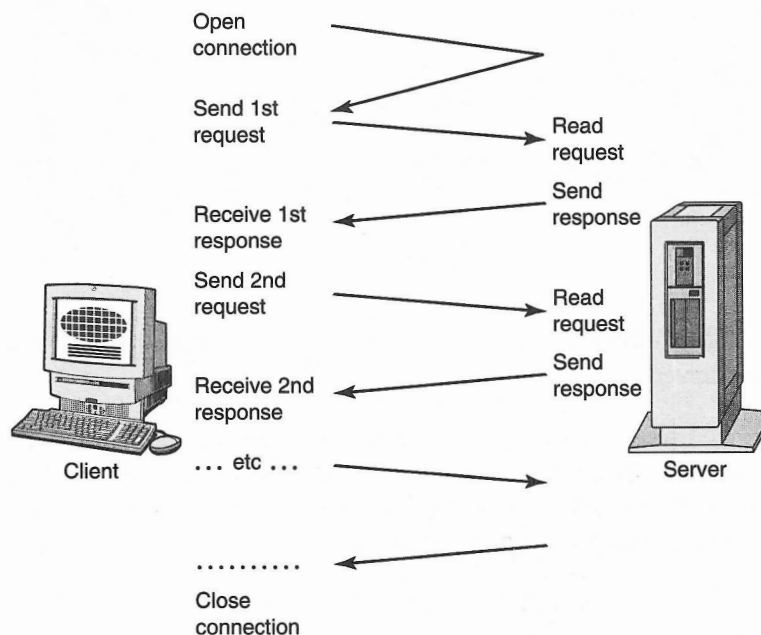


Figure 4.1 a) HTTP without persistent connections. Each request is made over a new connection.



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.

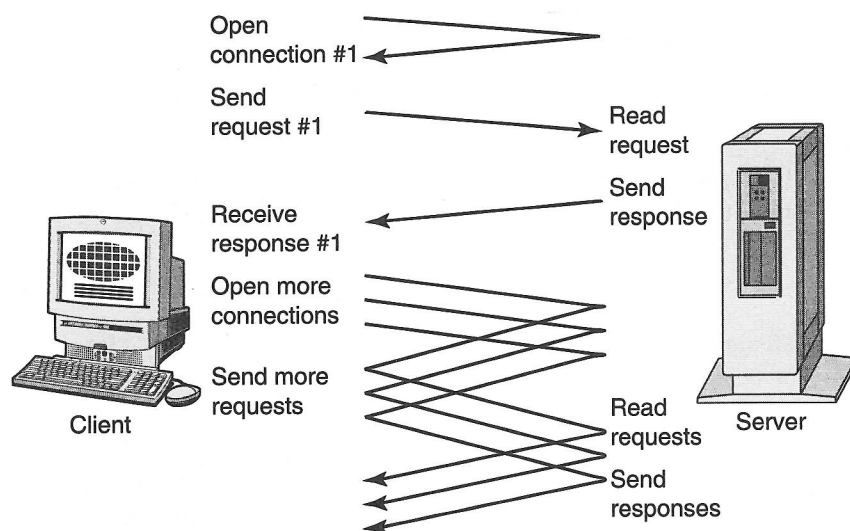


Figure 4.2 Multiple simultaneous connections to increase speed and provide better perceived performance.

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



Figure 4.3 A

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.

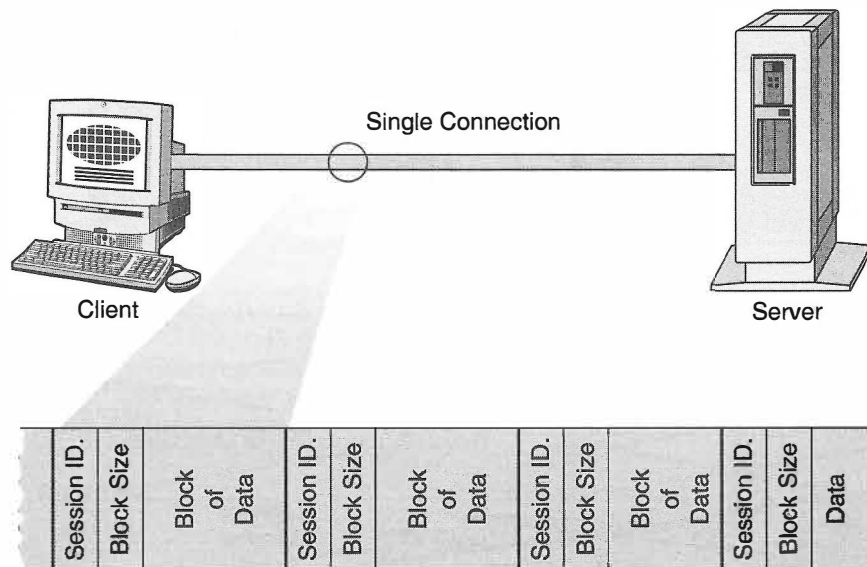


Figure 4.3 A session layer protocol.

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 congestion 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.

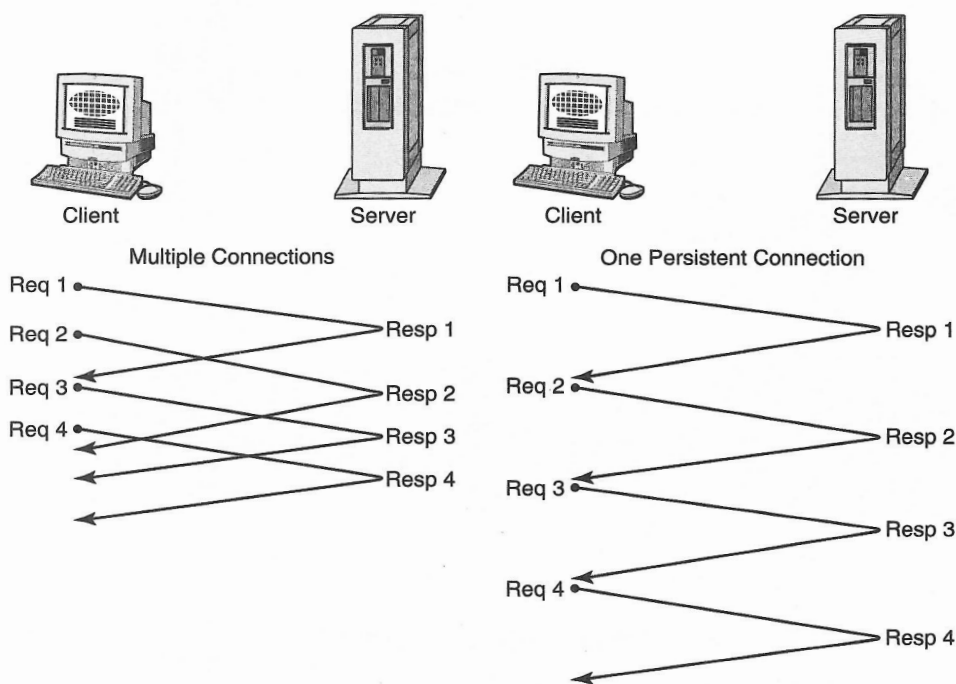


Figure 4.4 Comparison between multiple simultaneous connections and a single persistent connection.

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

there
there

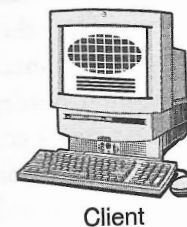


Figure 4.5 Two simultaneous persistent connections.

HTTP
To fix
HTTP
mean
the p
illustr

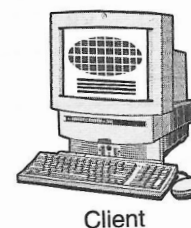


Figure 4.6 HTTP/1.1 persistent connections.

P
exten

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

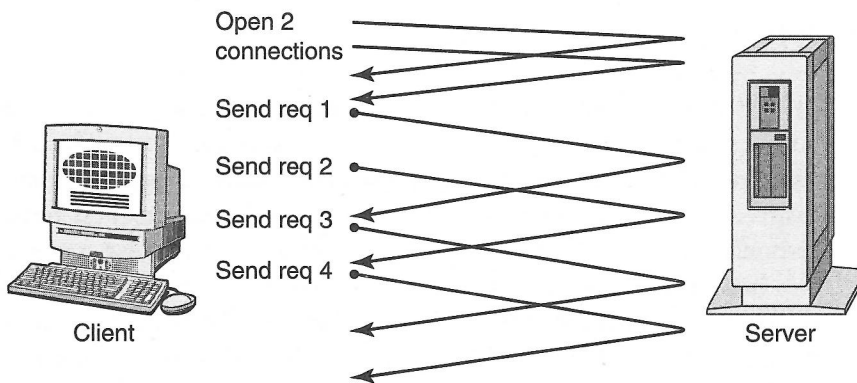


Figure 4.5 Two simultaneous persistent connections.

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.

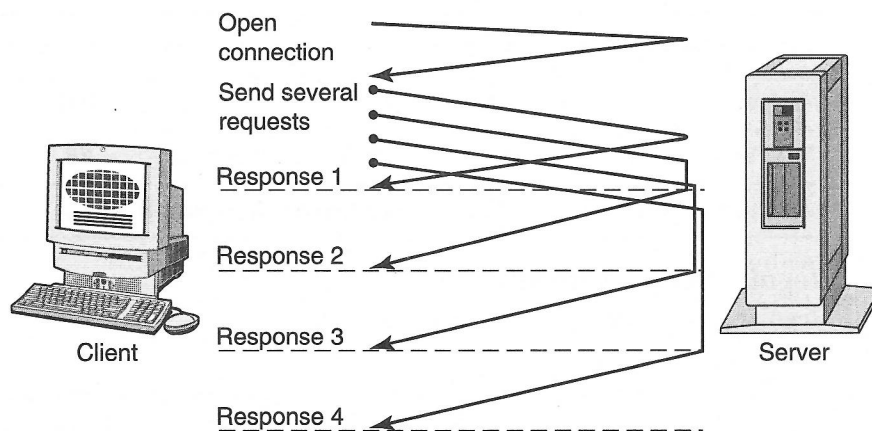


Figure 4.6 HTTP/1.1 request pipelining with persistent connections.

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.