# Addressing the Challenges of Web Data Transport

by

*Venkata Narayana Padmanabhan*

B.Tech. (Indian Institute of Technology, Delhi) 1993
M.S. (University of California at Berkeley) 1995

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Randy H. Katz, Chair
Professor Steven R. McCanne
Professor J. George Shanthikumar

1998

# Addressing the Challenges of Web Data Transport



by

*Venkata Narayana Padmanabhan*

**Abstract**

## Addressing the Challenges of Web Data Transport

by

*Venkata Narayana Padmanabhan*

*Doctor of Philosophy in Computer Science*

*University of California at Berkeley*

*Professor Randy H. Katz, Chair*

In just a few years since its inception, the World Wide Web has grown to be the most dominant application in the Internet. In large measure, this rapid growth is due to the Web's convenient point-and-click interface and its appealing graphical content. Since Web browsing is an interactive activity, minimizing user-perceived latency is an important goal. However, layering Web data transport on top of the TCP protocol poses several challenges to achieving this goal.

First, the transmission of a Web page from a server to a client involves the transfer of multiple distinct components, each in itself of some value to the user. To minimize user-perceived latency, it is desirable to transfer the components concurrently. TCP provides an ordered byte-stream abstraction with no mechanism to demarcate sub-streams. If a separate TCP connection is used for each component, as with HTTP/1.0, uncoordinated competition among the connections could exacerbate congestion, packet loss, unfairness, and latency.

Second, Web data transfers happen in relatively short bursts, with intervening idle periods. It is difficult to utilize bandwidth effectively during a short burst because discovering how much bandwidth is available requires time. Latency suffers as a consequence.

To address these problems, we first developed a new connection abstraction for HTTP, called *persistent-connection HTTP* (*P-HTTP*). The key ideas are to share a persistent TCP connection for multiple Web page components and to pipeline the transfers of these components to reduce latency. These ideas, developed by us in 1994, have been adopted by the HTTP/1.1 protocol. The main drawback of P-HTTP, though, is that the persistent TCP connection imposes a linear ordering on the Web page components, which are inherently independent.

This drawback of P-HTTP led us to develop a comprehensive solution, which has two components. The first component, *TCP session*, decouples TCP's ordered byte-stream service abstraction from its congestion control and loss recovery mechanisms. It *integrates* the latter mechanisms across the set of concurrent connections between a pair of hosts, thereby combining the flexibility of separate connections with the performance efficiency of a shared connection. This integration decreases download time by up to a factor of ten compared to HTTP/1.0 layered on standard TCP. TCP session does not alter TCP's messaging semantics, so deployment only involves local changes at the sender.

The second component of our solution, *TCP fast start*, improves bandwidth utilization for short transfers by reusing information about the network conditions cached in the recent past. To avoid adverse effects in case the cached information is stale, TCP fast start exploits priority dropping at routers, and augments TCP's loss recovery mechanisms to quickly detect and abort a failed fast start attempt.

In addition to the two challenges we have discussed, a third challenge arises from the increasing deployment of *asymmetric access networks*. Although Web browsing has asymmetric bandwidth requirements, bandwidth asymmetry could adversely impact Web download performance due to a disruption in the flow of acknowledgement feedback that is critical to sustaining good TCP throughput. To avoid performance degradation, we have developed end-host and router-based techniques, that both reduce disruption in the feedback and reduce TCP's dependence on such feedback. In certain situations, these techniques help decrease download time by a factor of fifteen.

This thesis includes mathematical and trace-based analysis, simulation, implementation and performance evaluation. In addition to the algorithms that we have designed and software that we have developed, our contributions include a set of paradigms for advancing the state-of-the-art in Internet transport protocols. These paradigms include the use of shared state and/or persistent state, and the exploitation of differentiated services mechanisms in routers.

Professor Randy H. Katz, Chair

*To my parents and my sisters, Priya and Sumi.*

*Vidya dadaathi vinayam*

*vinayaath yaathi paathrathaam*

*paathrathwaath dhanam apnothi*

*dhanaath dharmam thatha sukham*

*(ancient Sanskrit saying)*

**English Translation**

Education gives modesty;

through modesty one becomes worthy;

through worthiness one gets wealth;

through wealth one can be righteous and happy.

# Table of Contents

**Chapter 1**
**Introduction** 1

1.1 Growth of the Internet 3
1.2 TCP/IP Protocol Architecture 5
1.3 World Wide Web 7
1.3.1 Web Data Characteristics 7
1.3.2 Web Traffic Characteristics 9
1.4 Transmission Control Protocol (TCP) 10
1.5 Mismatch Between the Web Application and the TCP Protocol 12
1.6 Impact of Heterogeneous Access Network Technologies 13
1.7 Addressing the Challenges 15
1.8 Design Principles 17
1.9 Thesis Outline 18

**Chapter 2**
**Background and Related Work** 20

2.1 Transmission Control Protocol (TCP) 20
2.1.1 TCP Connection Setup 21
2.1.2 TCP Protocol Control Block (TCB) 22
2.1.3 TCP Acknowledgements 23
2.1.4 Loss Detection and Recovery 24
2.1.5 Congestion Avoidance and Control 25
2.1.6 TCP Options 28
2.1.7 Summary 29
2.2 World Wide Web 30
2.2.1 Hypertext Transfer Protocol version 1.0 (HTTP/1.0) 30
2.2.2 Performance Problems 32
2.2.3 Summary 33
2.3 TCP Enhancements 33
2.3.1 TCP NewReno 33
2.3.2 Explicit Congestion Notification (ECN) 34
2.3.3 TCP Vegas 35
2.3.4 Transaction TCP (T/TCP) 35
2.3.5 TCP Protocol Control Block Interdependence 36

2.3.6 Rate-Based Pacing (RBP) ... 37
2.3.6.1 TCP over Asymmetric Networks ... 38

2.4 Other Transport Protocols ... 38
2.4.1 NETBLT ... 39
2.4.2 Packet-Pair Bandwidth Probing ... 39

2.5 Summary ... 40

**Chapter 3**
**Methodology** **41**

3.1 Experimental Network Testbed ... 42
3.1.1 DirecPC Satellite Network ... 42
3.1.2 Hybrid Wireless Cable Modem Network ... 44
3.1.3 Ricochet Packet Radio Network ... 45

3.2 Web Server Traffic Trace Analysis ... 46
3.2.1 The IBM 1996 Olympics Web Server ... 46
3.2.2 Data Collection Methodology ... 47
3.2.3 Reconstructing the TCP Packet Trace ... 48

3.3 Simulation Study ... 49

3.4 Implementation ... 51

3.5 Evaluation ... 51

3.6 Summary ... 52

**Chapter 4**
**Analysis of HTTP Performance** **53**

4.1 HTTP/1.0 Basics ... 53
4.1.1 Sequential Retrieval ... 54
4.1.2 Concurrent Retrieval ... 55

4.2 Performance Analysis ... 55
4.2.1 Mathematical Analysis ... 55
4.2.2 Impact of Connection Length on Throughput ... 59
4.2.3 Impact of Limited Bandwidth ... 59

4.3 Analysis of Traces from the IBM Olympics Web Server ... 61
4.3.1 TCP Window Size Distribution ... 62
4.3.2 Effectiveness of TCP Loss Recovery ... 64
4.3.3 Effectiveness of TCP Congestion Control ... 65
4.3.3.1 Throughput Analysis ... 65
4.3.3.2 Congestion Avoidance Analysis ... 67
4.3.3.3 Congestion Control Analysis ... 68

4.4 Summary ... 71

## Chapter 5
## Persistent-Connection HTTP 72


5.1 Goals of P-HTTP ....73

5.2 Persistent Connections ....73
5.2.1 Multiplexing Logical Data Streams ....75
5.2.2 Negotiating the Use of Persistent Connections ....77
5.2.3 Terminating Persistent Connections ....78

5.3 Pipelining ....79
5.3.1 The GETALL method ....79
5.3.2 The GETLIST method ....81

5.4 Implementation ....81
5.4.1 Client-Server Implementation ....81
5.4.2 Proxy-based Implementation ....83

5.5 Experimental Results ....86
5.5.1 Terrestrial Wide-Area Network ....88
5.5.2 Satellite-based Wide-Area Network ....88
5.5.3 Interaction of P-HTTP with TCP Congestion Control ....91

5.6 Extensions of P-HTTP ....92
5.6.1 Persistent Connections in HTTP/1.1 ....93
5.6.2 Application-independent Multiplexing Protocols ....93

5.7 Limitations of P-HTTP ....93

5.8 Summary ....95


## Chapter 6
## TCP Session 96


6.1 Motivation ....96

6.2 TCP Session Overview ....98

6.3 Design and Operation of TCP Session ....100
6.3.1 Session Control Block ....100
6.3.2 Integrated Congestion Control ....102
6.3.3 Connection Scheduling ....105
6.3.4 Integrated Loss Recovery ....106
6.3.4.1 Loss Recovery Rules ....109

6.4 Implementation in BSD/OS ....111

6.5 Performance Results ....113
6.5.1 Competition versus Sharing between Concurrent Connections ....113
6.5.2 Simulation Results ....115
6.5.2.1 Impact of the Number of Simultaneous Web Downloads ....117

6.5.2.2 Impact of Integrated Loss Recovery ..... 119
6.5.2.3 Impact of the Number of Inline Images ..... 120
6.5.2.4 Impact of the Inline Image Size ..... 122
6.5.2.5 Impact of Interfering Bulk Transfer Traffic ..... 123
6.5.2.6 Impact of Heterogeneous Traffic Mix ..... 124
6.5.2.7 Impact of the Link Speed and Delay ..... 125
6.5.3 Summary of Results ..... 127
6.6 Limitations of TCP session ..... 128
6.7 Summary ..... 129

**Chapter 7**
**TCP Session: Advanced Issues** **130**

7.1 Connection Scheduling ..... 130
7.1.1 Hierarchical Round-Robin Scheduler ..... 131
7.1.2 Comparison With Alternatives ..... 133
7.1.2.1 Scheduling Independent TCP Connections ..... 133
7.1.2.2 Scheduling Logical Data Streams within a Single TCP Connection ..... 135
7.1.3 Summary ..... 136
7.2 Adapting Integrated Congestion Control for Proxy Hosts ..... 136
7.3 Potential For False Retransmissions ..... 138
7.4 Summary ..... 140

**Chapter 8**
**TCP Fast Start** **141**

8.1 Motivation ..... 142
8.2 Design of TCP Fast Start ..... 143
8.2.1 Basic Idea ..... 144
8.2.2 Sender Algorithm ..... 145
8.2.2.1 Initiation and Termination of Fast Start ..... 145
8.2.2.2 ..... Initialization of State Variables 145
8.2.2.3 Clocking Out Data During Fast Start ..... 146
8.2.2.4 Quick Detection/Recovery From Failed Fast Start Attempt ..... 146
8.2.3 Router Mechanism ..... 148
8.3 Implementation in BSD/OS ..... 149
8.4 Performance Results ..... 151
8.4.1 BSD/OS Measurements ..... 151
8.4.2 Simulation Results ..... 153
8.4.2.1 Single Web Download with Constant Load ..... 155
8.4.2.2 Single Web Download with Changed Load ..... 156
8.4.2.3 Impact of RED Buffer Management ..... 159
8.4.2.4 Competing Web Downloads with Constant Load ..... 160

8.4.2.5 Competing Web Downloads with Changed Load ... 161
8.4.2.6 Competing, Heterogeneous Web Downloads with Changed Load ... 163
8.4.3 Summary of Results ... 164
8.5 Discussion ... 164
8.6 Summary ... 166

**Chapter 9**
**Asymmetric Access Networks** **167**

9.1 Motivation ... 168
9.2 Analysis of Performance Problems ... 169
9.2.1 Unidirectional Transfer ... 169
9.2.2 Bidirectional Transfers ... 172
9.2.3 Summary of Performance Problems ... 174
9.3 Solution to the Problems Caused by Bandwidth Asymmetry ... 174
9.3.1 Reducing Disruption in the Ack Stream ... 174
9.3.1.1 Ack Congestion Control (ACC) ... 175
9.3.1.2 Ack Filtering (AF) ... 176
9.3.1.3 Acks-first Scheduling ... 177
9.3.2 Adapting to Disruption in the Ack Stream ... 177
9.3.2.1 Sender Adaptation (SA) ... 177
9.3.2.2 Ack Reconstruction (AR) ... 178
9.4 Performance Evaluation ... 178
9.4.1 Unidirectional Traffic ... 180
9.4.2 Bidirectional Traffic ... 181
9.4.2.1 Bulk Transfers ... 181
9.4.2.2 Web Download ... 184
9.4.3 Summary of Results ... 186
9.5 Discussion ... 187
9.6 Summary ... 188

**Chapter 10**
**Conclusions and Future Work** **190**

10.1 Challenges and Solutions ... 190
10.2 Contributions ... 191
10.2.1 Separation of Protocol Service Model from Protocol Mechanisms ... 191
10.2.2 Shared State ... 192
10.2.3 Persistent State ... 192
10.2.4 Exploiting Support for Differentiated Services ... 193
10.2.5 Redefining the Coupling Between the Data and Control Paths ... 193
10.2.6 Summary ... 194

10.3 Design Principles Re-visited ........................................ 194
10.3.1 Robustness ........................................ 194
10.3.2 Wide Applicability ........................................ 195
10.3.3 End-to-End Principle ........................................ 195
10.3.4 Incremental Deployment ........................................ 196
10.3.5 Summary ........................................ 196
10.4 Directions for Future Work ........................................ 196
10.4.1 Sharing State Across Hosts ........................................ 196
10.4.2 Comprehensive Use of Persistent State ........................................ 197
10.4.3 Guarding Against Malicious Users ........................................ 197
10.4.4 Improved Metrics for Quantifying Perceived Performance ........................................ 197
10.4.5 Multicast-Based Dissemination of Web Data ........................................ 198
10.5 Availability ........................................ 198

**Appendix A**
**Performance Analysis Tools ........................................ 199**

**Appendix B**
**Implementing TCP Session ........................................ 201**

**Bibliography 208**

# List of Figures


Figure 1.1 Snapshot of a Web page. ....2
Figure 1.2 The exponential growth of the Internet and the Web ....4
Figure 1.3 The OSI network protocol stack. ....6
Figure 1.4 Contrasting P-HTTP and TCP session. ....16
Figure 2.1 TCP fast retransmission mechanism. ....24
Figure 2.2 TCP congestion control dynamics. ....27
Figure 2.3 A schematic depiction of Web clients, proxies, and servers. ....30
Figure 2.4 HTTP/1.0 timeline. ....32
Figure 3.1 DirecPC network topology. ....43
Figure 3.2 Hybrid wireless cable modem network topology. ....44
Figure 3.3 Ricochet packet radio network topology. ....45
Figure 3.4 IBM Olympics server topology. ....47
Figure 3.5 Example simulation topologies. ....50
Figure 4.1 HTTP/1.0 timeline with sequential and concurrent connections. ....54
Figure 4.2 Analytically computed Web page download time. ....58
Figure 4.3 Throughput versus connection length. ....60
Figure 4.4 Distribution of the receiver advertised window. ....62
Figure 4.5 Correlation of throughput and number of concurrent connections. . ....66
Figure 4.6 Distribution of packet loss across concurrent connections. ....69
Figure 4.7 Congestion backoff factor versus number of concurrent connections. ....70
Figure 5.1 P-HTTP timeline. ....74
Figure 5.2 GETALL and GETLIST timeline. ....80
Figure 5.3 Proxy-based implementation of P-HTTP. ....83
Figure 5.4 Proxy-based approach to emulating several protocol configurations. ....85
Figure 5.5 Experimental network configuration. ....87
Figure 5.6 P-HTTP: WAN results. ....89
Figure 5.7 P-HTTP: DirecPC experimental results. ....90
Figure 5.8 Illustration of persistent connection without pipelining. ....91
Figure 5.9 Illustration of slow-start re-start. ....92
Figure 6.1 TCP session congestion window dynamics. ....104
Figure 6.2 Illustration of TCP fast retransmission. ....107
Figure 6.3 Illustration of later acks in a TCP session. ....108
Figure 6.4 Illustration of loss recovery using later ack. ....110
Figure 6.5 Rules for integrated loss recovery. ....111
Figure 6.6 Dynamics of concurrent connections with TCP and TCP session. ....114
Figure 6.7 TCP session: Simulation topology and parameter settings. ....116
Figure 6.8 Download time and packet loss versus simultaneous connections. ....118

| | | |
|---|---|---|
| Figure 6.9 | Impact of integrated loss recovery. | 119 |
| Figure 6.10 | Download time and fairness versus number of inline images | 120 |
| Figure 6.11 | Download time and fairness inline image size. | 121 |
| Figure 6.12 | Download time versus interfering bulk transfers. | 122 |
| Figure 6.13 | Impact of a heterogeneous traffic mix. | 123 |
| Figure 6.14 | Impact of link bandwidth: 28.8 Kbps and 45 Mbps. | 125 |
| Figure 6.15 | Impact of large RTT: 400 ms. | 125 |
| Figure 7.1 | Hierarchical round-robin scheduling. | 132 |
| Figure 7.2 | Sequence number trace with connection scheduling. | 134 |
| Figure 7.3 | Sequence number trace with connection scheduling and stall. | 135 |
| Figure 7.4 | Adapting integrated congestion control to proxy hosts. | 138 |
| Figure 7.5 | Impact of integrated loss recovery on false retransmissions. | 139 |
| Figure 8.1 | TCP fast start: download time versus Web page size. | 152 |
| Figure 8.2 | TCP fast start: congestion window and sequence number plots. | 153 |
| Figure 8.3 | TCP fast start: simulation topology and parameter settings. | 154 |
| Figure 8.4 | Constant-load experiment. | 155 |
| Figure 8.5 | Changed-load experiment. | 157 |
| Figure 8.6 | Impact of fast start without priority dropping. | 158 |
| Figure 8.7 | Impact of RED buffer management. | 159 |
| Figure 8.8 | Constant-load experiment: Impact of bandwidth and delay. | 160 |
| Figure 8.9 | Changed-load experiment: Impact of bandwidth and delay. | 162 |
| Figure 9.1 | Cable network: downstream throughput versus upstream bandwidth. | 170 |
| Figure 9.2 | Cable network: bidirectional transfers. | 172 |
| Figure 9.3 | Bandwidth asymmetry: simulation topology. | 179 |
| Figure 9.4 | Download time versus page size. | 180 |
| Figure 9.5 | Ack filtering: bidirectional traffic. | 182 |
| Figure 9.6 | Ack congestion control: bidirectional traffic. | 184 |
| Figure B.1 | Ack congestion control: bidirectional traffic. | 206 |

# List of Tables


| | | |
|---|---|---|
| Table 1.1 | Internet traffic distribution by application | 5 |
| Table 1.2 | Web data types | 8 |
| Table 1.3 | Access network technologies. | 14 |
| Table 4.1 | Summary of IBM Olympics server trace analysis | 63 |
| Table 6.1 | Alternative approaches to Web data transport | 97 |
| Table 8.1 | TCP fast start: download time with heterogeneous hosts | 163 |
| Table 9.1 | Cable network: throughput with bidirectional traffic | 181 |
| Table 9.2 | Cable network: download time with bidirectional traffic | 185 |

# Acknowledgements

As I stand at the threshold of earning my doctorate, I am overwhelmed when I recall all the people who have helped me get this far.

First and foremost, I would like to thank my Ph.D. advisor, Professor Randy Katz, for his constant support, guidance, and inspiration. Randy is a truly remarkable advisor who grants students a lot of freedom to explore new ideas, but at the same time interacts closely with them. I have greatly benefited both from his excellent technical advice and from his role-model as a successful team leader and researcher. I look forward to continuing my association with him in the future.

I am indebted to Jeff Mogul for introducing me to the problem of improving HTTP performance, which eventually led to my dissertation. Jeff has been a great mentor and collaborator. I would like to thank him and the Compaq (formerly Digital) Western Research Laboratory for providing me generous support, beginning with a summer internship, and continuing with access to their computing facilities as an honorary researcher.

I would like to thank Professor Steve McCanne for being on my qualifying exam and thesis committees. Steve's insightful comments and constructive criticisms have been extremely useful to me. I am also grateful to Professors George Shanthikumar and Bob Brodersen for serving on my committees and providing valuable feedback on the ideas presented in this thesis.

I have had very productive collaborations with Hari Balakrishnan, Jeff Mogul, Srini Seshan and Mark Stemm, which have contributed significantly to this thesis. I have also had valuable discussions on various aspects of the thesis with B. R. Badrinath, Sally Floyd, Tom Henderson, T. V. Lakshman, Giao Nguyen, Vern Paxson, and K. K. Ramakrishnan. My graduate student career has also been enriched by interactions with several researchers including Mary Baker, Eric Brewer, Ramón Cáceres, Domenico Ferrari, Akthar Jameel, Barry Leiner, Reiner Ludwig, Kevin Mills, Rob Ruth, Subir Varma, and Yongguang Zhang. I would like to thank them all.

This work was supported primarily by DARPA under contract DAAB07-95-C-D154. It was also supported by grants from Hughes Aircraft Corporation, Hybrid Networks Inc., Metricom Inc. and the California MICRO program. Son Dao and Yongguang Zhang of Hughes Research Labs, Ed Moura of Hybrid Inc., and Mike Ritter of Metricom Inc. helped set up the network testbed for my experimental work. Brian Shiratsuki, Keith Sklower and Ken Lutz were always willing to help straighten out any kinks that developed in the testbed. Keith Sklower was largely responsible for getting our connection to the Hughes DirecPC network off the ground. I am grateful to them all.

Kathryn Crabtree was ever helpful in guiding me through the administrative labyrinth in Berkeley. I am convinced that every research organization should have someone as adept as her in keeping bureaucratic red tape at bay! Terry Lessard-Smith and Bob Miller were great project administrators. I doubt if large systems projects in Berkeley would run half as smoothly as they do without their tireless efforts, or be half as much fun as they are without the excellent project retreats they organize.

My graduate student colleagues have made my stay at Berkeley a truly enjoyable one. Their collective intellectual energy has been a great motivator for me. I would like to thank Elan Amir, Hari Balakrishnan, Yatin Chawathe, Armando Fox, Jeff Gilbert, Steve Gribble, Tom Henderson, Todd Hodes, Dan Jiang, Bruce Mah, Giao Nguyen, Mark Stemm, Helen Wang, and Tao Ye.

The "443 Soda gang" of Elan, Hari, Todd, Mark, and members-at-large, Tom, Giao, and Srini was always an enthusiastic (and leak-proof!) audience for almost any topic under the sun, including many politically-incorrect ones. I certainly hope the gang will continue to flourish in cyberspace (with the protection of PGP encryption, of course!) as the ranks of the members-at-large swells.

I would like to thank all my teachers, from elementary school through graduate school, for providing me with an excellent education. I am particularly indebted to my professors at the Indian Institute of Technology for giving me a solid foundation in Computer Science and Engineering.

Most important of all, I would like to express my gratitude to my family for being an unstinting source of support and encouragement. My grandparents have inspired me through their courage in overcoming the challenges of life. I am saddened that they did not live to see my graduation. My parents have taught me the value of education and have worked hard to provide me the very best of it. They have always been there when I have needed them. My sisters, Priya and Sumi, have been wonderful sources of support. The tinge of jealousy I feel in them both having beaten me to earning the title "Doctor" is effectively neutralised by their abiding affection! Last but not the least, my brothers-in-law, Sekhar and Umesh, have been great friends and have given me excellent advice drawing on their own careers as doctoral students, and subsequently, researchers.

# Chapter 1

# Introduction

The decade of the 1990s has seen unprecedented growth in wide-area data networks. This period has seen the graduation of the Internet from an experimental network used primarily by the research community to a commercial network used by millions of users. This rapid growth has been fueled primarily by one application, namely the World Wide Web (the "Web" for short). The easy to use point-and-click interface coupled with the appealing graphical content have made the Web an increasingly popular medium for publishing and accessing information. The dominant status of the Web in the Internet today is underscored by recent measurements which show that Web traffic constitutes over 70% of the total on wide-area backbone links in the Internet [102].

Information in the Web is arranged as *pages* stored on *servers*. Each page typically consists of text and other data such as *inline images*, with *hyperlinks* leading to other pages. Figure 1.1 shows the snapshot of a Web page. Users interact with the Web by browsing Web pages and following hyperlinks to other pages with a Web client program called the *browser*. Being able to *download* and browse Web pages quickly is crucial for good user-perceived performance.

The *Hypertext Transfer Protocol* (HTTP) [10] defines the way Web clients and servers communicate with one another. HTTP is layered on top of the *Transmission Control Protocol* (TCP) [89],


**Figure 1.1 Snapshot of a Web page showing text, inline images and hyperlinks.**

the de facto standard protocol for reliable data transport in the Internet. The structure of Web information and the manner in which users access it interact poorly with TCP. *Addressing the performance challenges that arise because of the mismatch between the Web application and the TCP protocol forms the core of this thesis*. We preview the challenges here, and defer a detailed discussion to later sections.

1. The transmission of a Web page from a server to a client involves the transfer of multiple distinct components, such as inline images, that comprise the page. A component is often of some value to the user independent of other components of the same page. So it is desirable to transfer the components concurrently and independently of each other to minimize user-perceived latency. However, TCP has historically been more attuned to applications such as remote login (e.g., Telnet [90]) and bulk file transfer (e.g., FTP [91]) which, to first order, transfer an undifferentiated, linear sequence of bytes.

2. Since Web browsing is an interactive activity driven by a human user, it involves bursts of activity corresponding to page download interspersed with idle periods corresponding to *user think time*. So Web traffic is bursty. The amount of data transferred during each burst is usually a few tens of kilobytes [65], which is much larger than a single telnet packet containing a few key-

strokes, but is also much shorter than typical bulk transfers. This poses a dilemma in the context of Web data transfer. On the one hand, we would like each burst transferred as quickly as possible. On the other hand, a burst may be too large to be transmitted without first probing the network to determine the available bandwidth. But the latter is often a time-consuming process (as it is in TCP).

3. Web data transfer tends to be highly asymmetric with an order-of-magnitude more data flowing from servers towards clients than in the opposite direction [65][1]. This asymmetry is partly responsible for the deployment of *asymmetric access networks*, which provide (relatively) low-cost, high-bandwidth connectivity from the network core towards end users (*downstream*), but much slower-speed connectivity in the opposite direction (*upstream*). Bandwidth asymmetry adversely impacts the performance of TCP data transfer in the downstream direction because the limited bandwidth in the upstream direction disrupts the flow of receiver feedback needed to sustain data transfer.

The rest of this chapter is organized as follows. We begin in Section 1.1 with a discussion of the rapid growth of the Internet in general and the Web in particular, which underscore the practical relevance of the problem we are trying to solve. In Section 1.2, we discuss the philosophy and design of the TCP/IP protocol suite that have played a key role in enabling this growth. In Section 1.3, we discuss the distinguishing characteristics of Web data and traffic. We follow this up with an overview of TCP in Section 1.4. In Section 1.5, we identify specific challenges that arise from the mismatch between the Web application and the TCP protocol. In Section 1.6, we discuss the impact of heterogeneous access network technologies on performance. Having enumerated the performance challenges, we outline the techniques that we have developed to address these challenges in Section 1.7. In Section 1.8, we discuss the design principles that have guided our work. We conclude in Section 1.9 with an outline of the rest of this thesis.

## 1.1 Growth of the Internet

The Internet, as we know it today, has evolved from the ARPANET, a research network sponsored by the U.S. Department of Defense. The original configuration in December 1969 consisted of

1. Such asymmetry is understandable given that it is much easier for individual users to consume information than to generate information.

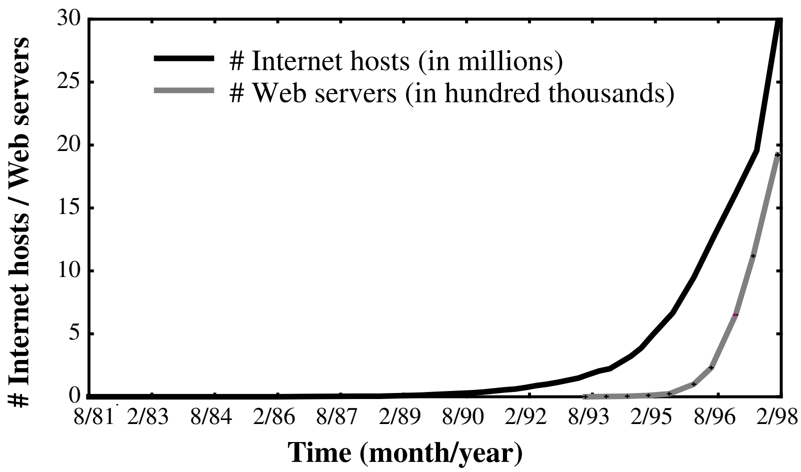



**Figure 1.2 The exponential growth in the number of Internet hosts and the number of Web servers. (Sources: [79, 40])**

four nodes (*hosts*), all located in the U.S. Today the Internet is estimated to have about 30 million hosts spread across more than 180 countries. This growth has sustained an exponential rate and shows no signs of slowing down. Figure 1.2 provides a glimpse of this rapid growth.

Two factors have contributed to this growth. The first factor is the ability of the Internet protocol architecture (colloquially referred to as the *TCP/IP protocol architecture*) to accommodate and assimilate heterogeneous network technologies. This architecture also enables the network to be grown in a decentralized manner with minimal involvement of a central controlling authority. This has facilitated the addition of new networks to the Internet on a daily basis.

The second factor that has fueled the growth of the Internet is the large number of useful applications it has supported and continues to support. Historically, these have included electronic mail, network news, file transfer, remote login, etc. In recent years, Web browsing has overtaken these to become by far the most popular application in the Internet, as evident from Table 1.1. Just as the total number of Internet hosts, the number of Web servers is also growing at an exponential rate, as depicted in Figure 1.2. In fact, it is the immense popularity of the Web that is largely responsible for the rapidly increasing numbers of hosts and users in the Internet. The Web is poised for even greater growth in the future as it increasingly becomes the method of choice for interfacing to a wide-range of devices such as printers, cameras, etc. The dominant position of the Web makes the subject of this thesis interesting and relevant. Next, we discuss the TCP/IP protocol architecture in some detail.

| Protocol/Application | % packets | % bytes |
|---|---|---|
| HTTP (Web) | 70 | 75 |
| FTP (file transfer) | 3 | 5 |
| SMTP (e-mail) | 5 | 5 |
| NNTP (network news) | 1 | 2 |
| Telnet (network terminal) | 1 | < 1 |
| DNS (name service) | 3 | 1 |

**Table 1.1 The fraction of traffic measured at a busy backbone link in August 1997 corresponding to various application-level protocols. (Source: [102])**

## 1.2 TCP/IP Protocol Architecture

The Internet, as any other network, depends on a variety of protocols for its operation. Since networks comprise multiple distributed entities, network protocols are needed to establish a common basis for communication between these entities. Network protocols are analogous to a *lingua franca* that enables a group of people to communicate with each other.

It has long been recognized that a network protocol architecture should subdivide the communication problem into *layers* and have well-defined interfaces between these layers [111, 20]. Such a subdivision simplifies protocol design. However, many approaches to layering have been proposed and how function should be assigned to layers has been an ongoing controversy. Among the many approaches, the *OSI (Open Systems Interconnection) Reference Model* [111] is perhaps the most elaborate, and provides a useful framework to structure discussion of other approaches to layering as well.

The OSI model defines a seven-layer network protocol stack. Figure 1.3 lists the seven layers. Each layer provides a set of services that builds on the services provided by the layers lower in the stack. Briefly, the physical and data link layers include functionality needed for basic connectivity. The network layer enables the routing of data packets from one host to another via intermediate nodes. The transport layer provides services such as reliable end-to-end data delivery between a

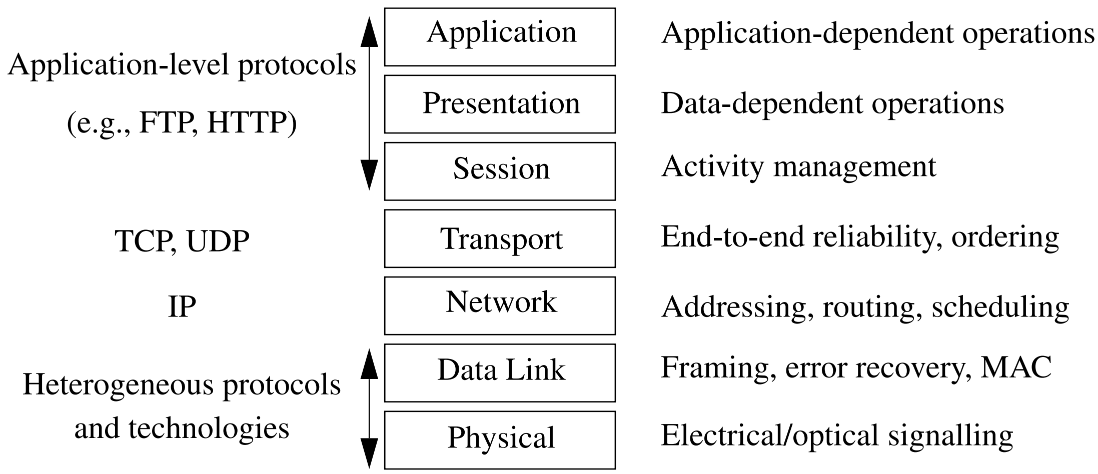



**Figure 1.3 The 7-layer OSI network protocol stack. The annotations on the right indicate the typical role of each layer. The annotations on the left show how Internet protocols map on to the OSI model.**

pair of hosts[1]. The session through application layers provide enhanced end-to-end services, such as data format conversion, that are often data-specific or application-specific.

It is useful to consider how the TCP/IP protocols [20] map on to the seven-layer OSI model. As shown in Figure 1.3, the TCP/IP protocol stack has fewer layers than the OSI model. The lowest two layers in the OSI model, physical and data link, operate within the confines of a single link. These layers provide a means for higher layers to move data from one end of a link to the other. As such, there is minimal interaction between the design of these layers and the TCP/IP protocols, the latter being concerned with only higher-layer issues. *IP* (*Internet Protocol*) is a network-layer protocol that provides addressing and routing services. Both these services exploit hierarchy to facilitate the growth of the network in a decentralized manner. *TCP* (*Transmission Control Protocol*) and *UDP* (*User Datagram Protocol*) are the two key transport-layer protocols in the Internet architecture. Finally, the top three layers in the OSI model are usually subsumed in application-level protocols, which tend to be as diverse as applications themselves. Examples include *HTTP* (*Hypertext Transfer Protocol*) for the Web, *FTP* (*File Transfer Protocol*) for file transfer and *SMTP* (*Simple Mail Transfer Protocol*) for electronic mail.

---

1. Unless otherwise specified, we only refer to *unicast* transport protocols in all of our discussion.

While layering is a convenient design tool, it can often lead to inflexibility and performance inefficiency when carried over to implementation [21]. This is primarily because the fixed function of each layer and the rigid interfaces between layers often result in a situation where one layer performs suboptimally because the information it needs for optimal operation is only available to another layer. In other words, there is a *mismatch* between the two layers.

Efficient Web data transport is difficult, in part, because of a mismatch between the application layer and the TCP layer. In this thesis we analyze this mismatch and the performance problems that it causes. We then propose a solution, *TCP session*, which resolves these problems by redefining the relationship between the functions performed by the TCP layer and the interface it provides to the application layer. Over the next three sections, we discuss the key characteristics of the Web application and the TCP protocol, and identify challenges arising from their mismatch.

## 1.3 World Wide Web

The World Wide Web is a large distributed system consisting of servers and clients spread across the Internet. Web servers are repositories of information. Web clients request information from servers, usually on behalf of human users.

Web clients and servers communicate using the *Hypertext Transfer Protocol* (*HTTP*) [10]. HTTP defines several *methods* that a client can specify in requests that it sends to the server. The most commonly used method is *GET*. The client issues a GET request to retrieve a *resource* from the server. A resource can be a static file or the output of a script executed at the server. The GET request identifies the resource with a *Uniform Resource Locator* (*URL*) [8].

We discuss the characteristics of Web data and Web traffic, which are very different from those of traditional applications and which lead to a mismatch with TCP.

### 1.3.1 Web Data Characteristics

As noted before, Web data is served in logical units known as *pages*. A Web page is usually composed of multiple components such as text, images, video, audio, etc. that are embedded in it. Table 1.2 provides a sampling of the diverse data types that occur in Web pages and their frequency of occurrence, as reported in [107]. The individual components are embedded in a Web page using the *Hypertext Markup Language* (*HTML*) [9].

| Web Data Type | Description | Frequency of Occurrence |
|---|---|---|
| HTML | Hypertext Markup Language | 76.3% |
| TXT | ASCII text | 2.2% |
| PS | Postscript | 1.8% |
| AU | Sun audio | 0.7% |
| WAV | Microsoft WAVE | 0.3% |
| GIF | Graphical Interchange Format | 61.7% |
| JPEG | Joint Pictures Expert Group | 7.8% |
| MPEG | Motion Pictures Expert Group | 0.3% |
| MOV | QuickTime Movie | 0.2 |

**Table 1.2 A subset of Web data types and the frequency of their occurrence in Web pages. (Source: [107])**

Users typically *download* entire Web pages from servers, rather just an individual component of a page. They use a *browser* program to download pages. During a download, the browser uses HTTP to first retrieve the HTML file corresponding to the Web page. It then parses the HTML to determine which components, if any, are embedded in the page. Finally, it retrieves each of the embedded components using HTTP. Although not necessary, the components of a Web page usually reside on the same server as the HTML file.

The browser renders the components in a manner that is convenient for the user to perceive. For instance, image components are displayed on a screen while audio components are played on an audio device. The browser may invoke a separate helper application to render components of data types that it is unable to process.

The interactive nature of Web browsing makes it critical to minimize latency. However, an important point is that the browser can often render a component while still in the process of retrieving it. *So the latency perceived by the user is often determined by the entire process of rendering the Web page, and not necessarily the time to download the entire page*. For instance, rendering a set of (progressively-coded) inline images concurrently may reduce user-perceived latency compared to rendering them sequentially, even if the total download time is the same in both cases.

A Web page component may be more or less valuable to the user than other components. Perceived latency would improve if the valuable components are rendered more quickly, even if the less valuable ones are slowed down as a result. For instance, the user may be more interested in viewing the headline image on a news page than an advertisement banner.

Thus, Web data transport involves the transfer of multiple, independent components. It is desirable to have the flexibility to control the progress of the individual transfers depending on the (possibly dynamically varying) preferences of the user. These characteristics make Web data transfer fundamentally different from the traditional bulk data transfer. The latter, to first order, is only concerned with the transfer of an undifferentiated, linear sequence of bytes.

### 1.3.2 Web Traffic Characteristics

We now discuss the characteristics of the traffic between Web servers and clients in aggregate, without being concerned with the internal structure of the traffic that we discussed in the previous section.

A user's Web interactions are often in the form of *browsing sessions*. During a browsing session, the user may download several pages from the same server. The presence of hyperlinks pointing to others pages residing on the server contributes to this locality of reference. There is often an idle period, called the *user think time*, between successive Web page downloads within a browsing session. This period corresponds to the time that the user spends perusing the Web pages retrieved previously or engaged in other activity. The length of the idle period typically ranges from a few seconds to several minutes; [65] reports a median user think time of 15 seconds.

As a result, Web traffic tends to be bursty. Each Web page download constitutes a burst. Individual Web pages tend to be small in size. Several studies have indicated that Web page sizes are typically on the order of a few tens of kilobytes; [65] reports an average page size of 26-32 KB and average component size of 8-10 KB. Therefore, the bursts tend to be intermediate in length compared to those in traditional applications — much longer than the 1-2 KB bursts in Telnet, but at the same time, much shorter than bulk transfers that may be hundreds of kilobytes long.

To minimize latency, it is important that a Web page transfer utilizes the available bandwidth effectively. However, the bursty nature of Web traffic makes this task difficult. We discuss this mismatch in Section 1.5, after outlining the TCP protocol in the next section.

## 1.4 Transmission Control Protocol (TCP)

TCP is the predominant transport protocol in the Internet. TCP builds on the service that IP provides of transporting data packets from a source host to a destination host. The distinguishing characteristic of the transport layer compared to layers beneath it in the stack is that it operates *end-to-end*, i.e., it involves peer-to-peer communication between processes at the communicating end-hosts. Before discussing TCP in detail, we outline the functions that the transport protocols may perform in general:

- *Demultiplexing*: A network level address, such as an *IP address*, only identifies a host. The transport layer could provide a mechanism to address individual processes on a host, and demultiplex incoming data packets to the correct recipients.
- *Reliable data delivery*: Packets may be dropped due to resource constraints on the network path from the source host to the destination host. Indeed, this is a deliberate design choice made in the IP service model [20]. The transport layer could perform *loss recovery* to mask packet drops from the higher layers.
- *In-order data delivery*: The network layer could deliver packets to the receiving host in a different order than they were sent in. The transport layer could again mask such reordering from the higher layers.
- *Flow control*: This is a mechanism to keep a sender from transmitting data packets faster than the receiver is able to consume them.
- *Congestion control*: This is a mechanism to keep a sender from overwhelming shared network resources (such as link bandwidth and router buffer space) by sending too fast.

TCP is a full-fledged transport protocol that performs *all* of the functions listed above. The fundamental abstraction in TCP is a *connection*, which provides reliable, ordered byte-stream service to a higher layer, i.e., a user of TCP. In this model, the sender transmits a sequence of bytes over a connection and *all* of those bytes are delivered to the receiver in the *same order* they were sent in. This simple abstraction for reliable communication greatly simplifies the task of an application developer. This has led to the adoption of TCP as the basis for a wide range of applications, including remote login, file transfer, and more recently, the Web.

TCP performs the functions listed above at the granularity of a connection. Each TCP connection transfers an ordered byte-stream independently of other connections. TCP provides no message or record boundaries within a connection. Each connection also performs congestion control and loss recovery independently of other connections. The tight coupling of the ordered-byte stream service abstraction with congestion control and loss recovery mechanisms leads to suboptimal performance for Web transfers, as we will discuss in Section 1.5.

Since TCP provides reliable data delivery on top of a network that could drop packets, it includes mechanisms to detect data loss and recover from it. The TCP receiver sends *acknowledgement* packets (*acks*) to the sender to indicate which packets it has received. The TCP sender buffers all unacknowledged data and infers data packet loss by observing acks. When it detects a loss, the sender retransmits the corresponding data from its buffer.

The TCP sender detects packet loss either based on acks it receives in response to data it had transmitted (*data-driven loss recovery*) or the absence of an ack for data sent well in the past (*timer-driven loss recovery*). In either case, loss recovery happens independently for each connection. Data-driven loss recovery is the more desirable alternative from the viewpoint of quick recovery because as discussed in [108], timers need to be set conservatively to contend with potential variability in feedback (i.e., ack) delay.

Data-driven loss recovery is more effective the greater the length of the data stream. Intuitively, each data packet provides the sender a means to probe the receiver because of the response it elicits in the form of an ack. The greater the number of probes, the easier it is for the sender to reliably determine which packets have reached the receiver and which have not.

Acks also play a critical role in TCP congestion control. Since the available network bandwidth is not known *a priori*, a TCP connection starts with a low data rate. As acks are received, indicating the successful delivery of data to the receiver, the connection transmits new data packets (*ack clocking*). It also ramps up its data rate (*slow start* [51]). The connection may eventually congest the network and cause packet loss. When a packet loss happens, the connection backs off by halving its data rate.

## 1.5 Mismatch Between the Web Application and the TCP Protocol

Having discussed the key characteristics of Web data and traffic, and the salient features of the TCP protocol, we now discuss the performance problems arising from their mismatch.

First, a Web page download involves multiple, logically-separate transfers between the server and the client. TCP in itself does not provide any means to demarcate sub-streams within a connection. So, with the HTTP/1.0 protocol [10], Web client and server applications map each transfer onto a separate TCP connection. Doing so provides a separate data stream for each transfer, which is desirable. However, the concurrent connections compete in an uncoordinated manner, which aggravates congestion and causes unpredictable performance for each connection. A connection carrying valuable data may perform worse than one carrying less important data. Furthermore, data-driven loss recovery suffers because each connection is typically short in length (only a fraction of the length of the entire Web page transfer). Therefore, the first challenge of Web data transport is:

***Challenge #1:*** *Supporting multiple, concurrent data streams between a server and a client efficiently.*

Second, it is difficult for Web transfers to utilize the available bandwidth effectively because of the bursty nature of Web traffic. At the onset of a burst, the sender is unaware of the level of load in the network, so it is forced to probe the network to discover how much bandwidth is available. But this probing requires time. In particular, TCP's slow start process may require several round trip times (RTTs) to ramp up the data rate to match the available bandwidth. As a result, the latency for the short burst increases and its bandwidth utilization suffers.

To avoid these problems, a connection may start with a high data rate. However, doing so increases the chance that the network will become congested, especially when several connections start with a high data rate simultaneously. Network congestion could lead to heavy packet loss and degraded performance for all connections. Clearly, this is not a desirable outcome.

Therefore, the second challenge of Web data transport is:

***Challenge #2:*** *Enabling short and bursty transfers to utilize the available network bandwidth effectively, without adversely impacting the stability of the network.*

Before we present our solutions to these challenges, we discuss a third challenge, which arises due to the characteristics of certain access network technologies.

## 1.6 Impact of Heterogeneous Access Network Technologies

Access networks provide connectivity to remote locations such as residences and small offices. Access network technologies span a broad spectrum, as summarized in Table 1.3. The traditional and still dominant technology is dialup phone lines. However, its extremely limited bandwidth has spurred the development and deployment of higher-speed alternatives such as cable modem networks. However, some of these new access network technologies interact poorly with TCP, and hence cause performance problems for Web transfers. We list below the key characteristics of these networks and the performance problems these cause:

1. *Bandwidth asymmetry:* This arises in a variety of networks, including those based on ADSL, cable modem and, satellite technologies. The ratio of downstream to upstream bandwidth can range from 10 to 1000 depending on whether the network is inherently two-way, or requires a separate upstream channel such as a dialup phone line. Although Web browsing has asymmetric bandwidth requirements, bandwidth asymmetry could adversely impact the performance of Web downloads in the downstream direction. This is so because congestion in the upstream direction could disrupt the flow of acks, which is critical to sustaining good TCP throughput in the downstream direction. Performance is especially impacted when there is other concurrent traffic in the upstream direction.

| Access Technology | Physical Medium | Bandwidth | Latency |
|---|---|---|---|
| Dialup modem | PSTN | 28-56 Kbps D, 28-33 Kbps U | 50-100 ms |
| ISDN | PSTN | 64-128 Kbps | 10 ms |
| ADSL | PSTN | 1.5-9 Mbps D, 16-640 Kbps U | |
| 1-way cable modem | Coax cable/HFC | 10-30 Mbps D, 28-56 Kbps U | |
| 2-way cable modem | Hybrid fiber coax | 10-30 Mbps D, 0.096-4 Mbps U | |
| MMDS | Wireless (2.5 GHz) | 10-30 Mbps D, 28-56 Kbps U | |
| LMDS | Wireless (28 Ghz) | >100 Mbps | |
| CDPD | Wireless (850 Mhz) | 19.2 Kbps | |
| Packet radio | Wireless (915 MHz) | 30-40 Kbps | 400 ms |
| Satellite | Wireless (Ka, Ku) | 16 Kbps-60 Mbps D<br>16 Kbps-2 Mbps U | 20-400 ms |

**Table 1.3 A summary of various access network technologies. "D" and "U" refer to the downstream and upstream directions, respectively.**

2. *Large delay-bandwidth product:* This situation arises in particular in geostationary satellite networks, in which the latency is large (several hundred milliseconds) and so is the bandwidth (several Mbps). Because of TCP slow start, short Web transfers utilize the available bandwidth very poorly. A similar situation can arise even in highly asymmetric terrestrial networks when there is a bidirectional flow of traffic (Chapter 9).

3. *Constrained bandwidth:* At the other extreme are bandwidth-constrained access networks such as dialup modems and the mobile data networks with bandwidths of the order of a few tens of Kbps. In such networks, multiple simultaneous data streams of a Web transfer can cause severe congestion and degrade performance.

The trend towards extremely high-speed network backbones makes the access network a critical factor in end-to-end performance. This leads to our third challenge:

***Challenge #3:*** *Achieving high bandwidth utilization and low latency over a wide range of access network technologies, including asymmetric-bandwidth, large bandwidth-delay product and bandwidth constrained networks.*

## 1.7 Addressing the Challenges

In this thesis, we develop several new techniques to address the three challenges that we have identified in this chapter.

We first developed a new connection abstraction for HTTP, called *persistent-connection HTTP* (*P-HTTP*). The key ideas are to share a *persistent TCP connection* for the transfer of multiple Web page components and to *pipeline* the transfers of these components. By avoiding the competition among connections that afflicts HTTP/1.0, P-HTTP reduces latency significantly. The ideas in P-HTTP, developed by us in 1994 [85], have been adopted by the HTTP/1.1 protocol [30].

However, a fundamental drawback of P-HTTP is that the service abstraction of a TCP connection imposes strict ordering on Web transfers that correspond to possibly unrelated components of a Web page, as illustrated in Figure 1.4(a). This violates the *application-layer framing* (*ALF*) principle enunciated in [21] and can degrade user-perceived latency.

This drawback of P-HTTP led us to develop a comprehensive solution, which has two components. The first component, *TCP session,* was proposed by us in [82] and refined in [6]. TCP session decouples TCP's ordered byte-stream service abstraction from its congestion control and loss recovery mechanisms. It treats data belonging to the set of concurrent connections between a server and a client as a single, unified data stream for the purposes of congestion control and loss recovery, thereby *integrating* the latter mechanisms across the set of connections. However, the individual connections retain their independent byte-stream abstractions. *Thus, TCP session combines the flexibility of separate connections with the performance efficiency of a shared connection.* The integration of two TCP connections into a session is illustrated in Figure 1.4(b). TCP session decreases download time by up to a factor of ten compared to HTTP/1.0 layered on standard TCP.

TCP session has the important attribute that it is backward compatible with the existing TCP protocol. TCP session does not alter TCP's messaging semantics, so deployment only involves local

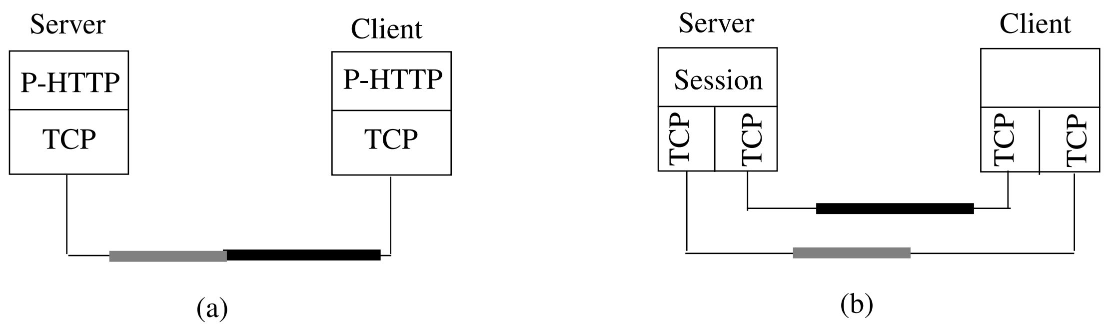



**Figure 1.4 The progress of two logically-separate Web transfers with (a) P-HTTP and (b) TCP session. P-HTTP imposes a linear order on the two transfers while TCP session does not. P-HTTP operates both at the client and the server whereas TCP session is confined to the server.**

changes at the sender. In the context of the Web, the senders are primarily the servers. And servers are probably easier to upgrade than the more numerous clients.

The second component of our solution is *TCP fast start*, proposed by us in [84]. The key idea is using *persistent state*. TCP fast start improves bandwidth utilization for short transfers by reusing information about the network conditions cached in the recent past. Thus, it can decrease the cost of probing for bandwidth significantly, especially in networks with a large bandwidth-delay product.

To avoid adverse effects in case the cached information is stale, TCP fast start exploits priority dropping at routers. By marking packets sent during fast start as low-priority, the sender minimizes the adverse impact that fast start could have on other connections in the network. TCP fast start also augments TCP's loss recovery mechanisms to quickly detect and abort a failed fast start attempt.

In addition to TCP session and TCP fast start, we have developed a set of end host-based and router-based techniques to improve performance substantially in asymmetric networks [5]. These techniques reduce both the disruption in the flow of receiver feedback (in the form) of acks and TCP's dependence on such feedback. In certain situations, these help decrease Web page download time by a factor of fifteen.

## 1.8 Design Principles

Before concluding this chapter, we discuss the principles that have influenced the design of the techniques developed in this thesis:

1. *Robustness*: While the techniques we design may be targeted to addressing performance problems that arise in specific situations (such as short Web transfers or asymmetric-bandwidth networks), they should not cause performance degradation in other situations or create new performance problems of their own.

2. *Wide applicability*: Our solution should be as widely applicable as possible across networks and applications. For instance, techniques designed to improve the performance of Web browsing should also help other applications, such as distributed transaction processing, that have similar communication requirements. In particular, it undesirable to have a solution that is tied to a specific application-level protocol, such as HTTP.

3. *End-to-end principle*: To the extent possible, our solution should only require the participation of end-hosts in the network, such as servers and clients. There should be minimal additional burden placed on routers in the interior of the network. This results in a scalable design because the additional work entailed in supporting a new set of hosts gets distributed across them. This principle, first formulated in [96], has been a cornerstone of the growth and success of the Internet.

4. *Incremental deployment*: It should be possible to deploy our solution incrementally since the size and the heterogeneity of the Internet preclude a global deployment in a short span of time. Even a partial deployment should yield significant performance benefits. In this context, a partial deployment could mean that only subset of the solution is deployed and/or its deployment is confined to a subset of the network.

We believe that these represent good engineering principles that have contributed to the success of the Internet. In Chapter 10, we evaluate how successful we have been in conforming to these principles.

## 1.9 Thesis Outline

The remainder of the thesis is organized as follows. In Chapter 2, we present background information on the TCP and HTTP protocols, and survey related work, pointing out its relationship with our work.

In Chapter 3, we discuss our research methodology. This chapter includes a discussion of our experimental network testbed, traffic traces, simulation and implementation environments, and performance evaluation metrics.

In Chapter 4, we present an analysis of HTTP performance. We develop a simple mathematical model for HTTP. We also analyze the interactions between HTTP and TCP using traffic traces from the 1996 Olympics Web server run by IBM.

This analysis leads into Chapter 5, where we introduce our initial solution, *persistent-connection HTTP* (*P-HTTP*). We discuss two key ideas, persistent connections and pipelining. We demonstrate the significant performance benefits of P-HTTP via experimental evaluation. We also point out its limitations, which set the stage for the comprehensive solution presented in Chapter 6 through Chapter 8.

In Chapter 6, we present *TCP session*. We discuss the three components of TCP session: integrated congestion control, connection scheduling, and integrated loss recovery. We discuss how TCP session combines the strengths of both HTTP/1.0 and P-HTTP, and present a detailed performance analysis.

In Chapter 7, we discuss some advanced issues pertaining to TCP session, including exposing connection scheduling to applications, and adapting integrated congestion control to environments with Web proxies.

We present *TCP fast start* in Chapter 8. We discuss the benefits and potential pitfalls of reusing cached information. We explain how TCP fast start avoids the pitfalls by exploiting priority dropping in routers, and by employing augmented loss recovery procedures. Our performance analysis includes experiments over the DirecPC satellite network [27].

In Chapter 9, we discuss a set of end-to-end and router-based techniques to address the problems arising in asymmetric networks. We present performance evaluation of both unidirectional and bidirectional traffic. Our analysis includes measurements over a cable modem network from Hybrid, Inc [47].

We conclude this thesis with a summary of our work and contributions in Chapter 10. We revisit the design principles listed in Section 1.8 and evaluate how successful we were in conforming to them. We also provide directions for future work.

In Appendix A we discuss the software tools we used for our analysis and evaluation. These tools include both existing ones and new ones that we developed.

In Appendix B, we discuss the implementation of the integrated loss recovery component of TCP session in detail.

# Chapter 2

# Background and Related Work

In this chapter, we present background information relevant to our work. We also survey related work and point out their relationship to our work. First, we discuss the TCP protocol. Then we discuss the HTTP protocol that underlies the World Wide Web, and point out performance problems that arise due to a mismatch between TCP and HTTP. Finally, we discuss how a variety of transport protocol techniques, developed both in the context of TCP and otherwise, are useful in some ways but fall short in others in addressing Web performance problems.

## 2.1 Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) [89,101] is the *de facto* standard protocol for unicast data transport in the Internet. TCP is used in a wide variety of applications and application-level protocols including the World-Wide Web (HTTP), file transfer (FTP) and electronic mail (SMTP).

TCP is a connection-oriented protocol. A TCP connection establishes a bidirectional data pipe between two processes, possibly on different hosts. The service abstraction provided is that of a reliable, ordered byte stream. Bytes are sent into the data pipe at one end and come out in the same order and without losses at the other end. To support this service abstraction, each byte of data transferred on a connection is assigned a unique sequence number, which is a 32-bit wide number.

The sequence number is incremented by one for each new byte. The *initial sequence number* (ISN) is picked at random at the time of connection setup.

Although a TCP connection transfers a continuous stream of bytes, in practice it does so in discrete units called *TCP segments*. A TCP segment consists of a sequence of data bytes with an associated TCP protocol *header*. Among other things, the header specifies the range of sequence numbers corresponding to the data contained in the segment and the port number of the connection at the source and destination hosts. The port numbers enable the receiving TCP layer to perform demultiplexing, i.e. separate segments based on the connection to which they belong. The source and destination host addresses are carried in a separate network layer (IP) header. A TCP connection is identified by the 4-tuple <*source IP address, source port number, destination IP address, destination port number*>. Note that although the TCP layer operates on TCP segments, it is entirely possible lower layers (such as the IP layer) to fragment and reassemble segments transparently to TCP.

To understand what influences the performance of a TCP connection, we discuss the core mechanisms in TCP including connection establishment, protocol control block, acknowledgements, loss recovery and congestion control algorithms, and TCP options. We note that the discussion here is largely confined to the "Reno" flavor of TCP which is in widespread use today. We discuss a variant of TCP Reno in Section 2.3.1.

### 2.1.1 TCP Connection Setup

Connection setup involves a three-way handshake between the sender and the receiver. The purpose of this handshake is to synchronize the communicating peers, so it is termed as the *SYN handshake*. The specific goals of the SYN handshake are:

1. The selection of the initial sequence number (ISN) for communication in each direction. ISNs must be chosen carefully to avoid any overlap with sequence numbers used in the recent past. An overlap could cause data from an old connection in the recent past to be mistaken for that belonging to the new connection, and thus violate TCP's reliability guarantees.

2. The negotiation of various TCP protocol options. These options include the maximum segment size (MSS), timestamp, selective acknowledgement (SACK), etc. We describe these in detail in Section 2.1.6.

The SYN handshake involves three steps. The initiator of the connection, host A, first sends a SYN packet to its peer, host B. Included in this SYN packet is the ISN for communication from host A to B. Upon receiving this SYN, host B picks an ISN for communication from host B to A, and includes this in the SYN packet it sends back. This packet also acknowledges the SYN sent from A to B. Finally, host A completes the three-way handshake by acknowledging the SYN sent by host B. Thus, the connection setup procedure consumes one network round-trip time (RTT) between A and B plus the one-way delay from A to B. However, A can start sending data to B concurrently with the final acknowledgement packet, so the effective start up delay is only one RTT.

### 2.1.2 TCP Protocol Control Block (TCB)

Given that TCP is a connection-oriented protocol, state must be maintained to manage the connection at the two communicating end points (*hosts*). The TCP protocol control block (TCB) is the repository of this state. Part of this state simply identifies the connection (e.g., source/destination *address* and source/destination *port number*) while the rest captures the dynamic aspects of the connection. We briefly describe some of the more important state variables:

- *sndnxt*: the sequence number of the next data byte to send
- *snduna*: the sequence number of the oldest unacknowledged data byte
- *cwnd*: the congestion window size
- *ssthresh*: the slow start threshold
- *srtt*: an exponentially smoothed estimate of the round trip time (RTT) between the sender and the receiver.
- *rttvar*: the mean deviation in RTT
- *rto*: the retransmission timeout value

We will explain these state variables in more depth in the discussion below.

### 2.1.3 TCP Acknowledgements

The Internet is a shared resource. The bandwidth of network links and buffers at routers are instances of shared resources. During times of high load, the network may be forced to drop packets due to insufficient resources. As such, the IP layer only provides *best-effort* service. It does not guarantee that all packets injected into the network are delivered to the destination. Since TCP provides a *reliable* byte-stream service, it must include mechanisms to detect and recover from packet loss.

Fundamental to loss detection and recovery is the TCP acknowledgement mechanism. The header of a TCP segment includes an acknowledgement field. This allows the receiver to inform the sender which data segments in a connection it has received. Thus the typical operation of a TCP transfer is as follows. The sender stores a copy of each data segment it sends out in a local buffer. As the segments are delivered successfully to the receiver, acknowledgements ("acks" for short) are sent back to the sender. When the sender receives an ack, it clears out the corresponding data from its local buffer. It also updates *snduna* to reflect the newly acknowledged data.

TCP acks are *cumulative*. A value of `X` in the acknowledgement field implies that the all data up to and including sequence number `X-1` has been delivered to the receiver. Recently, a *selective* acknowledgement (SACK) option has been standardized to complement the cumulative acknowledgement. We discuss this in more detail in Section 2.1.6.

Since TCP is a full-duplex protocol, each of the end points of a connection could send data to its peer. Acks for the data transfer in one direction could be carried as part of packets belonging to the data transfer in the opposite direction. This motivates the *delayed ack algorithm*. Rather than acknowledging incoming packets immediately, the receiver delays the ack in the hope that it can be piggybacked onto a data packet it sends back in the reverse direction. But delaying an ack too long may cause the sender to stall. The policy is to send one ack for every other data packet received or if an unacknowledged data packet has been held for more than a threshold period (typically 200 ms). However, in practice data transfer on a connection tends to be unidirectional, so acks are not usually piggybacked on to data packets. This is of significance in the context of asymmetric networks, which we discuss in Chapter 9.

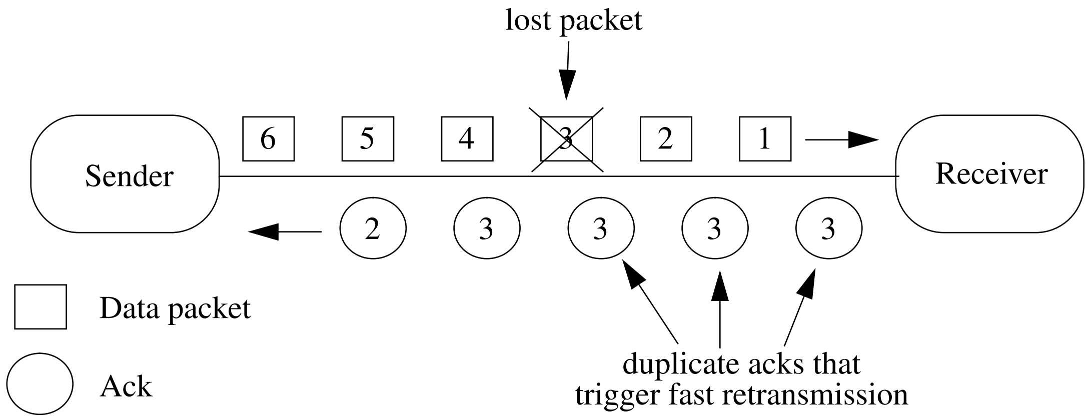



**Figure 2.1 Illustration of duplicate acks that trigger fast retransmission. We use packet sequence numbers rather than byte sequence numbers for ease of illustration. An ack carries the sequence number of the next packet expected in sequence.**

### 2.1.4 Loss Detection and Recovery

As mentioned above, mechanisms for loss detection and recovery are needed to support TCP's reliable data transport service. A TCP sender uses acks (or lack thereof) to detect losses. This can happen in one of two ways — *data-driven* or *timer-driven*.

Data-driven loss recovery, also known as *fast retransmission* in TCP, is based on the observation that successive segments within a TCP connection often get delivered to the receiver in the same order as they were sent. Whereas IP routing is dynamic, routes do not change very frequently, so successive segments tend to traverse the same route through the network and tend not to get reordered with respect to each other. Therefore, when a TCP receiver misses a packet on a connection but receives several packets that are later in sequence, it is highly likely that the missing packet was lost due to congestion in the network rather just reordered. If the starting sequence number of the missing packet is `X`, then the receipt of each packet with sequence number beyond `X` will cause the receiver to generate an ack with value `X`. These are termed as *duplicate acks* because they all convey identical acknowledgement information (Figure 2.1). When the sender sees more than a threshold (typically 3) number of duplicate acks, it infers that the next packet in sequence is lost. The use of a threshold larger than 1 guards against a small amount of reordering that may happen within the network.

Whenever a TCP sender has outstanding data that has not yet been acknowledged, it sets a timer known as the *retransmission timer*. If there is unacknowledged data at the time the timer expires, then the oldest packet among these is assumed to have been lost. The retransmission timeout value (RTO) is set large enough so that the chance of a premature timeout is small. The RTO is set taking into account both the mean round-trip time (RTT) between the sender and the receiver, and the variation in it. In most modern implementations of TCP, *RTO = mean RTT + 4 * mean deviation in RTT*. For reasons of efficiency in practical implementations, RTO is set at a coarse granularity, typically a multiple of 500 ms.

Data-driven loss recovery has a couple of advantages over timer-driven loss recovery. First, data-driven loss recovery does not entail any substantial idle period whereas timer-driven loss recovery, by its very nature, does and thus increases the latency and reduces the throughput of data transfer. Second, by not being dependent on timers, data-driven loss recovery avoids the difficult problem of setting timers correctly [108].

However, data-driven loss recovery requires a long data stream to be effective. That is, if there are not a sufficient number of data packets sent following the packet that was lost, an insufficient number of duplicate acks are produced to invoke fast retransmission. For instance, in Figure 2.1 if any of data packets 4, 5 and 6 had not been sent, an insufficient number of duplicate acks would be generated for the sender to reliably infer that packet 3 was dropped in the network. The sender would detect the loss of this packet only when its retransmission timer expires.

Once the sender has determined that a packet is lost, it attempts to recover from the loss by retransmitting a copy of the packet from its local buffer. Since the loss of a packet is also indicative of congestion in the network, the sender also invokes congestion control action, as we discuss next.

### 2.1.5 Congestion Avoidance and Control

The goal of the TCP congestion avoidance and control algorithms, developed by Jacobson [51], is to maintain the network in a state in which it is operating efficiently. Congestion refers to a situation where the network is so heavily loaded that router buffers fill up causing packet drops. In short, the network does wasteful work forwarding packets which are dropped before reaching their final destination. Clearly, it is highly desirable to avoid such a situation. *Congestion avoidance* minimizes the chances that the network enters such a state. *Congestion control* rectifies the situation quickly if it ever does.

Central to TCP congestion avoidance is the *ack clocking* mechanism, which attempts to maintain an equilibrium in the number of packets that are outstanding in the network. A new packet is injected into the network when an ack for a previous packet is received, indicating that it has left the network. Besides maintaining equilibrium, another benefit of ack clocking is that it obviates the need for timers to clock out packets.

Another critical component of TCP congestion avoidance is the *slow start* algorithm. For slow start, the TCP sender maintains a dynamically varying congestion window (*cwnd*, for short) that determines the amount of outstanding data that the connection can have. When a new connection starts up, *cwnd* is set to one segment and slow start is initiated. Each time the acknowledgement for a packet is received, *cwnd* is incremented by one segment. Thus during slow start, *cwnd* grows exponentially, doubling each RTT. Beyond a threshold window size, called the slow start threshold (*ssthresh*), the sender enters the *linear phase* during which *cwnd* is incremented by one segment every RTT. Thus by growing *cwnd* in a gradual manner, the TCP sender tries to minimize the chances of pushing the network into a congested state.

The exact same procedure is repeated whenever a connection resumes activity after having been idle for longer than a threshold, typically one RTT or one RTO [54]. The rationale for this is two-fold. First, slow start is a convenient way to rejuvenate the ack clock after it has stalled due to a connection's inactivity. Second, network conditions can change drastically while a connection is idle. In such a case, the old value of *cwnd* would be invalid, so *cwnd* should be reset.

In spite of the congestion *avoidance* algorithms employed by TCP, the network could still enter a state of congestion. There are several reasons for this. First, the TCP sender keeps probing the network for more bandwidth by increasing its congestion window size. It detects congestion by first overdriving the network and then discovering packet loss (by way of missing acks). Second, since network resources such as link bandwidth and router buffers are shared, multiple connections in aggregate could lead to congestion. Thus, there is a need for a congestion *control* mechanism.

The goal of TCP congestion control is to alleviate congestion, when it arises, by decreasing the level of load substantially. The loss of a packet (which is detected using the algorithms described

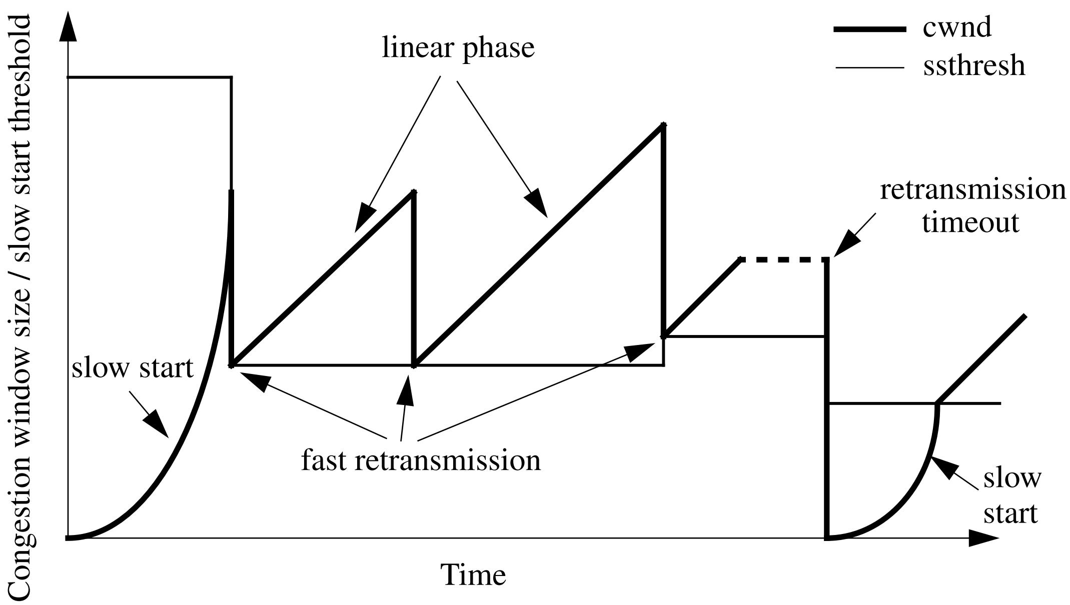



**Figure 2.2 The dynamics of the congestion window size (*cwnd*) and slow start threshold (*ssthresh*) of a TCP connection.**

in Section 2.1.4) is treated as a signal of congestion[1]. In response, the TCP sender sets its slow start threshold, *ssthresh*, to one half the size of the congestion window at the time the loss is detected. It also decreases the size of its congestion window, *cwnd*. In the case of fast retransmission, where the loss is detected in a data-driven manner, *cwnd* is halved. In the case of a retransmission timeout, *cwnd* is reset to one segment. The rationale for the more severe action in the latter case is that a retransmission timeout happens when fast retransmission was not triggered and is typically indicative of more severe congestion[2].

After *cwnd* is reduced during fast retransmission, *fast recovery* occurs. Here, the TCP sender gradually lets some of the packets, that have already been injected into the network, drain out until the amount of outstanding data is equal to the new (halved) value of *cwnd*. Beyond this point, a new data packet is injected each time a (duplicate) ack is received. This maintains the amount of out-

1. The underlying assumption is that packet loss is only caused by congestion. This is a reasonable assumption for most (wireline) networks; a notable exception is error-prone wireless networks.
2. With tail-drop gateways that are predominant in the Internet, it is a moot point whether the number of packets losses suffered by a connection (which often determines whether a retransmission timeout occurs) is at all correlated with the severity of congestion.

standing data in equilibrium. Fast recovery is terminated when a new (non-duplicate) ack is received and the linear phase of congestion avoidance is initiated.

In contrast, upon a retransmission timeout, *cwnd* is reset to one segment and slow start is initiated. The slow start process progresses only up to the new (reduced) *ssthresh*, beyond which the linear phase takes over. Thus a timeout slows down a connection more than fast retransmission does.

In summary, a TCP sender starts off with a small congestion window, which it then keeps growing. This eventually pushes the network beyond the brink, causing packet loss. When it discovers this (by way of missing acks), the sender cuts down its window size, thereby slowing down its sending rate. This cycle repeats.

### 2.1.6 TCP Options

TCP options are intended to enhance TCP functionality without requiring any changes to the core protocol itself. The options are negotiated by the two communicating peers, and come into effect only if both support it. We briefly describe some of the options of interest to us.

***Maximum segment size (MSS):*** This option allows the sender and the receiver to negotiate the maximum segment size for a connection. Each side usually requests the MSS to be the maximum transmission unit (MTU) of its local network minus the sizes of the IP and TCP headers. The smaller of the sizes requested by each side is chosen as the MSS for the connection. This eliminates the need for TCP segments to be fragmented and reassembled for transmission over the local networks at the sender and the receiver. However, avoiding fragmentation and reassembly altogether requires considering the MTU of all links along the path between the sender and the receiver. So a more elaborate procedure, such as path MTU discovery [76], would be needed to determine the appropriate MSS.

***Timestamp:*** The timestamp option [53] allows each of the communicating peers to include a timestamp in a TCP segment and have the timestamp echoed back in the corresponding ack. Thus when the timestamp option is enabled, each segment includes a timestamp and an echoed timestamp, each a 4-byte quantity. If the timestamp is, in fact, a measure of the wall clock time, it can be used by the sender to measure the round-trip time (RTT). Another use of the timestamp option is as an appendage to the 4-byte sequence number field to provide *protection against wrapped* *sequence numbers* (*PAWS*), which could otherwise cause old data segments to be mistaken for new ones, especially in high-speed networks. This requires the timestamp to be a monotonically increasing quantity which is incremented by 1 at least each time the sequence number wraps around. [53] recommends that it be incremented no more frequently than once every 1 ms and no less frequently than once a second.

***Selective acknowledgement (SACK):*** The SACK option [66] enables the receiver to convey more information to the sender about the received data than the standard TCP cumulative ack does. SACK overcomes the limitation of cumulative acks that the sender only has enough information to detect at most one packet loss per RTT. This is because a cumulative acks does not convey any information about packets beyond the last packet delivered to the receiver in sequence. This limitation means that the loss of multiple packets within a window could result in significant performance degradation. The SACK option enables the receiver to specify exactly which data it has received. This specification is in the form of zero or more (up to a maximum of 3 or 4) SACK blocks that are included in acks whenever there are gaps in the sequence of bytes received. Each SACK block specifies the starting and ending sequence numbers of a contiguous block of data received. The sender uses the SACK blocks to learn where the gaps are and then selectively retransmit the missing data.

### 2.1.7 Summary

In summary, TCP is a unicast transport protocol that provides a reliable, ordered byte-stream service. TCP loss detection and recovery happens either in a data-driven manner (the fast retransmission algorithm) or in a timer-driven manner (retransmission timeout). Data-driven loss recovery is more desirable but requires a longer data stream in order to be successful. TCP employs the slow start algorithm, both at connection start up time and when activity is resumed after an idle period, to gradually probe the network for bandwidth. In spite of this congestion avoidance procedure, congestion could happen. TCP responds to such situations by decreasing its congestion window size by half or more.

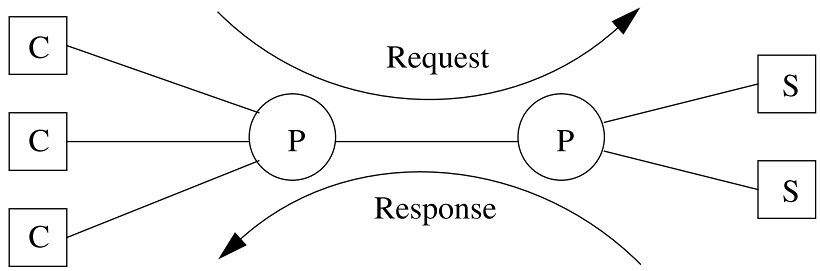



**Figure 2.3 A schematic of Web clients (C), proxies (P) and servers (S). Requests flow from clients to servers via proxies while responses flow in the opposite direction.**

## 2.2 World Wide Web

Having discussed the underlying transport protocol, TCP, in detail, we now turn to the application-level protocol that drives the Web. As we discussed in Chapter 1, the Web is a wide-area client server system. Servers are repositories of information and clients access this information.

In the Web, clients can either directly communicate with the server or do so indirectly via intermediate agents called *proxies*. In general, there could be many proxies along the communication chain between the client and the server, as illustrated in Figure 2.3. These serve as points of aggregation for requests from multiple clients and responses from multiple servers. However, at any one time, communication always happens via unicast between a pair of hosts, and the protocol used is identical to that used between clients and servers, namely HTTP.

### 2.2.1 Hypertext Transfer Protocol version 1.0 (HTTP/1.0)

HTTP is the application-level protocol underlying the Web. It defines the request-response interaction between clients and servers. HTTP/1.0 is the predominant version of the protocol today.

HTTP is largely a stateless protocol. There is no state carried over from one request-response interaction to the next. HTTP messages are passed in a format similar to MIME [11]. A request message from a client to a server typically contains the following:

- *Method:* specifies the action to be performed on the specified resource. Examples include GET (to retrieve a resource), HEAD (to retrieve just the corresponding meta-data) and POST (to annotate or append to a resource). GET is by far the most commonly used method in the Web.
- *Universal Resource Identifier (URI):* specifies the resource of interest. In practice, a URI can take several forms, including Universal Resource Locator (URL) [8] and Universal Resource Name (URN) [98]. A URI could specify a static resource such as an HTML or an image file, or a dynamic resource that is generated by a program (script) executed at the server.
- Other information such as the identity of the client program (e.g., a Web browser), an "if-modified-since" clause specifying a conditional GET, etc.

The response from the server to the client typically contains the following:

- *Status:* indicates the status of the request — success, failure, redirection, etc.
- *Entity:* represents the core of the response and consists of two parts:
  - *Entity header:* specifies meta-information pertaining to the entity body, including the content encoding, content length, content type, last modification time and expiration time.
  - *Entity body:* the content of the file requested or the output of the server-side script executed, encoded as specified in the entity header.

We note that the above description is far from complete. We have only indicated the more important contents of requests and replies in the "typical" case. In practice, there may be additional contents or fewer contents.

In general, HTTP can be layered on top of any transport protocol that provides reliable data delivery service. However, in practice, it is most commonly layered on top of TCP, the predominant transport protocol in the Internet. The timeline for a typical Web page download is shown in Figure 2.4. The client first sets up a new TCP connection to the server (involving a SYN exchange) and then sends a "GET" HTTP request ("REQ" in the figure) for the desired resource. Assuming the request succeeds, the server sends back its response ("REP") on the same TCP connection. Since TCP does not provide any mechanism to demarcate messages within a connection and since the initial designers of HTTP chose not to layer a framing protocol on top of TCP, the server closes

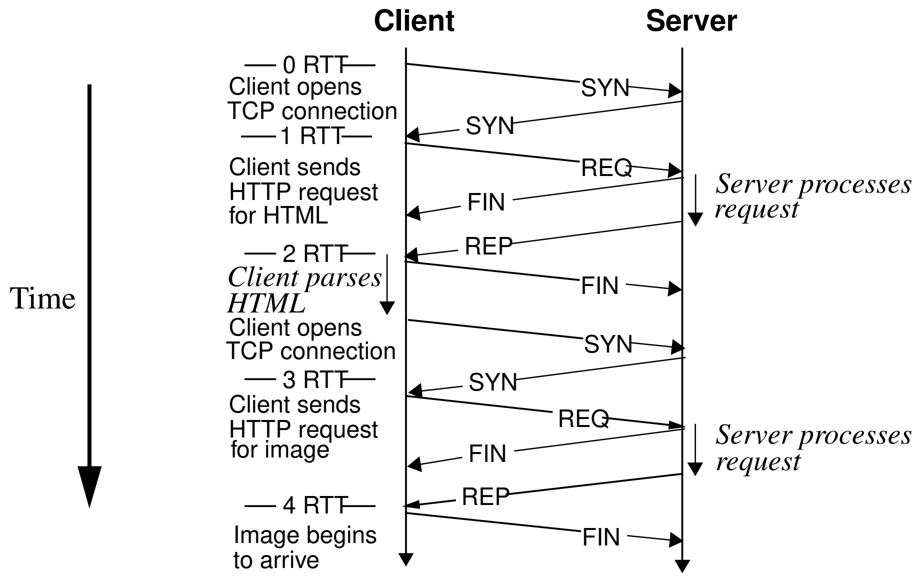



**Figure 2.4 The timeline of an HTTP/1.0 interaction between a client and a server showing the download of an HTML file and one inline image.**

the connection once it is done sending the entire response ("FIN"). In case more files need to be transferred (e.g., inlined images embedded within an HTML file), a separate TCP connection and a separate request-response exchange is used for each such transfer.

### 2.2.2 Performance Problems

Although the choice of TCP as the underlying transport protocol for HTTP is natural given the latter's prevalence in the Internet, their direct integration gives rise to several performance problems.

1. Connection establishment incurs at least one RTT of latency due to TCP's SYN handshake.

2. Invoking slow start when initiating the connection requires several RTTs to grow the window large enough for efficient operation. However, since Web transfers tend to be short in length, most connections terminate before the window grows to the optimal size, and thus network capacity is underutilized.

3. The connections comprising a Web transaction could be launched either sequentially or concurrently. Sequential connections add to latency, while concurrent connections could worsen congestion in the network (e.g., [32]).

4. When individual connections are short in length, the opportunity for data-driven loss recovery is diminished (as explained in Section 2.1.4). Therefore, timer-driven loss recovery dominates, which degrades connection throughput.

In addition, bandwidth asymmetry can affect TCP, and consequently, also HTTP.

5. Although Web traffic tends to be highly asymmetric, bandwidth asymmetry can still limit throughput. Because of ack clocking, a slow down or disruption in the flow of acks because of bandwidth constraints could in turn slow down or disrupt the flow of data in the opposite direction even though there may be no bandwidth constraint in that direction.

### 2.2.3 Summary

HTTP/1.0, while simple, leads to poor performance when directly integrated with TCP. Problems arise mainly because of the need for multiple connections and the short length of each connection. Our initial solution, *persistent-connection HTTP* (*P-HTTP*) (Chapter 5), eliminates the need for multiple connections by implementing a framing protocol on top of TCP. The new HTTP/1.1 protocol has adopted the main ideas of P-HTTP. However, this solution has its limitations, in particular ordering is unnecessarily imposed on independent messages carried over a TCP connection. To address these limitations, we develop new solutions, *TCP session* (Chapter 6 and Chapter 7) and *TCP fast start* (Chapter 8), that make the use of multiple connections, as in HTTP/1.0, more efficient.

## 2.3 TCP Enhancements

We now discuss various TCP enhancements that have been proposed in the literature. Where appropriate, we point out how these techniques help address some of the performance problems that arise in the context of the Web (Section 2.2.2). We note, however, that many of the techniques discussed below were not necessarily motivated by the Web. Indeed, some even predate the Web.

### 2.3.1 TCP NewReno

TCP "NewReno" [46] seeks to improve performance of the TCP fast recovery phase by making better use of acknowledgement information than TCP Reno. As noted in Section 2.1.4, the fast recovery immediately follows fast retransmission and continues until a new cumulative ack is received. However, fast recovery would be terminated even in the case of a *partial* new ack, i.e., one that does *not* acknowledge all data that was outstanding at the time fast recovery was initiated. A partial new ack provides very useful information because it indicates that the next packet in sequence was probably lost. Therefore, the sender could immediately retransmit the missing packet and continue to be in the fast recovery phase, and thereby avoid the penalty of a timeout. In general, this technique reduces the occurrence of timeouts when multiple packets are lost from within the same window.

### 2.3.2 Explicit Congestion Notification (ECN)

Our discussion of TCP congestion control assumed packet loss to be the only signal of congestion that a sender receives. This is an appropriate assumption in today's Internet because of the predominance of *drop-tail* routers. In the face of congestion when their queues fill up, such routers do no more than drop packets from the tail of the queue. Explicit congestion notification (ECN) is an alternative in which routers actively monitor their queue length to detect the onset of congestion and provide *explicit* feedback to the senders when this happens.

There are many different ways for ECN to be generated by routers and to be interpreted by senders. A particular scheme, that builds on previous work such as DECbit [92] and is gaining widespread acceptance, is *Random Early Detection* (*RED*) [33]. The RED algorithm involves the router computing an *exponentially weighted moving average* of its output queue length. When the average exceeds a threshold, packets are marked with ECN at random[3]. The probability of marking is tied to the average queue length, and becomes 100% beyond a threshold. Random marking of packets has the nice property that the probability that a particular connection receives an ECN is proportional to that connection's share of the link bandwidth. This property is ensured without incurring the overhead of maintaining per-connection state at the router.

TCP receivers echo back the ECN in acks that they transmit back to the sender. Upon receiving an ECN, the sender reacts just as it would in case of a packet loss (except that it does not retransmit

3. An alternative to marking packets is dropping them.

packets). *Cwnd* is halved and *ssthresh* is updated. Thus ECN is able to effect congestion control without letting router queues fill up and having packets dropped.

RED in useful in two contexts in our work. First, TCP fast start performs better with RED at the bottleneck router because the router queue tends to have more free space to hold packets sent during the fast start phase (Chapter 8). Second, RED feedback from the upstream router in an asymmetric-bandwidth network enables the receiver to perform *ack congestion control* to alleviate upstream congestion (Chapter 9).

### 2.3.3 TCP Vegas

TCP Vegas [15] presents an alternative to both packet drops and ECN as indicators of congestion. The underlying principle is Congestion Avoidance by Round-trip Delay (CARD) [55]. CARD observes that the queue length at the bottleneck router begins to grow when congestion occurs, and consequently that the RTT increases. Thus, a sender could infer that increased RTT implies congestion, and react accordingly. TCP Vegas extends CARD with a technique to maintain the amount of outstanding data in the network between a minimum threshold and a maximum threshold. However, this approach can be quite sensitive to noise in the RTT measurement process. Moreover, since Web connections are short, the sender might never have an adequate number of measurements to provide an accurate RTT estimate [14].

### 2.3.4 Transaction TCP (T/TCP)

While TCP NewReno and Vegas attempt to improve congestion control dynamics, Transaction TCP (T/TCP) [12] optimizes the connection-establishment phase to reduce the start-up penalty for short transactions. The novel mechanism in T/TCP is called *TCP accelerated open (TAO)*. TAO bypasses the three-way handshake procedure during connection setup (Section 2.1.1), and thereby saves one RTT of latency. To avoid compromising TCP's reliability in the process, T/TCP introduces the notion of a *connection count (CC)*. Each time a host initiates a connection to another host, it increments CC and includes it in the SYN packet. Each host also maintains a cache of CC values from other hosts that have initiated connections to it in the recent past. If the CC received in the SYN packet is larger than the cached CC for the host initiating the connection, connection establishment concludes without the three-way handshake. On the other hand, if the received CC is smaller or there is not a CC either in the SYN packet or in the cache, the standard three-way handshake procedure is used.

In addition to TAO, T/TCP caches many of the TCP state variables, including the RTT estimate and the MSS, to initialize the protocol control block for new connections. The T/TCP functional specification [13] suggests that the congestion window could also be cached, but does not specify how this should be done nor does it describe the trade-offs and dangers of doing so.

While TAO by itself enhances Web transport performance, the techniques proposed in this thesis are orthogonal to TAO and thus can be used in conjunction with it. Unlike our approach, TAO requires support at both the communicating hosts, i.e., both servers and clients in the context of the Web. Our techniques confine changes only to the sender.

### 2.3.5 TCP Protocol Control Block Interdependence

The TCB Interdependence proposal [103] extends the caching and initialization aspects of T/TCP to other state variables, including the congestion window. The TCBs of multiple connections between a pair of hosts are considered as interdependent to improve the start up performance of each. The interdependence could be *temporal*, i.e., between connections that are sequential in time, or *ensemble*, i.e., between connections that are concurrent. When a new connection is opened, state variables such as the RTT estimate and MSS are initialized with the corresponding values from the TCB of previous/concurrent connections, as the case may be. However, the sharing of congestion window is more complex. In the case of temporal sharing, it is initialized its old value. In the case of ensemble sharing, it is initialized to be `1/N` of the sum of the congestion windows of the `N` concurrent connections. The congestion window of each of the remaining concurrent connections is also adjusted to keep the sum constant. TCB sharing has the nice property that it only requires implementation changes at one end, namely the sender.

However, TCB sharing has its shortcomings. First, it does not specify a mechanism for sharing state beyond initialization. So there is still the possibility of competition among concurrent connections, such as those between a Web server and a client. Second, initializing the congestion window with a large size, as may happen with temporal sharing, could cause a large burst of packets to be sent, leading to heavy packet loss. Finally, [103] does not present an implementation or a performance evaluation of TCB sharing.

Our TCP session and TCP fast start techniques were developed independently of TCB sharing but share its basic philosophy. However, these techniques avoid the shortcomings of TCB sharing. *TCP session*, presented in Chapter 6 and Chapter 7, maintains a unified congestion window for the set of concurrent connections and employs a scheduler to explicitly allocate a share of the window to each connection. Thus it eliminates competition amongst concurrent connections. In addition, TCP session also integrates loss recovery across the connections. Our *TCP fast start* work, presented in Chapter 8, avoids the problem of burstiness by breaking up a large burst into several smaller bursts and using software timers to space them apart. In addition, TCP fast start explicitly evaluates and addresses the problem arising when TCP state from the past is stale.

### 2.3.6 Rate-Based Pacing (RBP)

Normally, when a connection resumes activity after an idle period, the congestion window is reset to 1 segment and slow start is initiated. The goal of rate-based pacing (RBP) [105] is to reduce this overhead. The basic idea is to reuse the value of the congestion window from before the idle period, but to smoothly *pace* out packets rather than burst them out back-to-back. RBP uses the TCP Vegas algorithm [15] to estimate the rate at which to pace packets out.

While RBP avoids burstiness, it is still an open-loop scheme where the source (initially) injects packets into the network without any intervening feedback. If network conditions change in such a manner that RBP is too aggressive, the resulting packet losses could cause several problems. First, the connection doing RBP may suffer worse performance than if it had just done standard slow start. Second, the packet losses could degrade the performance of other connections in the network. Finally, if there are multiple congested links along the path of a connection doing RBP, overall network throughput could suffer because packets on this connection may traverse one congested link (at the expense of other connections), but then get dropped short of their destination at another congested link.

Our *TCP fast start* technique includes mechanisms to minimize performance degradation both for the connection and the network as a whole under adverse conditions. This is substantiated by the performance evaluation presented in Chapter 8.

#### 2.3.6.1 TCP over Asymmetric Networks

A new dimension in TCP research recently has been the analysis of performance in asymmetric access networks, such as ones based on cable modem technology. Lakshman et al. [64] studied the dynamics of TCP connections in asymmetric networks using mathematical analysis and simulation. They define the *normalized asymmetry* as ratio of the bandwidths in the downstream and the upstream directions divided by the ratio of the sizes of packets (data or acks) flowing in each direction. They conclude that a large value of this ratio can have an adverse impact on TCP performance. To alleviate the performance problems, they suggest drop-from-front packet discard policy and weighted round-robin scheduling at the upstream router.

Kalampoukas et al. [58] studied the impact of two-way (i.e., bidirectional) TCP traffic over asymmetric access networks. They show that downstream throughput could suffer because TCP traffic in the upstream direction may cause excessive queueing of acks at the upstream router. They propose a backpressure mechanism to improve downstream throughput, but conclude that this improvement comes at the expense of upstream throughput.

In our work, presented in [5] and in Chapter 9, we studied the impact of asymmetry on TCP performance, both for bulk transfers and short Web transfers, via simulation and experiments over real networks. We designed and evaluated new techniques, both end-to-end as well as router-based, to improve performance in the upstream as well as the downstream direction.

## 2.4 Other Transport Protocols

In this section, we briefly survey a few unicast transport protocols other than TCP. We identify specific mechanisms in these that are relevant to this thesis.

### 2.4.1 NETBLT

The NETBLT (Network Bulk Transfer) protocol [22] is intended for high-throughput bulk data transfer. The key ideas in NETBLT are:

1. The mechanisms for flow control and loss recovery are kept separate unlike in TCP.
2. Rate-based flow control is used in addition to a TCP-like window mechanism.
3. Loss recovery is driven by timers at the receiver.

In NETBLT, packets are clustered together into *buffers,* and the sender paces out packets within a buffer at a negotiated rate. The receiver sends back a single acknowledgement for all the packets in a buffer. It detects missing packets by setting a timer corresponding to the size of the buffer and the rate of transmission. If any packets in the buffer are missing when the timer expires, the receiver generates back a selective acknowledgement identifying the missing packets. In response, the sender retransmits the indicated packets.

Rate-based data transmission has the advantage that the sender is not dependent on regular feedback from the receiver. We incorporate rate-based data transmission into TCP to adapt to situations where the receiver feedback is irregular (as in asymmetric networks) or unavailable (as in TCP fast start).

### 2.4.2 Packet-Pair Bandwidth Probing

Packet-pair [61] is a technique that a host could use to quickly discover its fair share of the network bandwidth. In this approach, the sender transmits a pair of packets back-to-back and measures the time gap between the corresponding acks. If the router at the bottleneck link uses the fair queuing scheduling algorithm [23], the back-to-back packets would get spaced apart according to the connection's share of the bottleneck link bandwidth. The acks then preserve this spacing provided they do not encounter variable queuing delays on the return path.

Packet-pair probing could be used to initialize state in TCP fast start. However, packet-pair probing has limitations that impede its use. First, it depends on fair queuing at routers inside the network, but most routers in the Internet implement FIFO scheduling. Fair queueing is expensive, especially as the number of active connections at the bottleneck link becomes large. Second, in the context of TCP, probing may require more than two packets to be sent back-to-back because a receiver employing the TCP delayed ack algorithm would send only one ack in response to a pair of packets. Finally, packet pair probing consumes at least one RTT, which may be significant when transfers are short in length and/or the RTT is large.

[46] presents a technique similar to packet-pair in the context of TCP, to determine an appropriate initial value for *ssthresh*. Standard slow start is still used for bandwidth probing, so the associated overhead is still there. But on the positive side, even an inaccurate initial setting for *ssthresh* (because of FIFO rather than fair queuing) is no worse than the arbitrarily large initial setting that is used by default.

## 2.5 Summary

In this chapter, we presented an overview of TCP and HTTP, and argued that their blind integration results in several performance problems. These problems include:

- The overhead of setting up and tearing down several short connections
- The slow start penalty that each connection incurs.
- Competition among multiple simultaneous connections between a server and a client.
- Fewer opportunities for data-driven loss recovery given the short length of each connection.
- Performance degradation in asymmetric networks due to the ack bottleneck.

Techniques proposed in the literature mitigate some of these problems, but they leave many problems unaddressed. In this thesis, we develop a set of techniques that address all of the performance issues mentioned above. Some of the ideas proposed in the literature are orthogonal to our techniques, and so can be used in conjunction with our techniques to derive greater benefit.

In the next chapter, we discuss our research methodology.

# Chapter 3

# Methodology

In this chapter, we describe our research methodology. Given the wide diversity of networks and dynamically-varying traffic patterns in the Internet, it is difficult to conduct networking research in a limited environment and yet claim with a high degree of confidence that the results are generally applicable. Therefore, as we will discuss next, we use a broad-based methodology to maximize the confidence in our research results.

We begin with an overview of our methodology. In Section 3.1, we describe our experimental testbed containing a heterogeneous collection of access networks. This provides use In Section 3.2, we discuss the traffic traces we gathered and analyzed. Section 3.3 discusses the network simulator we used to evaluate our algorithms. Finally, we discuss our implementation platform in Section 3.4.

Since this thesis addresses the challenges of Web data transport, it is natural that the first step should be to identify these challenges. There are two ways in which we do this. The first was to characterize performance by conducting live experiments across the Internet and over a variety of access networks in our experimental testbed (Section 3.1). The second was to collect and analyze traffic traces at a busy Web server (Section 3.2).

Once we had identified performance problems, we designed new algorithms to overcome them. During this phase, we conducted simulation experiments to explore the design space (Section 3.3). We followed this up with an implementation of the algorithms on an actual network platform Section 3.4). This enabled us to identify, and if necessary re-design, any features of the algorithms that made implementation difficult. Finally, we evaluated the algorithms via extensive experiments in the simulator as well as our network testbed (Section 3.5).

We used a number of software tools, both existing ones and new ones that we developed, to facilitate our analysis and evaluation. These are described in Appendix A.

## 3.1 Experimental Network Testbed

We used a heterogeneous network testbed for our experiments. The goal of these experiments was to analyze the dynamics and interaction of the TCP and the HTTP protocols in detail, and to identify performance problems that arise. Our testbed enabled us to conduct controlled experiments with real implementations of these protocols. The specific network characteristics of interest in our investigation (and the networks with those characteristics in our testbed) include:

1. Large delay and/or large delay-bandwidth product (DirecPC satellite network).

2. Bandwidth asymmetry (Hybrid wireless cable modem network).

3. Low bandwidth (dialup phone line, Ricochet packet radio network).

The networks in our testbed are examples of emerging access network technologies outlined in Section 1.6. We briefly describe the key characteristics and the architecture of each network.

### 3.1.1 DirecPC Satellite Network

DirecPC direct broadcast satellite (DBS) network [27] is an example of a satellite-based network dedicated to providing high-speed data access to residential and commercial users alike. The primary advantage of a satellite-based network is its geographical reach and the ease with which connectivity can be provided to a new location once the satellite system is already in place.

Figure 3.1 depicts the architecture of the DirecPC DBS system, which provides *asymmetric* network connectivity to users. The system is centered around a geostationary satellite that receives

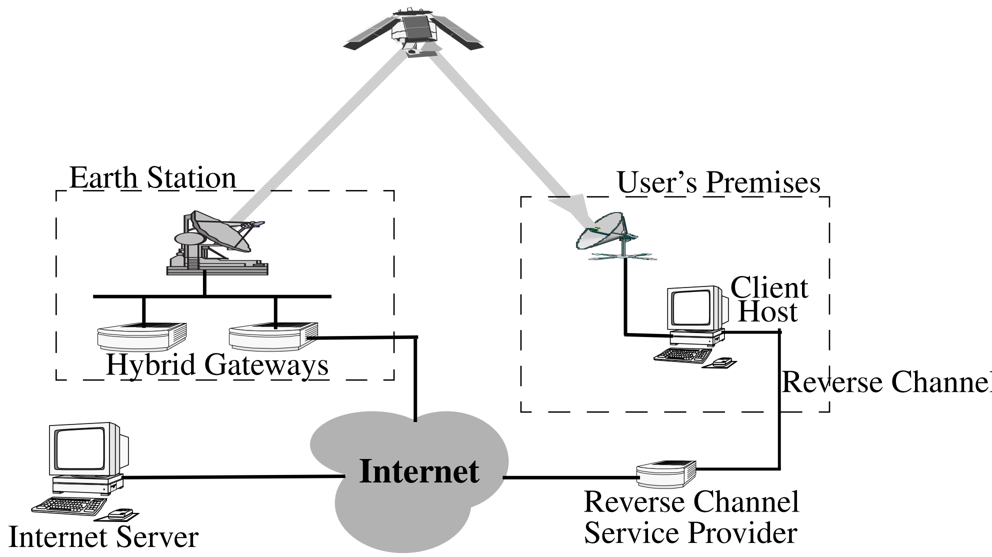



**Figure 3.1 The architecture of the DirecPC system.**

data from an earth station and broadcasts it *directly* to the users' premises. The bandwidth of this downstream channel is about 12 Mbps and it is shared among all users in a large geographical region via packet-level multiplexing. The advertised bandwidth per client is 400 Kbps, but the a client's bandwidth share could exceed this or fall short of this depending on the total number of clients that are active.

Connectivity in the reverse (upstream) direction is provided by another means, such as a dialup phone line or a terrestrial wireless network. So the data rate of the upstream link tends to be much lower, typically 15-30 Kbps, resulting in a high degree of bandwidth asymmetry.

The *hybrid gateway*, located at the earth station, mediates between the client hosts and other Internet hosts. It participates in routing data packets to and from each client host in an asymmetric manner [83]. A client host tunnel outgoing packets over its upstream link to the hybrid gateway, which then forwards them on to the actual destination host. Incoming packets specify the client host's address in the DirecPC domain as the destination, so get routed via the satellite link automatically.

The geostationary satellite link has a large delay of approximately 240 ms. Any communication between a client host and a server host in the Internet involves additional delays due to the upstream link and the Internet path leading to the server. So in practice the round-trip time for a TCP connection via the DirecPC network tends to be in the 300-400 ms range. This large delay

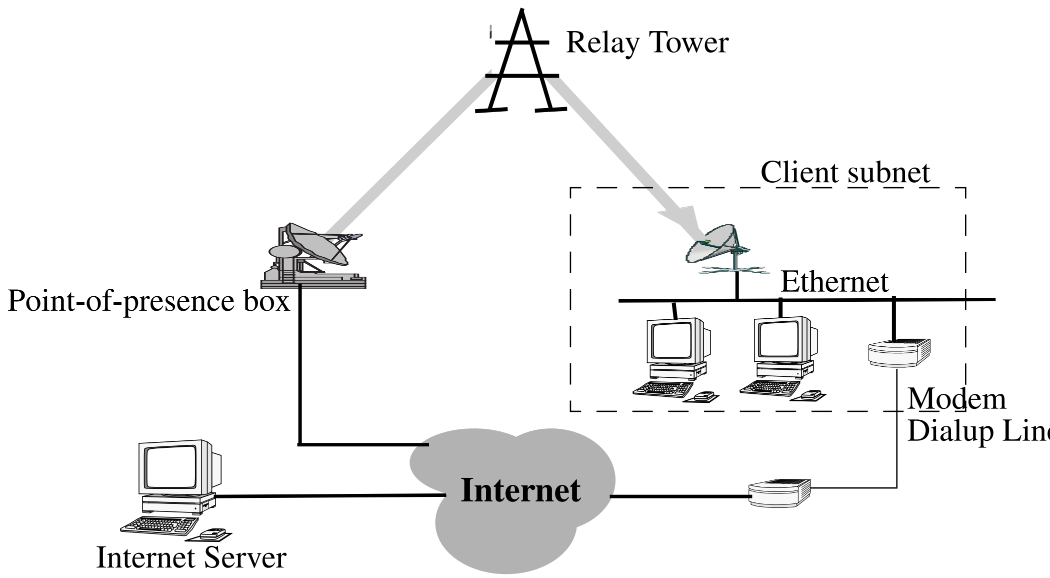



**Figure 3.2 The architecture of the wireless cable modem network.**

coupled with a relatively high bandwidth results in a large bandwidth-delay product, i.e., a large capacity "data pipe". Web transfers, which tend to be short in length, are unable to utilize this capacity effectively because of the slow start procedure. In Chapter 8, we present our *TCP fast start* technique to improve utilization, especially in networks such as this, by avoiding slow start when possible.

### 3.1.2 Hybrid Wireless Cable Modem Network

The "wireless cable" modem network from Hybrid, Inc. [47] is a high-speed access network designed to provide connectivity across a metropolitan area. Its architecture, depicted in Figure 3.2, is quite similar to that of the DirecPC network except on a smaller scale. A terrestrial transmission tower takes the place of the satellite and a point-of-presence (PoP) box that of the hybrid gateway.

Like the DirecPC network, this network is also asymmetric. It uses a multi-point multi--channel distribution (MMDS) link operating in the 2.4 GHz band to provide a 10 Mbps downstream broadcast channel. Upstream connectivity is provided via another means such as a dialup phone line. The delay of the wireless downlink is less than 5 ms, so the delay of the upstream link and that of the Internet tend to dominate.

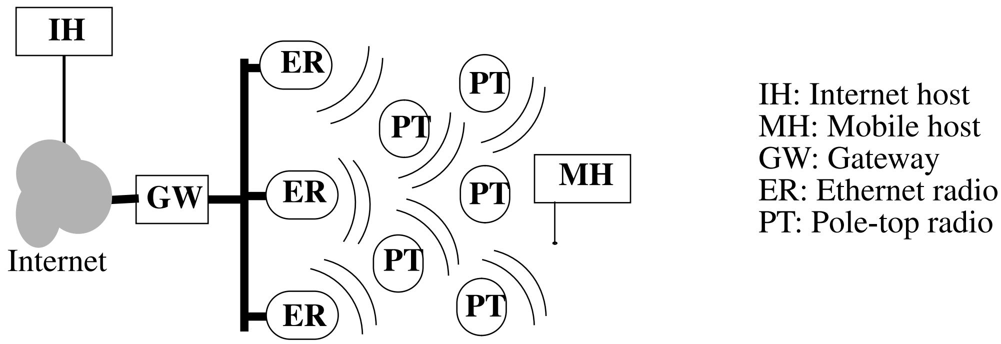



**Figure 3.3 Topology of the Ricochet packet radio network.**

Although the downlink is wireless, it is almost error-free owing to the fixed locations of the transmitter and the receiver while active[1]. So the link being wireless has little impact on the performance of TCP. The more important factor impacting performance is the high degree of asymmetry. This makes the upstream link prone to congestion which can degrade downstream throughput even when there is no congestion in that direction. We discuss this problem and develop solutions for it in Chapter 9. Given that it is asymmetry and not the wireless connectivity that impacts performance, we believe that our results extend to the (far more common) wireline one-way cable modem-based networks as well.

### 3.1.3 Ricochet Packet Radio Network

Another network in our testbed is the Ricochet packet radio network from Metricom, Inc. [71]. This is a wireless data network that provides connectivity to mobile users, much like the Cellular Digital Packet Data (CDPD) [17] and Global System for Mobile Communications (GSM) [42] networks.

The Ricochet network model is shown in Figure 3.3. It consists of *portable* radios that are carried around by users and fixed-location *pole-top* radios. A subset of the pole-top radios, called *ethernet* radios, have a wireline network connection that gateways into the Internet. Communication between a mobile host (connected to a portable radio) and a host on the Internet usually requires traversing multiple wireless hops (in addition to the wireline path through the Internet).

1. It is fairly easy to relocate the receiver, but doing so would lead to temporary network disconnection.

The key characteristics of the Ricochet network that is of interest to us are its low data rate and its large and variable latency. Although raw data rate of the channel is 100 Kbps, the effective data rate is only around 30-40 Kbps because of multiple hops in the wireless network and other sources of overhead. The one-way delay through the wireless network is of the order of 150-200 ms. But at times this can be a factor of 2-4 larger because of media access contention.

The low data rate of the network makes it prone to congestion. In Chapter 6, we discuss how our TCP session technique can alleviate congestion and improve throughput. The media access contention is aggravated when the flow of traffic is bidirectional, for example, TCP data packets flowing in one direction and acks in the other. As we showed in joint work with Balakrishnan [5], the techniques that help alleviate the effects of bandwidth asymmetry are also effective in this case. We discuss this briefly in Chapter 9. A more detailed investigation is the subject of a contemporaneous thesis by Balakrishnan [4].

## 3.2 Web Server Traffic Trace Analysis

In addition to analyzing performance via experiments in our testbed, we analyzed a large packet-level traffic trace. The traces were from the official Web server for the Summer 1996 Olympic Games [50]. This server was run by IBM. We analyzed:

1. The effectiveness of TCP loss recovery in the face of short and bursty Web connections.
2. The impact of multiple concurrent connections between the server and a client on TCP congestion control.

We describe the trace collection and post-processing steps here, and present details of the analysis in Section 4.3.

### 3.2.1 The IBM 1996 Olympics Web Server

The Olympics Web server experienced very high levels of traffic load and connection rates, and thus made for an ideal case study. To cope with the high level of load, the Web server was mirrored at four locations around the world: Southbury (CT, USA), Cornell (NY, USA), Keio (Japan), and Karlsruhe (Germany). In spite of the mirroring, the primary server site in Southbury experienced nearly 17 million Web hits on the most busy day.

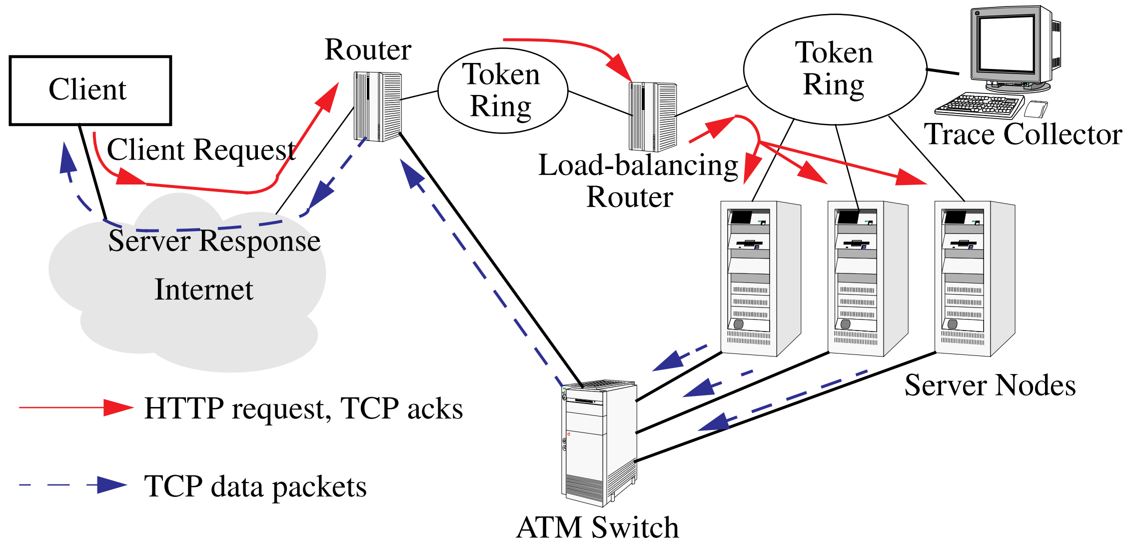



**Figure 3.4 The Web server and monitoring setup at the Southbury, CT site.**

The architecture and the network topology of the Web server at the primary site in Southbury are shown in Figure 3.4. During the Olympics, the Web site was connected via a T3 link (45 Mbps) to four U.S. Network Access Points (NAPs), located in Chicago (Bellcore and Ameritech), the San Francisco Bay Area (Bellcore and Pacific Bell), New York (Sprint), and Washington, D.C. (MFS Datanet). Requests for data from a client arrived at the Web site after being routed through the appropriate Internet NAP. Each request was subsequently redirected to a load-balancing connection router [48], that distributed them across several server nodes. These nodes then retrieved the appropriate Web objects and transmitted them across an internal ATM network and through the Internet to the clients. More details about the site structure and software are available from [49].

### 3.2.2 Data Collection Methodology

The primary Web server site was instrumented by Srinivasan Seshan and Mark Stemm to obtain packet-level traffic traces. All the traffic coming into the site was traced by running the *tcpdump* tool on a machine placed on the token ring connecting the load-balancing router to the server nodes (Figure 3.4). This machine was an IBM 150Mhz Pentium Pro PC running BSD/OS 2.1 from BSDI [16]. The first 350 bytes of every packet destined to TCP port 80 (the default HTTP port) were captured in the trace. This was then compressed on-the-fly using the *gzip* utility and periodically dumped to tape. Although the tracing system did not capture traffic (such as streaming audio) on ports other than the default HTTP port, during peak periods it still collected approximately 1 GB of compressed packet headers per hour.

Due to the location of the trace collection machine and the asymmetric path taken by the request and response streams, only packets coming from the clients *into* the Web server complex could be captured. These included the TCP SYN packet from clients at the time of connection setup, the HTTP request packets, and all TCP acknowledgments from clients in response to data sent by the server. Although none of the packets sent by the server to clients were traced, we were able to infer the approximate behavior of the outgoing packets since much of a TCP sender's activity is triggered by incoming ack packets (or the lack of such packets). To facilitate validation of such a reconstruction, we instrumented the TCP stack on a subset of the server nodes to trace the congestion window size, smoothed round-trip time estimate, and number of duplicate acknowledgments whenever a retransmission happened. We discuss the reconstruction procedure in some detail next.

### 3.2.3 Reconstructing the TCP Packet Trace

Reconstructing the entire TCP packet trace from just a trace of the incoming packets requires a way to faithfully reproduce the state of the sender for each TCP connection. For this purpose, we wrote a *TCP emulation engine* that took the trace of acknowledgments and other incoming packets as input and approximated the evolution of the sender's state variables (such as the congestion window size, the round trip time estimate, maximum packet transmission sequence number, etc.). As we explain below, this is possible to do because changes in sender-side state variables are triggered either by the reception of an ack or the expiration of a timer.

In essence, the engine uses the ack trace to emulate a subset of the functions that a TCP sender performs while an actual connection is in progress. Armed with the knowledge of the algorithms that the sender employs, it is possible to reconstruct the sender's behavior with complete accuracy if the evolution of the sender's state were completely driven by the arrival of acks[2] (as is usually the case if a connection experiences no losses). However, in practice packet losses do happen, sometimes causing the sender to time out. But it is possible to use the ack trace to infer the occurrence of such events as well. Doing so with our trace involved two steps. First, during times when there

2. This also assumes that the server always has data ready to send.

was no packet loss, the ack trace together with the reconstructed data packet trace was used to obtain an estimate of the round trip time and its variance. These were then used to compute the sender's retransmission timeout value (RT0). Second, a retransmission timeout event was flagged whenever more than RTO time had elapsed since the (inferred) transmission of a data packet but no ack had been received for it yet.

We validated the TCP emulation engine by comparing the times at which it predicted a retransmission event with the actual times at which retransmissions occurred, as reported by the server nodes with instrumented network stacks. This comparison revealed that the engine predicts the number of coarse timeouts, fast retransmissions, and slow-start retransmissions to within 0.75% of that recorded by the instrumented network stack. Consequently, the congestion window size computed by the engine is also highly accurate. The only other potential source of error is underlying loss rate in the packet capture process, but this was only about 0.5%.

In summary, our TCP emulation engine reconstructs the evolution of the TCP sender state for each connection between the Web server and clients, with a high degree of accuracy. This facilitates the analysis we present in Section 4.3.

## 3.3 Simulation Study

Our experimental and trace-based analyses helped us identify performance problems that afflict the TCP and HTTP protocols. In the next phase of our work, we designed new algorithms to address these problems. The algorithms include:

- Persistent-connection HTTP (P-HTTP)
- TCP session
- TCP fast start
- TCP for asymmetric networks

To aid in exploring the design space, and the process of tuning and evaluating new algorithms, we used the *ns* network simulator [81]. This is a packet-level discrete-event simulator that enables detailed simulation of a variety of network protocols using complex network topologies. The simulator comprises two components — a core written in the C++ language and a set of scripts written

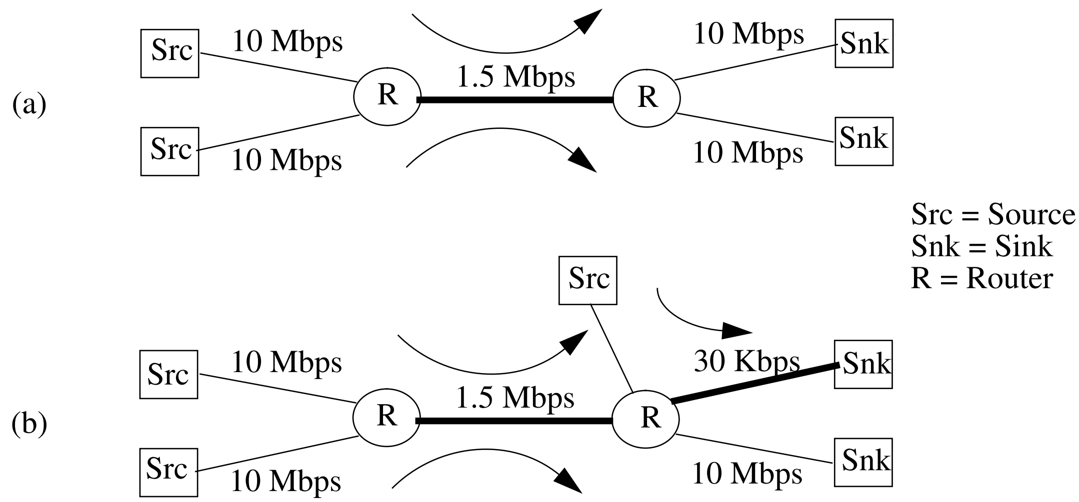



**Figure 3.5 Example simulation topologies with (a) a single bottleneck link, and (b) two bottleneck links (shown as thick lines).**

in Object Tcl (OTcl) [106]. The C++ core includes modules for a variety of protocols, including several flavors of TCP (Tahoe, Reno, NewReno, etc.). OTcl scripts are used to compose these modules and create simulation topologies.

We extended the C++ core with modules implementing the new algorithms we developed. These include TCP session, TCP fast start and asymmetric TCP. We also implemented link scheduling and buffer management disciplines for use in conjunction with these end-to-end algorithms.

We built a simple Web client and server in OTcl. These use the TCP modules in the C++ core to establish connections for the requests and responses. The client and server can be configured to emulate the HTTP/1.0 protocol, either with sequential or with concurrent connections, or P-HTTP.

The workload we used for simulation experiments was a mix of bursty Web-like traffic and bulk transfers. We used a variety of network topologies. Some of the topologies were patterned after the DirecPC satellite network and the Hybrid wireless cable modem network in our testbed. In addition, we uses simple network topologies with one or more bottleneck links, as illustrated in Figure 3.5. These topologies serve as useful abstractions of complex Internet paths since they capture the essential determinants of performance — the bandwidth and buffer size of the bottleneck link(s), and the end-to-end round trip time.

## 3.4 Implementation

After refining our new algorithms via simulation, we implemented them in an actual network protocol stack. Our object in doing so was two-fold:

1. To determine the changes, if any, needed to the algorithms to make them suitable for a real implementation.

2. To evaluate the algorithms in our network testbed and identify shortcomings, if any.

Many of our algorithms, including TCP session, TCP fast start and asymmetric TCP, involve changes to the TCP/IP protocol stack. Since the BSD protocol stack is well-documented and has influenced several other protocol stack implementations, we chose BSD/OS 3.0 [16] running on Pentium-based PCs as our software/hardware platform.

Some of our algorithms, for instance P-HTTP, only involve application-level changes in the Web client and server programs. So the choice of platform is not very critical. As we discuss in Chapter 5, our initial implementation was done on the Digital Ultrix and Digital UNIX platforms [26], while a more recent (and different) implementation was done on the BS/OS 3.0 platform.

We present details of our implementation in Chapter 5 through Chapter 9.

## 3.5 Evaluation

To evaluate our algorithms, we conducted experiments both in the simulator and over real networks. Experiments in the simulator gave us the opportunity to have complete control over the network and to scale up the load on it at will. Experiments over real networks enabled us to validate our simulation results.

The performance metrics we used in our evaluation include:

1. *Throughput*: the amount of useful data transferred in a unit time.

2. *Latency*: the time to complete the transfer of a given size.

3. *Fairness*: the extent of similarity across several instances of a particular metric (e.g., the latency experienced by a set of Web clients). We use the *fairness index* defined in [56] as

$$f = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \sum_{i=1}^{n} x_i^2}$$

where $x_1, x_2, ..., x_n$ are $n$ instances of the metric of interest.

4. Other metrics, such as the number of packet losses, retransmission timeouts and unnecessary retransmissions, that are of interest in specific cases.

## 3.6 Summary

In this chapter we described our research methodology. The first phase is experimental analysis. We described our experimental testbed and our setup to gather packet traces. The second phase involves designing new algorithms and refining them via simulation. We presented an outline of our simulation environment. The final phase is implementation and performance evaluation. We discussed our implementation platform and evaluation metrics.

In the next chapter, we present an analysis of HTTP performance.

# Chapter 4

# Analysis of HTTP Performance

In this chapter, we present a performance analysis of the Hypertext Transfer Protocol (HTTP), the application-level protocol underlying the Web. We analyze HTTP/1.0, the predominant version of the protocol in use today[1]. Given our interest in identifying performance problems, we analyze the interaction between HTTP and the TCP's algorithms for connection establishment, congestion control, and loss recovery.

We start with a description in Section 4.1 of the two ways in which HTTP/1.0 uses TCP connections: sequentially or concurrently. In Section 4.2, we present a performance analysis based on a simple mathematical model of HTTP as well as live experiments over the Internet. In Section 4.3, we analyze TCP connection traces obtained from IBM's 1996 Olympics Web server. We conclude with a summary in Section 4.4.

## 4.1 HTTP/1.0 Basics

As discussed in Section 2.2, a Web page download involves the transfer of multiple components such as the HTML file and the inline images. HTTP/1.0 uses a separate TCP connection for each

1. The new version, HTTP/1.1, is derived in part from our P-HTTP work. We discuss these in Chapter 5.

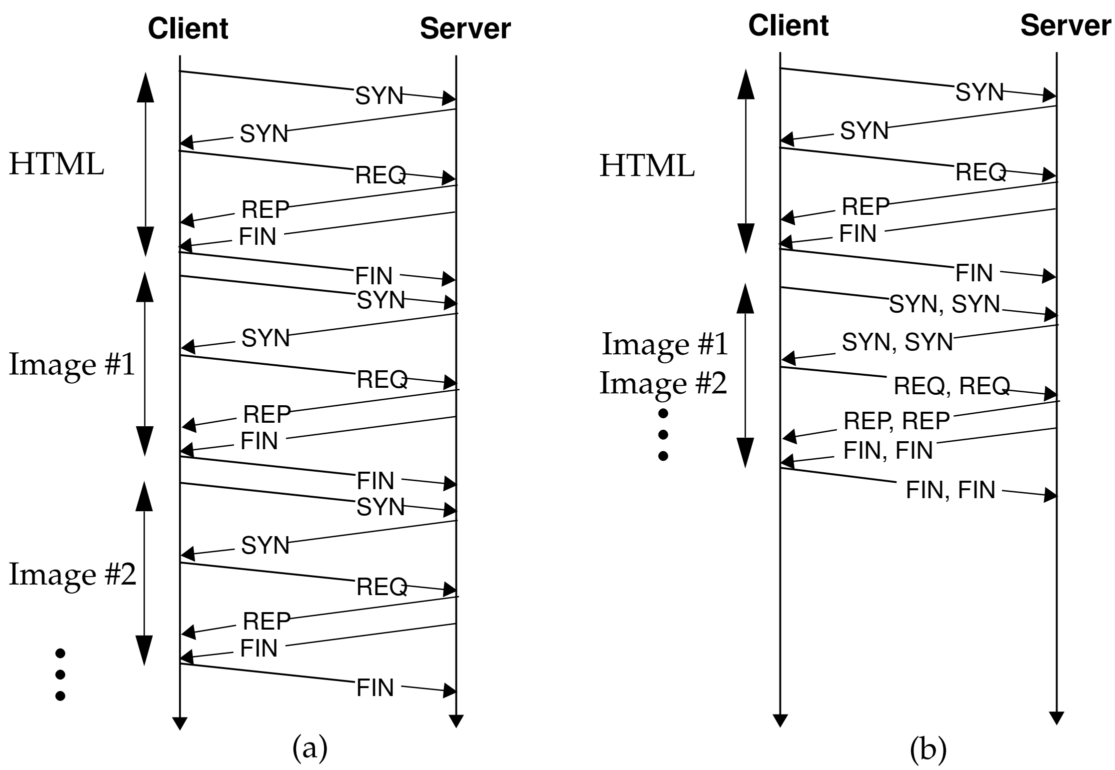



**Figure 4.1 Timeline of a Web page download with (a) sequential connections, and (b) concurrent connections. Note that although we show the concurrent connections in (b) as simultaneous, in practice there would be a time gap between the packets of the individual connections.**

component, in large part because TCP in itself does not provide a mechanism for demarcating multiple messages within a single TCP connection. The only mechanism that TCP provides for a sender to mark the end of a message is closing the corresponding TCP connection.

The individual components of a Web page can be retrieved either sequentially or concurrently. We discuss each possibility in turn.

### 4.1.1 Sequential Retrieval

The transmission of the components of a Web page from a server to a client is ordered sequentially. The client establishes a new connection and sends the request for a component only after it has received the previous component in its entirety. For a page with several inline images, the order in which the components are retrieved is usually the HTML file first, followed by image #1, then image #2, and so on. This is illustrated in Figure 4.1(a). Retrieving the HTML file and the inline images takes at least 2 RTTs each. As we will discuss in Section 4.2.1, it can take longer in practice, in part because of the overhead of TCP slow start.

### 4.1.2 Concurrent Retrieval

The transmission of the components of a Web page from a server to a client happens concurrently. For a page with several inline images, the client first retrieves the HTML file, identifies the inline images that the page contains, and then requests some or all of these images concurrently, without waiting for an intervening response from the server. This is illustrated in Figure 4.1(b). When the number of inline images is *nimages*, the number of concurrent connections could vary between 1 (which degenerates to sequential retrieval) and *nimages* (which corresponds to all the images being retrieved concurrently). Even with *nimages* concurrent connections, it is at least 2 RTTs before the client requests the inline images. Although concurrent connections may potentially reduce page download time by overlapping multiple transfers, there is the possibility of adverse interaction with TCP's congestion control and loss recovery algorithms. We will discuss this in Section 4.2.3 and Section 4.3.

## 4.2 Performance Analysis

We analyze HTTP performance based on a simple mathematical model as well as via live experiments over the Internet.

### 4.2.1 Mathematical Analysis

We present a simple mathematical analysis of the performance of HTTP with sequential retrieval and with concurrent retrieval. For the purposes of this analysis, we make the following simplifying (and optimistic) assumptions:

1. Both the transmission delay and the queuing delay for a packet are negligible compared to the propagation delay.

2. No packets are dropped in the network due to congestion.

3. The processing delay (due to the operating system, disk, display, etc.) at the server and client hosts is negligible compared to the propagation delay.

Assumption #1 allows us to isolate the effect of RTT from that of limited bandwidth. This is a reasonable assumption when the available bandwidth is large and the network load is low. For example, the transmission time for a 30 KB Web page download over a lightly-loaded wide-area T3 line (45 Mbps) is about 5 ms, which is much smaller than a typical wide-area round trip time (RTT) of the order of 100 ms. But clearly this assumption does not hold for slow links including dialup lines. We consider the effect of limited bandwidth in Section 4.2.3.

Assumption #2 is predicated on assumption #1. Since we assume that the transmission delay is negligible and that little queuing occurs, it is reasonable to assume that buffers never overflow.

Assumption #3 is reasonable given that even low-end PC-based servers can handle loads of a few thousand requests per second while keeping latency under 10 ms (as demonstrated by Web server benchmarks such as SPECWeb96 [99]). Only a small number of the busiest servers in the Web today experience a load that exceeds this level. And such busy Web servers often comprise a cluster of server nodes, so latency remains low even when the load exceeds the capacity of an individual node.

With these assumptions, the time to download a Web page is dominated by the number of network round trips incurred (with each round trip consuming time equal to one RTT). We assume that the Web page transfer involves the following steps: the client requests and downloads the HTML file, then parses the HTML file to determine the URLs of embedded objects such as inline images, and finally requests and downloads each individual image.

The download of a single file involves setting up a TCP connection, exchanging a request and a response, and finally tearing down the connection. Connection setup involves the SYN handshake, which contributes one RTT to the download time. A connection can be torn down asynchronously (i.e., without having either the sending or the receiving process wait for it to complete), so it does not add to the download time. This leaves the time taken by the request-response exchange.

We define the following variables for use in our analysis:

*nbytes* = size of a transfer

*mss* = the maximum segment size (MSS) of the TCP connection used in the transfer

*nsegs* = the number of segments in the transfer = ceiling of *nbytes*/*mss*

*rtt* = network round trip time

Since we assume that the transmission time and queuing delay are negligible, the time for the download is determined by the number of RTTs consumed as the TCP connection ramps up its window. Since we also assume that there are no packet losses due to congestion, the TCP connection remains in the slow start phase throughout. In this phase, the number of segments transferred in $x$ round trips is $2^x$-1. Therefore, the number of round trips needed to transfer *nsegs* segments is[2]:

$$\lceil \log(nsegs + 1) \rceil = \left\lceil \log\left(\left\lceil \frac{nbytes}{mss} \right\rceil + 1\right) \right\rceil$$

There is, in addition, the round trip time consumed for connection setup. Therefore, the time to complete a transfer of length *nbytes* is:

$$t(nbytes) = rtt \cdot \left(1 + \left\lceil \log\left(\left\lceil \frac{nbytes}{mss} \right\rceil + 1\right) \right\rceil\right)$$

We can now compute the download time, *T*, for a Web page comprising an HTML file of size *hbytes* and *nimages* inline images each of size *ibytes*.

With sequential retrieval, the HTML file and the inline images are transferred sequentially, so the download time is:

$$T_{seq} = t(hbytes) + n\mathrm{imag}es \cdot t(ibytes)$$

With concurrent retrieval, the HTML file is transferred first, then all the images are transferred concurrently. So the total time taken is:

$$T_{conc} = t(hbytes) + t(ibytes)$$

2. Note that the logarithm is taken to base 2.

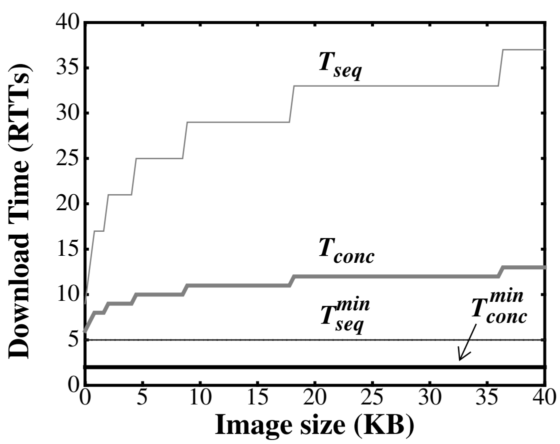
Fundamentally, there are only two constraints a Web page download: first, the HTML file has to be transferred before the client can determine which inline images to request[3], and second, the transfer of a file requires at least one RTT to send the request and receive the response. Thus the minimum possible completion time for the Web page download in question, with sequential and concurrent retrieval, respectively, is:

$$T_{seq}^{min} = rtt \cdot (1 + nimages)$$

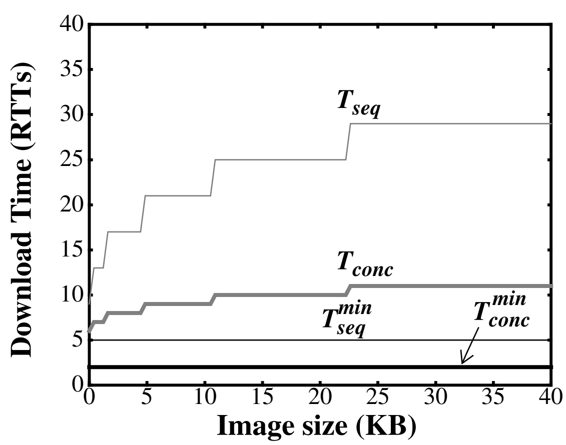
$$T_{conc}^{min} = rtt \cdot (1 + 1) = 2 \cdot rtt$$

Figure 4.2 illustrates the download time for the various cases graphically. The size of the HTML file is assumed to be 5 KB and the number of inline images to be 4. The download time is expressed as a multiple of the RTT and is computed using the formulae listed above. We make several important observations based on the figure:

1. Sequential retrieval of the inline images is far more time-consuming than concurrent retrieval.




(a)




(b)

**Figure 4.2 The download time for a Web page comprising a 5 KB HTML file and 4 inline images, as a function of the image size. The download time is expressed in multiples of the RTT. The MSS of the TCP connection is set either to (a) 576 bytes (a widely-used default setting for the MSS) or (b) 1500 bytes (the Ethernet MTU).**

3. For now, we ignore speculative techniques such as prefetching.

2. Both with sequential and with concurrent retrieval, much of the difference between the actual download time and the minimum possible is due to the TCP slow start procedure which is invoked each time a new connection starts.

3. The smaller the MSS is, the larger the download time is, owing to the larger number of RTTs consumed during slow start.

4. When transfers are short in length (for instance, when the inline images are small in size), the RTT for connection setup could constitute a significant fraction of the total transfer time.

There is a fundamental lower bound on the RTT due to the finite speed of light. Therefore, as technological advances bring forth faster computers and higher-bandwidth networks, it will become increasingly more important to minimize the number of round trips consumed.

### 4.2.2 Impact of Connection Length on Throughput

The TCP slow start algorithm also has an indirect impact on the throughput that a connection can achieve. This is because the throughput is bounded by the ratio of the TCP window size to the round trip time. Therefore, when the window size is small, as it tends to be during slow start, the throughput is also low.

Figure 4.3 shows the throughput for wide-area transfers of various length between a host in Berkeley, CA, USA and one in Germantown, MD, USA. The round trip time was approximately 71 ms. Figure 4.3(a) shows a standard plot while Figure 4.3(b) uses a log-scale for the connection length. We observe that the throughput rises, sharply at first and then more slowly, as the connection gets longer. However, the throughput of even a moderate-length transfer falls far short of that of a long transfer. For instance, a transfer of length 100 KB (which is longer than most Web transfers) achieves less than one-third the throughput of a 4 MB long transfer. This significant underutilization of bandwidth causes a corresponding increase in latency.

### 4.2.3 Impact of Limited Bandwidth

We assumed in Section 4.2.1 is that bandwidth is plentiful. In practice, this may not be the case. There are several instances where bandwidth may be limited; examples include slow speed access

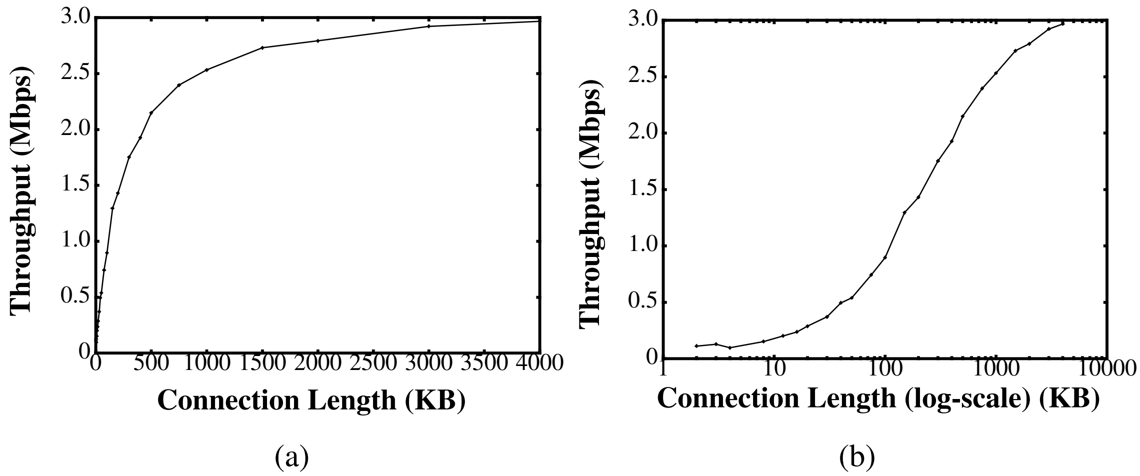



**Figure 4.3 The throughput between a host in Berkeley and one in Maryland as a function of the connection length. The connection length is plotted (a) on a linear scale, and (b) on a log scale.**

links such as dialup lines and heavily-loaded wide area links such as trans-continental links. The limited bandwidth could impact HTTP performance in several ways:

1. The transmission delay could become large. For instance, the transmission delay for a 1 KB packet over a 28.8 Kbps dialup line is about 280 ms.

2. The product of the bandwidth and the RTT represents the capacity of the data pipe. When the window size of a TCP connection, or the sum of the window sizes of a set of concurrent connections, exceeds the bandwidth-delay product, a queue forms at the bottleneck link. Such queuing has several adverse consequences:

   a. It increases the RTT. For example, a burst of four 1 KB segments over a 28.8 Kbps line could increase the RTT by more than a second. As a result, a TCP sender's retransmission timer might expire prematurely, causing performance degradation.

   b. The queue could grow so long as to cause a buffer overflow, forcing packets to be dropped. The delay incurred in recovering from these packet losses increases latency.

   c. While the inherent burstiness of slow start makes even a single connection susceptible to the adverse effects discussed above, the possibility is even greater with multiple concurrent connections. This is because each connection grows its window independently of the others, and does not back off until one or more of *its* packets are dropped.

Therefore, evaluating HTTP performance in general requires considering the effects of limited bandwidth in addition to the number of round trips consumed. However, it is difficult to model the effects of limited bandwidth analytically, especially due to the dominance of transient effects such as a packet loss burst during slow start. In light of this, we chose to study these effects via traffic trace analysis rather than pure mathematical analysis.

## 4.3 Analysis of Traces from the IBM Olympics Web Server

We analyzed traces from the official Web server for the 1996 Olympic Games. Because the server was based on HTTP/1.0, clients used a separate TCP connection to retrieve the individual components of each Web page. The components were retrieved either sequentially or concurrently depending on the client Web browser and its configuration. The Web server implemented the Reno variant of TCP.

The pre-processed traces contained a complete record of TCP events at the server, including packet transmission and reception, packet retransmission and timeouts. We analyzed the traces to answer to several questions pertaining to Web data transport performance:

1. How large is the effective window size of TCP connections used for Web transfers? How often is the effective window size constrained by the receiver-advertised window? A frequent occurrence of the latter would indicate underutilization of network bandwidth.

2. How effective are the TCP loss recovery mechanisms in the context of Web transfers? What are the potential benefits of more sophisticated algorithms such as selective acknowledgements?

3. How does the performance seen by a client correlate with the number of concurrent connections that it establishes to a server? A positive correlation would indicate that clients have an incentive to use a large number of concurrent connections. But as the answer to the next question indicates, such use of connections could have a detrimental impact on the network as a whole.

4. How does the establishment of multiple concurrent connections between a client and a server impact congestion control?

On account of the extremely large size of the traces, we base the analysis presented here only on a subset gathered during a busy 3-hour period.

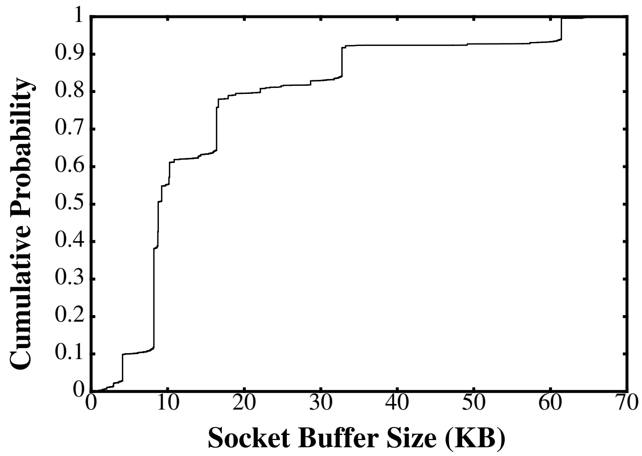



**Figure 4.4 Cumulative distribution function (CDF) of the receiver advertised window size.**

## 4.3.1 TCP Window Size Distribution

The TCP window size determines the amount of unacknowledged data that the sender can have outstanding in the network. It has a direct bearing on the throughput achieved by a connection, and consequently, the latency for completing a transfer of a certain size. In addition to the congestion control window, TCP also employs a flow control window, which indicates to the sender the amount of free buffer space that the receiver has. Since the size of this window is carried in acks transmitted by the receiver, it is often called the *receiver-advertised window*. The sender is constrained by both windows, so its *effective* window size is the smaller of its congestion window size and the receiver-advertised (flow control) window size.

In an actual implementation, the buffer space available to the receiver, and consequently the receiver-advertised window, is determined by the size of the buffer (called the *socket buffer*) allocated by the receiving application (such as a Web browser) at the time of connection establishment. Figure 4.4 shows the cumulative distribution function (CDF) of the receiver-advertised window size for the connections in the trace. There are marked upswings at commonly used socket buffer sizes such as 4 KB, 8 KB, 16 KB, etc. On the whole, the receiver-advertised window tends to be rather small. For instance, it is smaller than 8 KB in over 60% of the cases.

We also investigated how the receiver-advertised window compares with the congestion window size (as computed by the TCP emulation engine). Analysis of the traces reveals that in approximately 14% of all connections, the latter grew to be larger than the former, i.e., the receiver-advertised window was the limiting factor in performance. Of course, this also means that *in the vast majority of connections, the congestion window remains smaller than the receiver-advertised window, which in itself tends to be small* (as evident from Figure 4.4).

The reason that the receiver-advertised window tends to be small is that most operating systems tend to use a small receiver buffer by default, and applications such as Web browsers often do not explicitly request a larger buffer. On the other hand, the congestion window tends to be small because the individual connections tend to be short in length; the mean connection length is 5.5 KB and the median 3.1 KB [7]. As a result, slow start fails to build up a large window.

| Trace Statistic | Value | % |
|---|---|---|
| Total connections | 1650103 | |
| Total packets | 7821638 | |
| During slow-start | 6662050 | 85 |
| During congestion avoidance | 1159588 | 15 |
| Total retransmissions | 857142 | |
| Fast retransmissions | 375306 | 44 |
| Slow-start retransmissions | 59811 | 7 |
| Coarse timeouts retransmissions | 422025 | 49 |
| Avoidable with SACKs | 18713 | 4 |
| At least one duplicate ack received | 104287 | 25 |

**Table 4.1 Summary of Analysis Results (percentages are relative to the corresponding category)**

The tendency connections to be short in length also implies that slow start dominates the congestion avoidance phase. As indicated in Table 4.1, an overwhelming majority (85%) of the packets were transmitted by the server during the slow start phase of the corresponding connection. Only 15% percent were sent during the congestion avoidance phase. The dominance of slow start means that connections spend the majority of their time probing for bandwidth rather than effectively utilizing their share of the network bandwidth.

In addition to reducing throughput and increasing latency, a small window size worsens loss recovery, as we discuss next.

### 4.3.2 Effectiveness of TCP Loss Recovery

When a TCP sender detects that a packet loss, it attempts to recover from the loss by retransmitting the corresponding packet. A packet retransmission fall into one of three categories:

1. Fast retransmission
2. Timeout-driven retransmission
3. Slow start retransmission

Fast retransmission and timeout-driven retransmission are triggered by the receipt of a threshold number of duplicate acks and the expiration of the retransmission timer, respectively. A slow start retransmission could happen during slow start following a retransmission timeout. In some cases, it may be an unnecessary retransmission. The original transmission of the packet may have reached the receiver, but the sender does not know this because the cumulative ack still indicates a smaller sequence number because of the loss of an earlier packet.

Table 4.1 summarizes results from the analysis of the trace. Approximately 49% of packet retransmissions resulted from timeouts and a further 7% from slow start following timeouts. Only 44% of the retransmissions were triggered by the fast retransmission algorithm. *Thus timer-driven loss recovery, which entails substantial idle time waiting for the timer to expire and consequently increases latency, is the dominant mode of loss recovery.*

There are two main reasons why a TCP Reno sender could suffer a retransmission timeout:

1. *When multiple packets are lost within a window*, it is highly likely that the sender is forced to time out in order to recover from losses other than the first one [28]. This is especially so when the effective window size is small.
2. *When an insufficient number of duplicate acks are received*, either because the are not a sufficient number of outstanding packets or because most of the packets in a window are lost, the sender may be forced to time out even to recovery from the first loss. As before, a timeout is more likely to happen when the effective window size is small.

One proposed technique to improve the performance of TCP loss recovery is *selective acknowledgements* (*SACK*) [67]. There are two ways in which SACK could potentially help. First, it could prevent the unnecessary retransmission of packets that have reached the receiver but have not yet been acknowledged by a cumulative ack because of an earlier packet loss. Thus, slow start retransmissions (7% of all retransmissions) could potentially be avoided.

Second, SACK allows for an alternative interpretation of the 3-duplicate ack threshold that triggers fast retransmission in cases where a timeout would otherwise be necessary. Basically, the sender could use SACK information to determine if a later packet, that is at least 3 packets beyond the one that is suspected to be lost, has reached the receiver. If so, the sender could immediately invoke fast retransmission, while making the same assumption as standard TCP about the likelihood and extent of packet reordering. The important point to note is that the sender need not wait for 3 duplicate acks. *In spite of this potential benefit, our analysis of the traces shows that SACK would have avoided only about 4% of the retransmission timeouts that happened.* This is because the effective window size is often too small for SACK to provide any information about packets much beyond the one suspected to be lost. Therefore, the sender is constrained to suffer a timeout.

In summary, retransmission timeouts are the dominant form of loss recovery for TCP connections in the Web, due mainly to the small window size of individual connections. The overhead of timeouts increases latency and decreases throughput. SACK does little to rectify the situation.

### 4.3.3 Effectiveness of TCP Congestion Control

Having discussed in some detail the impact that short connection lengths, and hence small window sizes, have on performance, we turn to the impact that multiple concurrent connections between a Web server and a client have. The point here is that TCP congestion control operates at the granularity of individual connections. So each of the concurrent connections invokes congestion control independently of the others. We discuss how this impacts the aggregate throughput and the aggregate congestion response of the concurrent connections.

#### 4.3.3.1 Throughput Analysis

We analyzed the traces to determine if there is any significant correlation between the throughput seen by a client host and the number of concurrent connections ($n$) it has open to the server. (Each connection could, for instance, be used to transfer a logically-separate component of a Web page.)

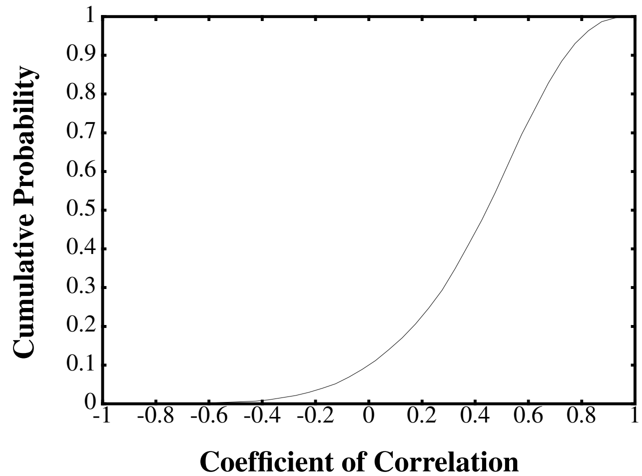



**Figure 4.5 CDF of the coefficient of correlation between throughput and the number of simultaneous connections. The CDF is obtained by aggregating across all hosts.**

A positive correlation would indicate that applications could appropriate an unfair share of the bottleneck link bandwidth by opening several concurrent connections to another host.

For each host, we divided the entire duration of its interactions with the server into periods during which the number of concurrent connections, $n$, is constant. A transition from one period to the next happens either when a connection terminates ($n$ decreases by 1) or when a new connection begins ($n$ increases by 1). For each such period, we computed the throughput as the ratio of the total number of useful bytes transferred during the period by all the connections to the duration of the period. We computed the throughput for a period only when a significant amount of data (at least 5 KB) was transferred during that period. Finally, we computed the correlation coefficient between the throughput and the number of concurrent connections, $n$. We performed this computation only for hosts which had at least 10 pairs of (*throughput, n*) samples.

By aggregating the correlation coefficient for different hosts, we obtained the cumulative distribution function (CDF), which is shown in Figure 4.5. It is clear from the figure that there is a positive correlation for about 90% of the hosts. The correlation coefficient is larger than 0.5 for about 45% of the hosts. The substantial positive correlation indicates that clients are better off performance-wise if they open a larger number of concurrent connections. But as we discuss next, there is a cost in terms of increased congestion which can worsen the performance of the network as a whole.

#### 4.3.3.2 Congestion Avoidance Analysis

There are several reasons why concurrent connections may improve performance for a client. These include:

1. The connection setup phase of concurrent connections could overlap in time, thereby cutting down latency by several RTTs.
2. Each of the concurrent connections has an initial window size of 1 segment. So *n* of them would have an aggregate initial window size of *n* segments. So during slow start, the aggregate congestion window size of the concurrent connections starts off at a higher point (*n* segments) on the exponential growth curve than the window for a single connection would (1 segment). This can potentially decrease the latency that the client experiences by *log(n)* RTTs.
3. During the congestion avoidance (linear) phase, each connection increases its window size by 1 segment per RTT. Therefore, *n* connections that are in the linear phase concurrently will increase the aggregate window by *n* segments per RTT. The faster window growth, again, has the potential of cutting down the latency experienced by the client.

Clearly, it is desirable to cut down or eliminate the connection establishment latency (the first point mentioned above). Opening concurrent connections is one way of doing so. The TCP accelerated open (TAO) technique employed by T/TCP (Section 2.3.4) is an alternative way. However, both a larger initial window size (the second point) and a faster window growth (the third point) are undesirable for two reasons.

First, these steps undermine TCP congestion avoidance. The aggregate initial window size is a function of the number of concurrent connections, *n*, opened by the client. The choice of *n* is made by the application without necessarily taking into account the network conditions, such as the bottleneck link bandwidth or the network load. While a large initial window size may be appropriate when the bandwidth-delay product is large (as in a geostationary satellite-based network), it could overwhelm a slower link, such as a dialup line, and cause heavy packet loss. Similarly, speeding up the pace of window growth may be appropriate for fast networks but not for slow ones.

Second, these steps contribute to unfairness. Consider two clients, $C_s$ and $C_c$, that are trying to download a Web page from a server. $C_s$ uses a single TCP connection at a time for the purpose while $C_{\mathbf{c}}$ opens multiple connections concurrently. Furthermore, assume that the two clients share the same bottleneck link. Ideally, the two clients should share capacity of the bottleneck link equally. But because of its more aggressive behavior, $C_{\mathbf{c}}$ would unfairly obtain better performance. Worse still, a client could decide to open multiple concurrent connections with the explicit purpose of exploiting this unfairness. For instance, the client could have the server split up a logical Web page component[4], such as an inline image, into multiple pieces and open a separate connection to retrieve each piece, just to improve its own performance.

#### 4.3.3.3 Congestion Control Analysis

In addition to the three points listed in the previous section, a client that opens multiple concurrent connections also benefits because the aggregate congestion control response of the connections is more aggressive than that of a single connection. When a TCP sender detects a packet loss (an implicit signal of congestion), it slows down, either by halving its congestion window size (with data-driven loss recovery) or by resetting it to one segment (with timer-driven loss recovery). These actions are taken by individual connections based on losses suffered by their own packets. So in a set of concurrent connections, only those that experience a packet loss slow down. This is in spite of the loss on even one connection being indicative of congestion along the (shared) path of all of the connections.

To evaluate the aggregate congestion control response of concurrent connections, we analyzed the traces to determine how packet losses tend to be distributed across the set of concurrent connections between the server and each individual client. Consider a period of time when a client host has $n$ concurrent connections open to the server. Suppose that one of the $n$ connections experiences a packet loss. This event marks the beginning of a *loss epoch*. We recorded the amount of outstanding data on each connection at such points in time. The time when this outstanding data on all of the connections has been acknowledged marks the end of the loss epoch. For each loss epoch, we recorded the number of concurrent connections, the outstanding window size of each connection and the distribution of packet loss events across the connections. We aggregated this data over all client hosts.

4. The *byte-range* option in HTTP [30] allows a client to request a specific range of bytes from a file.

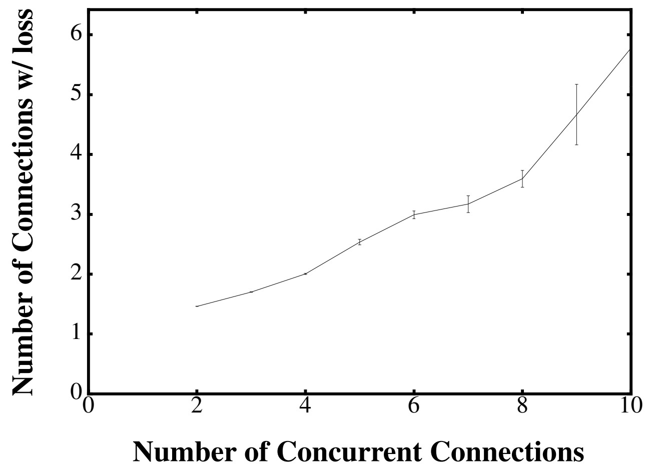



**Figure 4.6 Average number of connections that experience a loss within the same window when using concurrent connections.**

Of the set (of size *n*) of connections that are active during a loss epoch, only a subset (of size $m \leq n$) actually experiences a loss during that loss epoch. Only the connections in this subset cut down their individual congestion windows during the loss epoch. Assuming that all the TCP connections have the same congestion window size at the start of the epoch and that each TCP connection that experiences a loss halves its window, the effective multiplicative backoff in a host's total transmission window is *(1 - m/2n)*. Figure 4.6 shows that on average half the total number of concurrent connections experience a loss during a loss epoch, i.e., *m* is approximately *n/2*. So the effective multiplicative backoff factor is *3/4*, which represents a much more aggressive behavior than halving the window.

We complemented this back-of-the-envelope calculation with an analysis to obtain bounds on the effective multiplicative backoff factor. The analysis discussed previously told us the congestion window sizes of each connection during a loss epoch and which connection(s) experienced packet loss during that epoch. To obtain bounds on the effective multiplicative backoff factor, we made one of two extreme assumptions about how the set of connections that experience a loss during a loss epoch respond. To obtain an upper bound, we assumed that each such connection does a fast retransmission and halves its window. To obtain a lower bound, we assumed that each undergoes a retransmission timeout and resets its window to one segment. For each of the two cases, the ratio of the combined congestion window after before the loss event to that before gave us a bound on

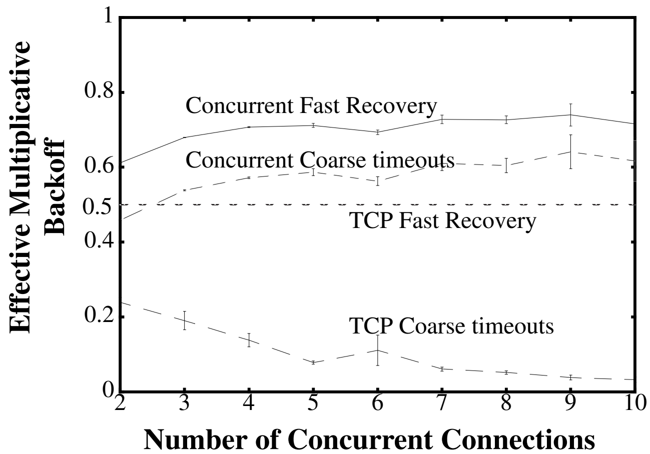



**Figure 4.7 Effect of concurrent connections on the extent of backoff in response to congestion-induced packet loss.**

the effective multiplicative decrease factor. These bounds are plotted as a function of the number of concurrent connections in Figure 4.7 (the curves labelled “Concurrent Fast Recovery” and “Concurrent Coarse Timeouts”). As a baseline comparison, we also compute the multiplicative backoff in congestion window that would result if the connections were treated as a single unit and (a) a fast retransmission occurred, or (b) a coarse timeout occurred. The corresponding curves are labelled as “TCP Fast Recovery” and “TCP Coarse Timeouts”, respectively, in the figure. With concurrent connections, the effective multiplicative backoff factor is in the range 0.6 to 0.75, significantly more aggressive than a single TCP connection which would backs off twofold or more (i.e., a factor of 0.5 or less).

Therefore, not only do multiple concurrent connections cause aggressive growth of the congestion window compared to a single TCP connection, they also cause reduced responsiveness to indicators of congestion such as packet loss[5]. The former makes it more likely that the network gets into a congested state. The latter confounds recovery from congestion. Together, these two effects could degrade overall network performance. It is difficult to quantify the degradation using the Web server traces, so we demonstrate this via simulation experiments in Chapter 6.

5. The situation is similar with other indicators of congestion such as ECN [31] that also operate at the granularity of individual connections.

## 4.4 Summary

In this chapter, we have analyzed the performance impact of composing HTTP/1.0 and TCP, using both mathematical analysis and traffic trace analysis. HTTP/1.0 maps each component of a Web page onto a separate TCP connection. Such a mapping is convenient because TCP in itself does not provide any means for demarcating multiple messages within a single connection. But it causes several performance problems, both with sequential retrieval of Web page components and with concurrent retrieval.

Sequential retrieval simplifies both the server and the client processes because they contend with only one active connection at a time. Moreover, the single connection in progress reacts to congestion by backing off twofold or more. On the downside, though, delay is incurred in establishing each connection, issuing the HTTP request and waiting for the response for each component, and slow start. The short length of each connection causes the congestion window to be small. So bandwidth utilization is poor and loss recovery often depends on retransmission timeouts, which further increase latency.

Concurrent retrieval reduces the delays due to connection establishment the HTTP request-response exchange because these happen concurrently over multiple connections. A fundamental problem though is that the concurrent connections operate independently of each other. In particular, the connections do not share indications of congestion along their shared path. Consequently, the set of concurrent connections as an aggregate is more aggressive than a single connection in terms of congestion avoidance and less responsive in terms of congestion control. By opening multiple concurrent connections, an application can obtain a disproportionate share of network bandwidth, and also exacerbate network congestion, to the detriment of other connections in the network. Furthermore, since each connection is still short in length, retransmission timeouts would remain the predominant form of loss recovery.

To tackle these problems, we proposed a new connection abstraction for HTTP, which persists across multiple HTTP transactions. Our proposal, called *persistent-connection HTTP* (*P-HTTP*), decreases Web download latency without resorting to aggressive use of network resources. We discuss P-HTTP in the next chapter.

# Chapter 5

# Persistent-Connection HTTP

In this chapter, we present our initial solution to the performance problems that arise from the composition of HTTP and TCP. We call this solution *persistent-connection HTTP* (*P-HTTP*)[1]. P-HTTP introduces a new connection abstraction for HTTP, that of a *persistent connection* which is reused for multiple HTTP transactions. P-HTTP avoids the overhead introduced by multiple short-length connections in HTTP/1.0, and thereby reduces latency. It further reduces latency by *pipelining* multiple transactions over the persistent connection. The ideas of persistent connections and pipelining in P-HTTP, developed by us in 1994 [85], have been adopted by the new HTTP/1.1 protocol [30]. We focus on P-HTTP in this chapter, and touch upon only briefly HTTP/1.1.

We begin by enumerating the specific goals of P-HTTP in Section 5.1. We then discuss the design of P-HTTP, specifically persistent connections and pipelining, in Section 5.2 and Section 5.3. We describe our implementation in Section 5.4 and present performance results in Section 5.5. In Section 5.6, we discuss extensions of P-HTTP. In Section 5.7, we point out limitations of P-HTTP, which motivated us to develop the *TCP session* and *TCP fast start* techniques presented in Chapter 6 through Chapter 8. Finally, we present a summary in Section 5.8.

1. The name "P-HTTP" was picked much after this work was done by Jeff Mogul and me. The credit for the choice of this name goes to Jeff [74].

## 5.1 Goals of P-HTTP

In addressing the performance problems arising from the composition of HTTP and TCP, the specific goals are to:

1. Reduce latency arising from:

   a. Connection establishment. The latency is dominated by the RTT for the SYN handshake, but it also includes the overhead of connection setup processing at the end hosts.

   b. HTTP request-response exchanges.

   c. Slow start.

2. Reduce the dependence on timeouts for loss recovery.

3. Make congestion avoidance and control as effective as with a single TCP connection.

We discuss the two key components of P-HTTP that help in meeting these goals, persistent connections and pipelining, in turn.

## 5.2 Persistent Connections

Since many of the performance problems of HTTP/1.0 arise from the use of a large number of short-length TCP connections, we eliminate the underlying cause by replacing the numerous short connections with a single *persistent* TCP connection. Multiple HTTP transactions *share* this connection. The transactions correspond to the retrieval of the individual components of a Web page and perhaps also multiple Web pages.

The logical transfers (or logical data streams) corresponding to the individual components of a Web page are mapped onto this connection. Furthermore, the connection is made *persistent*, i.e., long-lived, so that it can be shared across multiple client-server interactions that are spaced apart in time.

We need a way to map the logical transfers (or logical data streams) corresponding to the individual components of a Web page onto the persistent connection. A straightforward mapping would be to arrange the logical transfers in sequential order. So a typical Web page download would involve a request-response exchange for the HTML file followed, in sequence, by similar

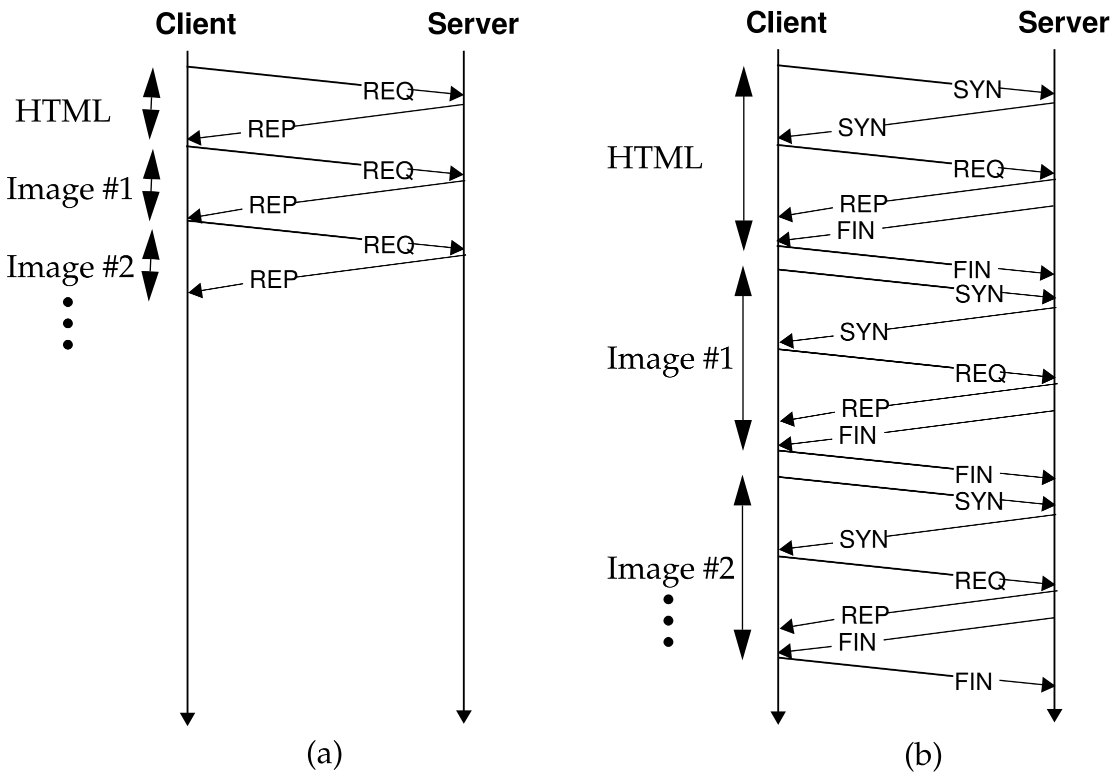



**Figure 5.1 Timeline of a Web page download with (a) a persistent connection, and (b) a separate connection for each component. In both cases the request-response exchange for the individual components happen sequentially, one after the other. In case (a) we assume that the persistent connection has already been established by a previous interaction between the client and the server.**

exchanges for each inline image. Figure 5.1(a) shows the timeline, assuming that the persistent connection has already been set up by a previous interaction between the same client-server pair. For comparison, we also show the HTTP/1.0 timeline with sequential retrieval in Figure 5.1(b) (reproduced from Figure 4.1(a)).

One clear benefit of a persistent connection is that it amortizes the overhead of TCP connection establishment over multiple logical data transfers, and even multiple Web page downloads. As illustrated in Figure 5.1, the download of a Web page composed of an HTML file and *nimages* inline images would benefit from a reduction of $(1 + nimages) \cdot rtt$ in latency under P-HTTP.

A second benefit of a persistent connection is that the overhead of slow start and other aspects of the TCP congestion avoidance/control algorithm is likewise amortized over multiple logical data transfers. From the viewpoint of the TCP connection, the several distinct logical data streams appear as a single data stream. The logical structure of the sub-stream within the TCP connection has no impact on the TCP algorithms. Thus in Figure 5.1(a), the slow start process initiated during the transfer of the HTML file continues through the subsequent transfer of the inline images. (Of course, if packet loss is detected, the congestion window size would be cut down and the linear growth phase possibly initiated.)

Unfortunately, the benefit of a persistent connection with respect to slow start overhead may not extend to multiple Web page downloads spaced apart in time even if they share the same connection. This occurs because TCP resets the congestion window to its initial value when a connection is idle for a certain period (Section 2.1.5). This phenomenon has been termed the *slow start restart problem* in [44]. The threshold used for the idle period is the RTT or the RTO, both of which tend to be smaller than the user "think time" that spaces successive Web downloads apart. Therefore, when the user requests a new Web page after the think time, slow start is initiated afresh. This increases latency.

Our solution is a new technique that we call *TCP fast start*. To first order, TCP fast start avoids repeated slow start by reusing the congestion window size cached in the recent past. We defer a detailed discussion to Chapter 8.

## 5.2.1 Multiplexing Logical Data Streams

Recall from Section 2.1 that a TCP connection only provides an ordered byte-stream data delivery service. The protocol does not provide any means for demarcating messages or logical data streams within a connection. Therefore, to multiplex several logical data streams onto a persistent connection, the sender (typically the server in a Web interaction) would, at the least, need a mechanism for demarcating messages at the application level. The server could then send out the messages (logical data streams) sequentially one after the other.

A simple mechanism to demarcate messages that are multiplexed onto a single TCP connection is to add the length of a message as a prefix to the body of the message itself. The *content-length* field in the HTTP/1.0 protocol header could be used for this purpose. One issue to consider, though, is that the length of a message is not always known at the beginning of a transmission, for instance when the message is generated as the output of a server-side script. In such cases, it is difficult to include the content-length field. We consider several alternative solutions:

- *Boundary delimiter*: The server could insert a boundary delimiter (perhaps as simple as a single character) if it can examine the entire data stream and "escape" any instance of the delimiter that appears in the data (as is done in the Telnet protocol [90]). This requires both the server and client to examine each byte of data, which is clearly inefficient.
- *Block-by-block data transmission*: The server could read the output of the script and send it to the client in arbitrary-length blocks, each preceded by a length indicator. A zero-length block would mark the end of a message. This would not require byte-by-byte processing, but it would involve a lot of extra data copying on the server, and would also require a protocol change.
- *Store-and-forward*: This is an extreme case of block-by-block data transmission where the block is the entire output of the script. The server could measure the length of the block and include it in the content-length field. This obviates the need for a protocol change (because the standard content-length field could be used), but requires extra copying and is not practical for arbitrarily large objects.
- *Separate control connection*: The server could use a separate control connection (as in the FTP protocol [91]) to notify the client of the length of a message that it transmits on the data connection. However, the extra connection adds overhead.

Because of the overhead imposed on the server (and possibly the client) by each of these approaches, we chose a simple hybrid approach. In this approach, the server keeps the TCP connection open and uses the content-length field in cases where it can determine the length of the message prior to transmission. In other cases, it marks the end of the message by closing the connection, just as in HTTP/1.0. In the common case (serving out static files), this avoids the cost of extra TCP connections; in the less common case (invoking scripts to generate content dynamically), it may involve the overhead of extra connections but does not add data-touching operations on either the server or the client, and requires no protocol changes. Static files were dominant compared to dynamically-generated data around the time we designed P-HTTP (ca. 1994), and continue to be so. (For instance, [41] reports that fewer than 1% of the bytes and 2% of the files in a large Web client trace were of MIME type CGI (Common Gateway Interface) [18] which represents the majority of dynamically generated data.) Note that a static file refers to one whose size is fixed. It does not necessarily mean static content. In fact, it includes such dynamic content as animated GIF [37,38], dynamic HTML (DHTML) [24], and Java applets [57].

Even dynamically-generated pages contain significant amounts of static data in the form of banners, buttons, text, etc. Therefore, keeping a persistent connection open may be advantageous even for dynamically-generated pages. So the client opens separate non-persistent connections for dynamically-generated data and reserves the persistent connection for static data. The client uses a simple heuristic to distinguish static data from dynamic data: if the HTTP request method is GET and the URL does not contain a "?", the data is assumed to be static and the persistent connection is used. In all other cases, a new non-persistent connection is established. Note that this heuristic is only a performance enhancement and does not impact the correctness of the protocol.

Dynamically-generated may data become more important in the future. Web-based search engines and database front-ends are examples of applications where this is happening. Rather than redesign P-HTTP to avoid the attendant performance penalty, we sidestep the problem by developing an entirely new technique, *TCP session*, which we discuss in Chapter 6.

### 5.2.2 Negotiating the Use of Persistent Connections

Persistent connections require both servers and clients with capabilities over and beyond those required by HTTP/1.0. Servers must multiplex multiple messages onto a persistent connection while the client must parse the messaging boundaries using the content-length field, and extract the individual messages. Successful deployment requires interoperability between servers/clients that have this capability and those that do not. One way to realize this interoperability is to introduce a mechanism for servers and clients to negotiate the use of persistent connections.

To provide such a negotiation mechanism, we defined a new *hold-connection-open* HTTP *pragma directive* [10] that specifies the use of persistent connections. To negotiate the use of persistent connections, the client includes this directive in its HTTP request. If the server understands the directive, it holds the connection open and echoes the pragma in its response. At this point, both the client and the server have successfully negotiated the use of a persistent connection. If the server does not understand the *hold-connection-open* pragma directive, it simply ignores the directive. So negotiation fails and the client and server revert to the standard HTTP protocol. Similarly, if the server receives a request that does not include the directive, it assumes that the client does not support persistent connections and falls back to the standard protocol.

Subsequent to our work, the HTTP/1.1 protocol has defined a similar negotiation mechanism as part of the protocol. However, HTTP/1.1 is different from P-HTTP in one respect, and that is it designates the use of a persistent connection as the *default* behavior of clients and servers.

### 5.2.3 Terminating Persistent Connections

A persistent TCP connection consumes resources both at the server and the client. These resources include the TCP protocol control block and the socket buffer. Because of resource constraints, the server and/or client may wish to limit the number of persistent connections it holds open at any one time. When the number of persistent connections is at the limit and a new connection is needed, (at least) one of the existing connections must be terminated. To solve this problem, we use a simple least-recently-used (LRU) policy to pick the victim. It is possible to use more sophisticated policies, such as picking victims in increasing order of the round trip time, but we did not investigate this issue for this thesis.

The hosts at either end of a persistent connection must handle the case that their peer closes the persistent connection without notice. The closure of the connection could result in several failures from the viewpoint of a typical client-server interaction:

1. A client's attempt to send a new request may fail.
2. The client may succeed in sending its request but the server may not succeed in sending its reply.

A failure of the first kind, while undesirable, can be dealt with by having the client establish a new (persistent) connection and send its request again. However, a failure of the second kind can have more serious consequences because the server would have processed the client's request but the client would not know about it. Although many HTTP requests do not have side-effects and most are idempotent, some are not. (An example of one that is not idempotent is a request to cast a vote in a Web-based on-line poll.) In the latter case, there is no graceful way of recovering from a failure.

Therefore, our policy is to prevent the client or the server from closing a connection in the midst of a request-response exchange. This is easy to do because each can determine, using only local knowledge, whether there are any requests or responses outstanding. This avoids the second failure mode, so only the first remains. There are two ways in which the first failure mode can occur:

- The server has already informed the client host that it has closed the connection. In this case, the client program may also be informed of this before it tries to reuse the connection for a new request to the same server. At worst, it will find out when its attempt to send out the request fails immediately because the network stack is aware that the connection has been closed.
- The server's message indicating connection closure (TCP FIN packet) and the client's request cross each other on the network. In this case, the server rejects the request when it receives it. The client program discovers the failure when its attempt to read the reply fails.

In either case, the client program recovers from the failure by establishing a new connection and sending the request afresh.

## 5.3 Pipelining

Even with a persistent TCP connection, a simple implementation of the HTTP protocol would still require at least one network round trip to retrieve each inline image, as illustrated in Figure 5.1(a). The client interacts with the server in a stop-and-wait fashion, sending a request for an inline image only after having received the data for the previous one.

However, this lockstep sequence of requests can be avoided since one component of a Web page (such as an inline image) in no way depends on previous components. The client could, therefore, *pipeline* multiple requests over the same connection. We consider two ways in which the client requests could do this: GETALL and GETLIST. We note at the outset that while pipelining could be applied in the context of Web page components of arbitrary type, our description here assumes inline images, the most common type of component embedded in Web pages.

### 5.3.1 The GETALL method

When a client does a GET on a URL corresponding to an HTML document, the server just sends back the contents of the corresponding file. The client then sends separate requests for each inline

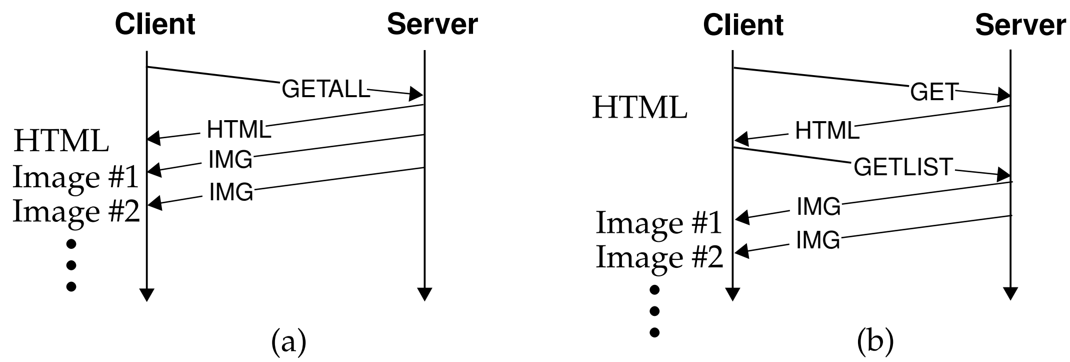



**Figure 5.2 Timeline of a Web page download over a persistent connection with pipelining enabled. (a) shows the use of GETALL, and (b) shows the use of GETLIST.**

image. Typically, however, most or all of the inline images reside on the same server as the HTML document, and will ultimately come from the same server.

We extended HTTP with a new *GETALL* method, which specifies that the server should return an HTML document and all of its inline images that reside on that server. On receiving this request, the server parses the HTML file to find the URLs of the images, then sends back the file and the images in a single response. The client uses the content-length field in the individual HTTP headers to split the response into its components (as described in Section 5.2.1).

Because GETALL relies on the server to parse HTML files, it imposes a new burden on the server. However, this additional overhead is offset by the server being relieved from parsing many additional HTTP requests, which the standard GET method would entail. If the cost of parsing HTML files is deemed too high, the server could keep a cache of the URLs associated with a subset of heavily-accessed HTML files, or even all of them.

The GETALL method can be implemented using the standard GET method, with a special pragma directive included in the request header to indicate that the client wants to perform a GETALL. This allows the client to inter-operate with a server that does not support GETALL. If the server supports GETALL, it includes the GETALL pragma directive in its reply header. The client waits for all the HTML and all the inline images. If the server does not support GETALL, it simply ignores the pragma directive. The non-inclusion of this directive in the server's response indicates to the client that it should revert to the standard GET method.

Web clients typically maintain a local cache of Web page components, such as inline images, to avoid unnecessary network interactions. However, a server has no way of knowing which of the inline images in a document are in the client's cache. So the GETALL method causes the server to return all the inline images, thereby defeating client-side caching in general. GETALL would still be useful in situations where the client knows that it has no relevant images cached (for example, if its cache contains no images from the server in question, or local caching is turned off).

### 5.3.2 The GETLIST method

To get around the limitation of the GETALL method, we defined a new *GETLIST* method, which allows a client to request a set of documents or images from a server. A client can use the GET method to retrieve an HTML file, and then use the GETLIST method to retrieve, in one exchange, all the images not in its cache.

Logically, a GETLIST is the same as a series of GETs sent back-to-back. In fact, we chose to implement it this way, since it requires no protocol change and it performs about the same as an explicit GETLIST would.

The GETLIST method incurs an additional network round trip (to retrieve the HTML file) compared to GETALL. However, more significantly, it avoids the drawback that GETALL with regard to client-side caching. So GETLIST is our method of choice for pipelining HTTP requests.

## 5.4 Implementation

We briefly describe two different implementations of P-HTTP that we have done. The first, involving a modified client and server, was our original one and is described in [85]. The second, our proxy-based implementation, is more recent.

### 5.4.1 Client-Server Implementation

The two components of P-HTTP, persistent connections and pipelining, require support at both the client and the server. Specifically, the use of persistent connections involves the following actions on the part of the client and the server:

- The client indicates its desire to use a persistent connection by including a *hold-connection* pragma in its initial request. It uses the content-length field to indicate the length of the request it is sending.

- If the server is capable and willing to use a persistent connection, it includes the *hold-connection* pragma in its response. It uses the content-length field in the request header to determine when it has received the entire request. Whenever possible, the server marks the end of its response using the content-length field. It closes the connection to a client when the content-length is not known beforehand. It also closes connections to conserve resources — either when the number of open connections exceeds a threshold or when a connection has been idle for longer than a threshold. Both these thresholds can be configured as appropriate.

- The client checks to see if the server's response includes the hold-connection pragma. If it does and if there is a content-length field in the reply, the client reads in the corresponding amount of data from the connection. It also uses the same (persistent) connection to send requests to the server in the future. If the server's response does not include either the hold-connection pragma or the content-length field, the client reads in data until the connection is closed by the server. Also, it establishes a new connection for future requests.

- If the client encounters the closure of the persistent connection by the server, either when trying to send a new request or while waiting for a response, it establishes a new connection and retries the request. If the server encounters a connection closure by the client, it does nothing. Since HTTP interactions are always initiated by the client, the onus of re-establishing the connection, if necessary, is placed on the client.

In addition, the client and the server do the following to support pipelining:

- Once the server has confirmed that it supports persistent connections (for instance, as part of the exchange to obtain the HTML file), the client is free to pipeline requests. It sends requests back-to-back using the content-length field in each request header to indicate its length.

- The server processes the requests from the client sequentially and sends back its response to each in the same sequence. It does not interleave the responses.

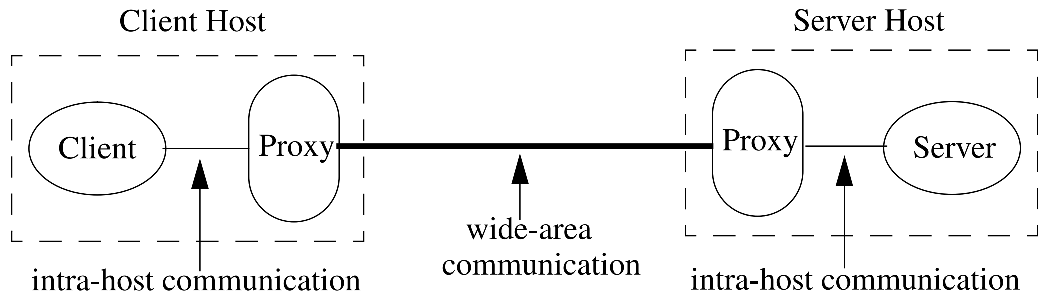



**Figure 5.3 The architecture of a proxy-based implementation of P-HTTP. P-HTTP operates only between the two proxies, leaving the client and server programs untouched.**

- It is essential that the server use the content-length field to indicate the end of a response. If it closes the connection instead, the client will not receive responses to later requests it may have pipelined. The client uses the same heuristic as mentioned in Section 5.2.1 to avoid forcing the server to close the connection because of dynamically-generated data. It only pipelines requests that use the GET method and do not include a '?' in the URL. A '?' indicates the likely use of a server-side script which may force the server to close the connection prematurely. Despite the restrictions, the client is able to pipeline the majority of requests, which are for static[2] files.

We implemented the client and server actions listed above by modifying the *Mosaic* version 2.4 browser and the *httpd* version 1.3 server, both from NCSA [77]. These represented the leading-edge in publicly-available client and server software at the time we did the implementation (circa mid-1994). We ported the modified software to two platforms: DEC MIPS running Ultrix and DEC Alpha running OSF. The modified software has been available via anonymous FTP since 1995 [86].

## 5.4.2 Proxy-based Implementation

Although our initial implementation of P-HTTP influenced later developments, including HTTP/1.1, it has the drawback of being tied to specific client and server software. As NCSA Mosaic and httpd have been replaced by newer software with added features, the reach of our implementation

2. As noted in Section 5.2.1, "static" files encompass "dynamic" content such as animated GIFs and Java applets.

has been restricted. Its suitability for performance testing has diminished because newer client and server software (such as Netscape Navigator [78], Internet Explorer [72] and Apache httpd [1]) include optimizations that are orthogonal to P-HTTP.

To get around these problems, we have implemented a simple, proxy-based version of P-HTTP. The key observation is that P-HTTP is primarily a technique to optimize network communication between a client and a server. If the client and server applications communicate with each other directly, implementing P-HTTP would involve modifying the applications. The key to avoiding this is to have the client and server communicate *indirectly* via proxies (Figure 5.3). With the proxies co-located with the client and the server, the network communication between the two proxies is the primary determinant of performance. That between the client and its proxy, and likewise the server and its proxy, constitutes intra-host or intra-LAN communication, and as such is far less expensive. Therefore, we can achieve much of the benefit of P-HTTP by having the proxies use a persistent connection and pipeline requests and responses on this connection.

The server software we used, Apache httpd version 1.3b5, supports the HTTP *keep-alive* mechanism which is similar to the *hold-connection* mechanism in our original implementation. Further, the server is capable of using the content-length field to separate out multiple requests and responses on the same connection. Therefore, we did not require a proxy at the server end.

The client software we used, Netscape Navigator version 3.01, allows us to configure the maximum number of concurrent connections that are opened to the same server. When a (local) proxy is used, this controls the number of concurrent connections established between the browser and the proxy. By varying the number of connections and by having the client-side proxy choose whether or not to use the keep-alive mechanism, we emulate there different protocol configurations:

1. *HTTP/1.0 with sequential connections* (Figure 5.4(a)): Only one connection is established at a time between the browser and the proxy, and the proxy does not use the keep-alive mechanism when communicating with the server. The net result is that a new connection is established between the proxy and the server for each HTTP request-response interaction.

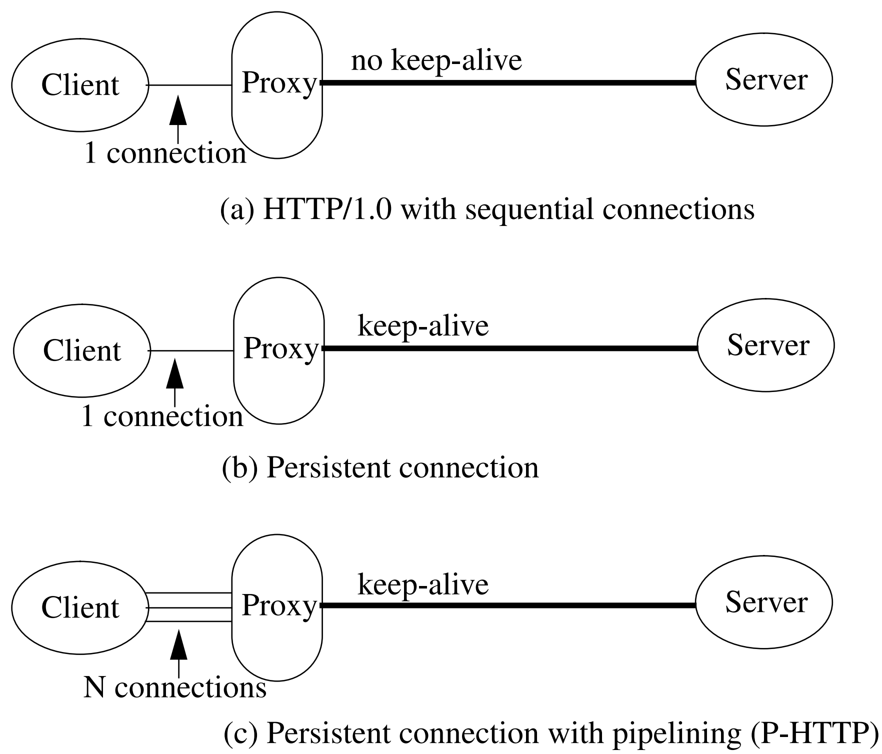



**Figure 5.4 Using a proxy-based implementation to emulate various protocol combinations.**

2. *Persistent connection* (Figure 5.4(b)): Only one connection is established at a time between the browser and the proxy, but the proxy does use the keep-alive mechanism when communicating with the server. Consequently, a persistent connection is established and used for communication between the proxy and the server. However, the single connection between the browser and the proxy prevents the pipelining of requests and responses.

3. *Persistent connections with pipelining* (P-HTTP) (Figure 5.4(c)): Up to N (>1) connections are established at a time between the browser and the proxy, and the proxy uses the keep-alive mechanism when communicating with the server. As a consequence, a persistent connection is established and used for communication between the proxy and the server. Moreover, the proxy pipelines up to N requests on the connection.

While the proxy-based implementation of P-HTTP is certainly more generally applicable than our original implementation, we decided against using it for our experiments. Our reasons for this decision are discussed in the next section.

## 5.5 Experimental Results

For our experiments, we wrote a simple Web client program that emulated HTTP/1.0, persistent connections and pipelining. This program sends out requests and reads in the responses, but does not display the retrieved material. We chose to use this rather than our proxy-based implementation because:

1. We discovered via experiments that the Netscape Navigator version 3.01 browser does not allow launching more than 6 concurrent connections. This limits the maximum number of requests that can be pipelined to 6, which is clearly undesirable.
2. A human user is needed to operate the browser, making it difficult to automate the experiment.
3. Using the simple client program allows us to isolate the network performance from other orthogonal issues such as rendering speed.

We conduct experiments using two different network configurations.

1. *Terrestrial wide-area network* (Figure 5.5(a)): The client host is located in Berkeley, CA, USA and the server host in Germantown, MD, USA. A 13-hop terrestrial path connects the server to the client and a 14-hop terrestrial path provides connectivity in the reverse direction. The client and server are connected to 10 Mbps Ethernet segments. The round trip time between the client and server is approximately 70 ms.
2. *Satellite-based wide-area network* (Figure 5.5(a)): The client and server locations are the same as above, but the client host also has an interface connected to the DirecPC satellite network [27]. Client-to-server communication happens via the same 13-hop terrestrial path as before, but communication in the reverse direction happens via a 2-hop path that includes a geostationary satellite link. The round trip time between the client and server via these asymmetric paths is approximately 335 ms.

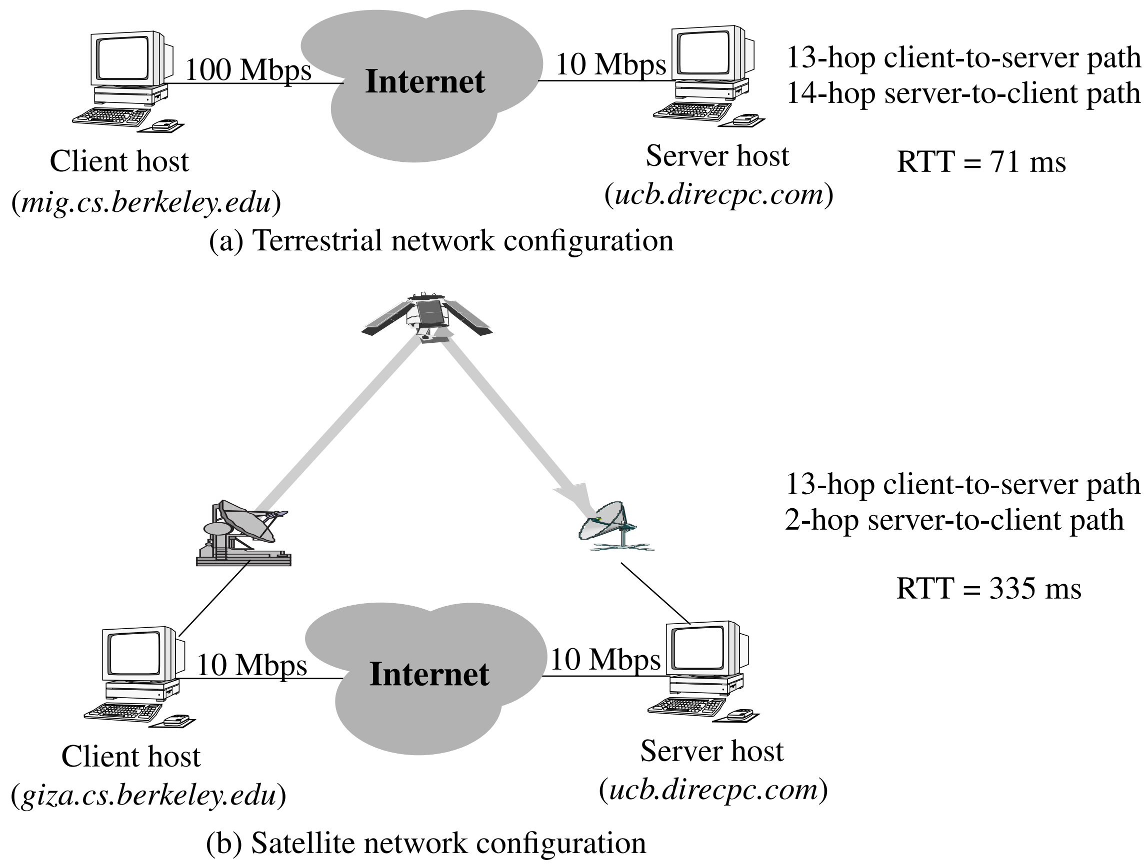



**Figure 5.5 The two network configurations used for our experiments. Note that both the browser and the proxy run on the client host.**

We create several test Web pages on the server. Each test page contains 1-10 inline images of identical sizes. We use GIF images of 3 sizes: 250 bytes (representative of small buttons or icons), 5 KB (approximately the average size of a GIF image as reported in [41]), and 40 KB (representative of larger images such as photographs). Thus we have a total of 10x3 = 30 test pages. We download each test page from the server and record the time taken, repeating this experiment 20 times. We turn off local caching at the browser to force the HTML file and the inline images to be downloaded from the server each time. We report the average download time, together with 95% confidence intervals, for each combination of image size and image count.

### 5.5.1 Terrestrial Wide-Area Network

Figure 5.6 shows the performance results for the experiments conducted across the terrestrial wide-area network. The general observation is that the use of persistent connections improves performance substantially. Pipelining results in an additional improvement.

In relative terms, the improvement in performance is greatest when the image size is the smallest. For instance, with 10 inline images, persistent connections and pipelining together decrease the average download time by approximately a factor of 6.13 for 250-byte images, 5.67 for 5 KB images and 4.73 for 40KB images. The reason is that the smaller the image size, the greater the cost of connection establishment and slow start as a fraction of the total download time. Since the use of persistent connections and pipelining reduces these costs, there is greater improvement when these costs are large in relative terms. We observe a similar trend when comparing the benefit of persistent connections and pipelining to that of persistent connections alone.

In absolute terms, however, the performance improvement is greater when the inline images are larger in size and number. This is evident from the different slopes of the curves in Figure 5.6, both within the individual graphs and across the three graphs. The reason for the different slopes is that the cost downloading an individual image is different in each case. With HTTP/1.0, it includes the cost of connection setup, the round trip consumed to request the image and start receiving the reply, and the entire cost of slow start for transmitting the reply. With persistent connections, the connection setup and slow start costs are amortized over several individual images, but the round trip for request-response exchange for each image remains. With pipelining added on, the round trip for the request-response exchange for the individual images overlap, further decreasing the cost per image.

### 5.5.2 Satellite-based Wide-Area Network

Figure 5.7 shows the performance results for the experiments conducted over the DirecPC satellite network. The general performance trends are the same as for the case of the terrestrial network. However, the performance gains in absolute terms is much larger than in the case of the terrestrial network owing to the much larger round trip time in the former.

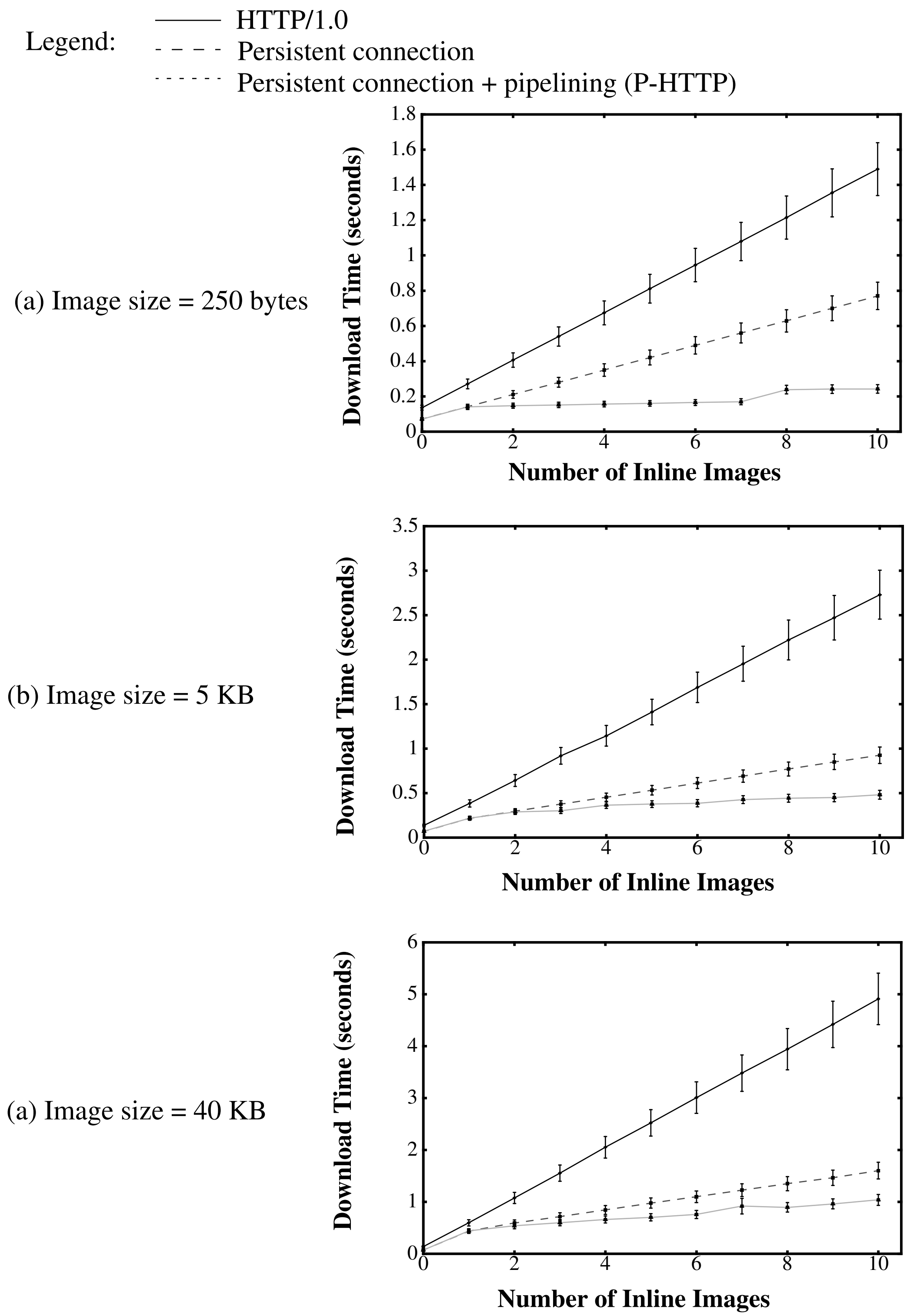



**Figure 5.6 Results of experiments conducted across a terrestrial WAN. The error bars correspond to 95% confidence intervals for the mean.**

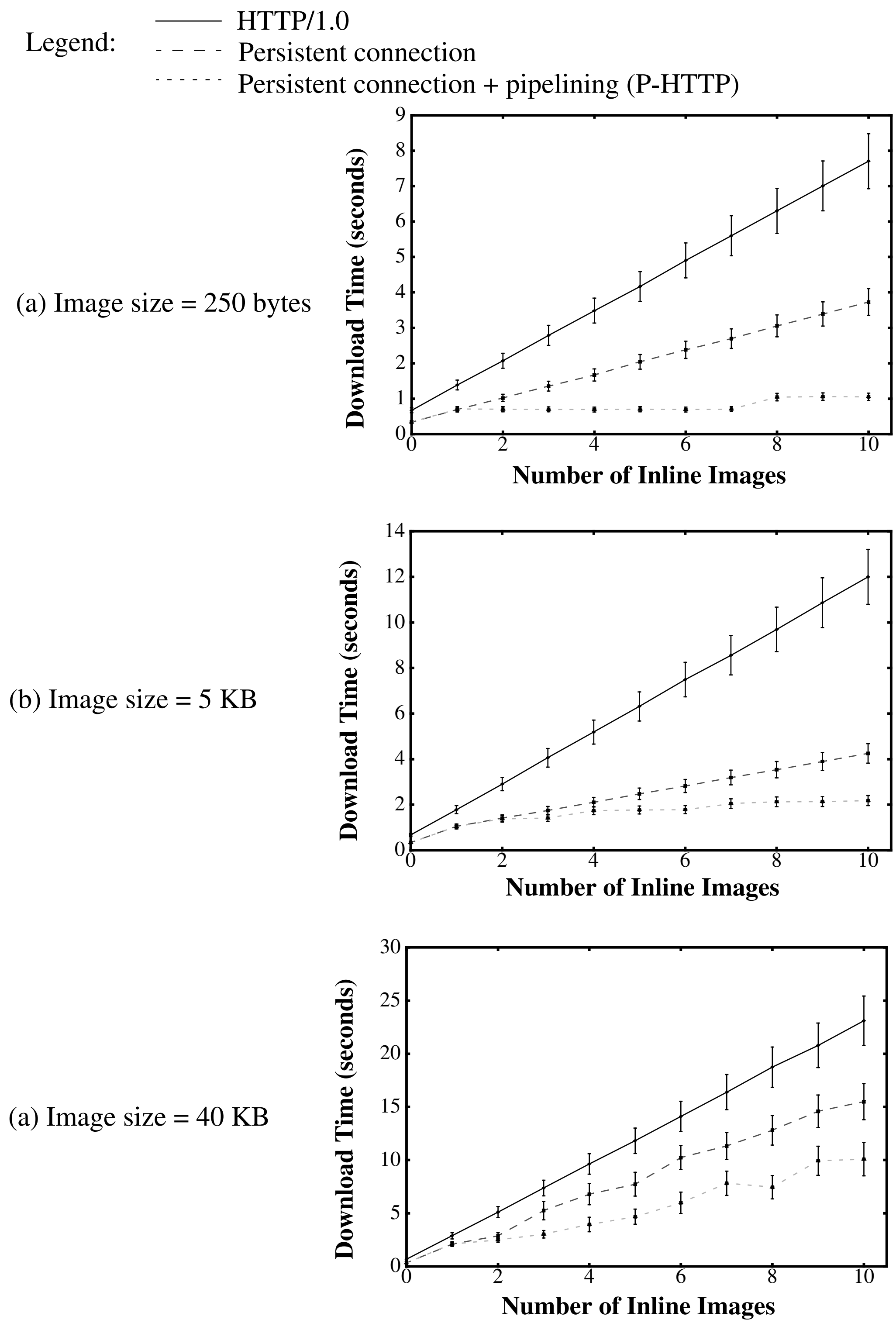



**Figure 5.7 Results of experiments conducted over the DirecPC satellite network. The error bars correspond to 95% confidence intervals for the mean.**

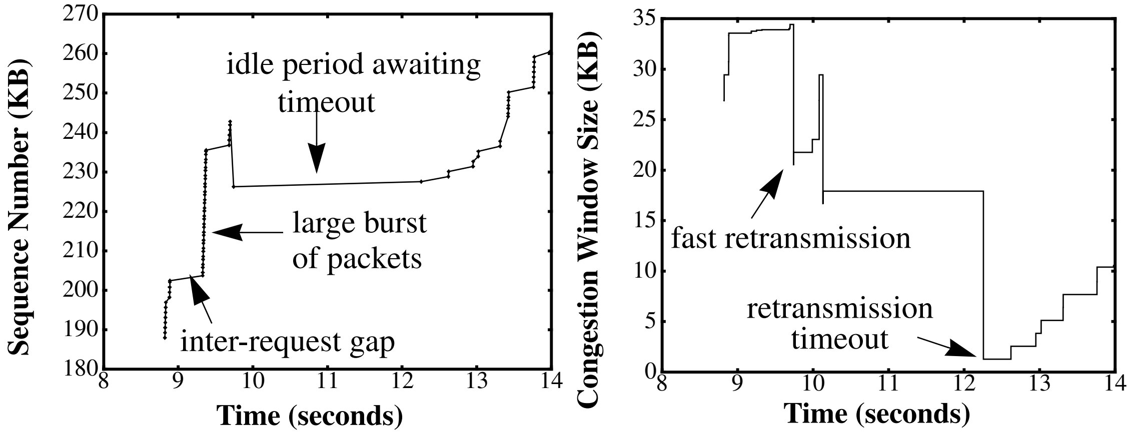



**Figure 5.8 A section of the sequence number and congestion window size traces illustrating the problem that arise when the download of a new inline image begins on a persistent connection but pipelining is not in use. The inline image size was 40 KB and the experiment was conducted over the DirecPC network.**

## 5.5.3 Interaction of P-HTTP with TCP Congestion Control

We now discuss two ways in which persistent connections and pipelining interact with the TCP congestion control algorithm. The first interaction arises when a persistent connection is used to download a Web page with several inline images, but the requests and responses are *not* pipelined. In this case, there is a period of duration approximately one RTT between successive image downloads during which the server idles waiting for the client to send its next request. Ack clocking dies down during this period because the server has no more data to send pending the client's new request. However, the server's congestion window size for the persistent TCP connection remains unchanged (because the idle period is not long enough for the congestion window size to be reset as discussed in Section 2.1.5). So when a new request arrives, the server starts off by sending a large burst of packets back-to-back. This could lead to heavy packet loss, often forcing the server to suffer a retransmission timeout. Figure 5.8 illustrates this using a sequence number trace and a corresponding congestion window size trace. Note that the use of pipelining avoids this problem by eliminating the gap between the download of successive images.

The second interaction with the TCP congestion control algorithm arises when the persistent connection remains idle between successive Web page downloads for long enough that the server resets its congestion window size. Such pauses are common during a Web browsing session as the

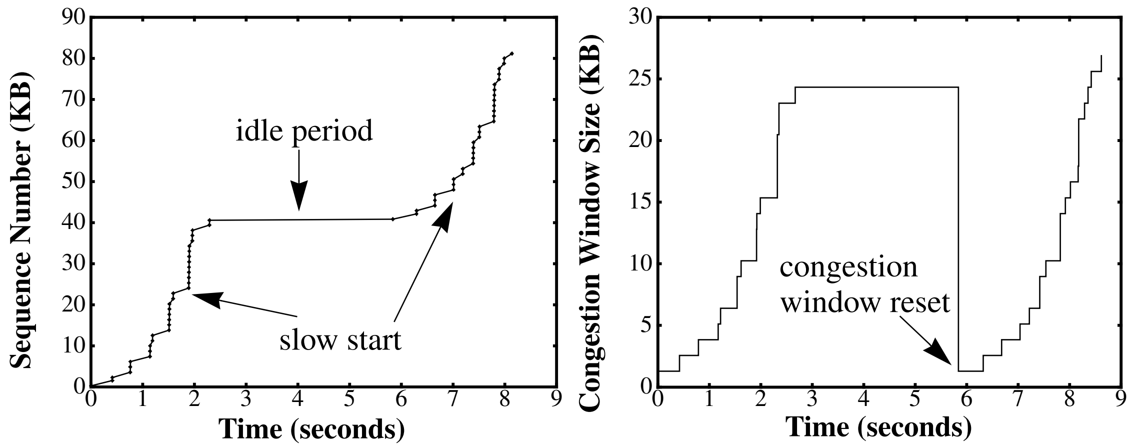



**Figure 5.9 Sequence number and congestion window plots illustrating the problem of repeated slow start when two downloads over a persistent connection are spaced apart in time. This experiment was conducted over the DirecPC satellite network.**

user usually spends some time perusing the retrieved material before initiating the next download. As a consequence, the entire penalty of slow start is incurred when the connection resumes activity, presumably to download a new Web page. This problem has been reported in the literature (for example, in [44], where it is called the *slow start re-start problem*), and Figure 5.9 illustrates it using traces. In Chapter 8, we present *TCP fast start,* our solution to this problem. TCP fast start tries to avoid repeated slow start by caching and reusing the congestion window size, but at the same time taking care to avoid a penalty when the cached information is stale.

We now turn to other work that has adopted and extended the key ideas of persistent connections and pipelining in P-HTTP.

## 5.6 Extensions of P-HTTP

We discuss the use of persistent connections in HTTP/1.1 and alternative ways of multiplexing data within a persistent connection.

### 5.6.1 Persistent Connections in HTTP/1.1

Our P-HTTP work has had a significant impact on the new version of the HTTP protocol, HTTP/1.1 [30], which was standardized in 1997. Our work is cited in the standard document, RFC 2068. Among the various new features and optimizations included in HTTP/1.1 are persistent connections and pipelining, as suggested by our work. The use of persistent connections, while not mandatory, is prescribed as the default behavior, i.e., clients and servers can assume that their peer supports persistent connections unless explicitly notified otherwise. A chunked encoding scheme, like the block-by-block transmission scheme described in Section 5.2.1, is used to when the content-length is not known. The standard allows clients to have up to 2 persistent connections open to the server, instead of just 1. The rationale is to decrease the chance of having a short transfer blocked by a previous long transfer. We discuss this head-of-the-line blocking problem further in Section 5.7, as a lead in to our work on TCP sessions. A performance analysis of HTTP/1.1 appears in [80].

### 5.6.2 Application-independent Multiplexing Protocols

P-HTTP multiplexes several logical data streams onto a TCP connection. But this is done with the active involvement of the client and server applications. An alternative would be to abstract out the multiplexing functionality into a separate library, such as the socket library that interfaces applications to the TCP/IP protocol stack. Session Control Protocol (SCP) [100] and the Session Multiplexing Protocol (WebMUX) [36] are two examples of such an approach. These protocols support more fine-grained interleaving of data belonging to different logical streams than either P-HTTP or HTTP/1.1. However, they suffer from some of the same drawbacks as P-HTTP and HTTP/1.1. First, applications would still have to be modified, or at least re-linked, to work with the new socket library. Second, the head-of-line blocking problem, alluded to in Section 5.6.1, still persists. We discuss these in more detail next.

## 5.7 Limitations of P-HTTP

In spite of the performance benefits, P-HTTP has its limitations. Some of these arise from the interaction with the TCP congestion control algorithm, while others arise because of the restrictiveness of the TCP service model. Many of the limitations apply equally to alternative application-level multiplexing techniques such as MUX and SCP discussed in Section 5.6.2.

1. The reliable, ordered byte-stream service provided by TCP causes logically-independent data streams that are multiplexed onto a single TCP connection to get coupled together in undesirable ways. For instance, data belonging to a certain inline image would get held up whenever data belonging to a different image, but which happens to lie earlier in the sequence number space, gets lost in the network and is awaiting retransmission[3]. Another consequence of the coupling is that it is difficult or even impossible to treat individual Web objects (such as inline images) differently in the interior of the network. Having such a capability would be useful, for instance, to selectively intercept certain Web requests and responses at a transparent cache (e.g., [19]) but let the others go through without interference.

   The approach adopted by HTTP/1.1 to address this problem is to allow the setting up of two persistent connections between the client and the server rather than just one. Clearly, this does not really solve the problem because the number of logical data streams could well exceed two. Furthermore, increasing the number of connections is counter to the basic goal of sharing and reusing connections.

2. The coupling of logically separate data streams can lead to denial-of-service attacks, as noted in [35]. A client could connect to a server via a proxy and start downloading a large file. This would tie up the persistent connection between the proxy and the server for a long time, effectively shutting out other clients that wish to connect to the same server via the proxy.

3. Sharing a connection between multiple data streams requires either that the application do the multiplexing explicitly or that it use a special communications library that does the multiplexing. These require the application to be modified, or at the least relinked with the new library. Since both the communicating peers (i.e., client and server in the context of the Web) have to agree on the framing format for multiplexing, modifications would be required at both ends.

3. This is akin to the *head-of-line blocking* problem in input-buffered switches.

Such modifications are undesirable when one considers that the need to support short and bursty data streams efficiently could extend beyond the Web, in particular to legacy applications, such as distributed database transaction processing, that may be difficult or expensive to modify.

4. It is difficult or impossible for separate application processes to share a single TCP connection. Therefore a Web browser and a helper application, both of which wish to communicate directly with a server, would end up using separate connections.

5. As mentioned in Section 5.5.3, the use of a persistent connection does not avoid the invocation of slow start each time the connection resumes activity after a pause. Since the total size of a Web page, including all embedded components, itself tends to be small on average (typically 20-30 KB), the cost of the repeated slow start can be quite significant.

## 5.8 Summary

In this chapter, we have discussed persistent-connection HTTP (P-HTTP), our initial solution to the performance problems that afflict HTTP/1.0. The two main ideas underlying P-HTTP are persistent connections and pipelining. The use of persistent connections amortizes the cost of connection set up over multiple Web page downloads. It also amortizes the slow start penalty over the download of multiple components (such as inline images) that constitute individual Web pages. Pipelining requests and responses over the persistent connection helps transfer the components of a Web page back-to-back, without any pauses in between, saving at least one RTT per component. Results from experiments both over a terrestrial wide-area network as well as a satellite-based one show that P-HTTP results in a significant improvement in performance, with a reduction in download time by up to a factor of 7.

The primary drawback of P-HTTP is that it couples together data streams that are otherwise independent. It is interesting to note that using a separate TCP connection for each logical data stream would avoid this problem. However, as discussed in Chapter 4, such an approach can have an adverse effect on the performance of the network as a whole. In the next chapter, we present our new solution, called *TCP session*, which uses a TCP separate connection for each logical data stream but yet does not suffer from the same performance drawbacks.

# Chapter 6

# TCP Session

In Chapter 4 and Chapter 5, we analyzed the performance of HTTP/1.0 and P-HTTP and identified their strengths and limitations. In this chapter, we present our new solution, which we call *TCP session*. We first provide the motivation for developing TCP session and then describe its design in detail. Next, we discuss implementation details, including the our implementation in the BSD/OS TCP/IP stack. We then present detailed performance results, both from simulation and from experiments using our implementation.Finally, we summarize the benefits and the limitations of TCP session. The latter motivates our work on TCP fast start presented in Chapter 8.

## 6.1 Motivation

In previous chapters we have discussed the problem of Web data transport in the context of three different schemes: HTTP/1.0 with sequential connections, HTTP/1.0 with concurrent connections, and P-HTTP. Table 6.1 summarizes the key advantages and disadvantages of each approach.

We note that each of these schemes has its share of pros and cons. Closer examination reveals that HTTP/1.0 with concurrent connections and P-HTTP complement each other's strengths and weaknesses. Our goal, therefore, is to design a solution that combines the strengths of both but is not

| | Pros | Cons |
|---|---|---|
| HTTP/1.0 with sequential connections | • Easy to write application using standard sockets API | • Sequential ordering imposed on logical data streams<br>• Overhead of repeated slow start<br>• Poor loss recovery |
| HTTP/1.0 with concurrent connections | • No coupling between logical data streams<br>• Easy to write application using standard sockets API | • Less responsive to congestion<br>• Overhead of repeated slow start<br>• Poor loss recovery<br>• Promotes unfairness |
| P-HTTP | • Good congestion control and loss recovery properties<br>• Overhead of slow start amortized over multiple transfers | • Couples together logical data streams causing problems such as head-of-line blocking<br>• Requires multiplexing support in application/socket library |

**Table 6.1 Summary of the pros and cons of various approaches to Web data transport.**

encumbered by the weaknesses of either. Specifically, we desire a solution with the following properties:

- Applications should be free to use as many logical data streams as they wish to communicate between hosts. These data streams should not be temporally ordered.
- The use of multiple data streams should not have an adverse effect on the performance of either the application in question or the network as a whole. In particular, it should not impact the efficacy of congestion control and/or loss recovery mechanisms.
- The solution should conform to the principle of wide applicability discussed in Section 1.8. By implication, it should not be tied to a specific application or application-level protocol (such as HTTP). Also, it should require little or no support from applications, and should have little or

no dependence on their structure in terms of their organization into processes (for instance, the organization of a Web client into a browser process and helper processes for individual MIME types).

- In accordance with the principle of incremental deployment discussed in Section 1.8, it should be possible to deploy the solution without any major changes to the network. A partial deployment should be robust and should also yield some of the performance benefits of a complete deployment.

The solution we develop enables applications to map individual data streams onto separate TCP connections. The aggregate of data streams is tied into what we call a *TCP session*[1]. This solution satisfies most of the requirements listed above. In the sections that follows, we quantify its performance benefits and also point out its limitations. The also discuss how the latter is addressed in later chapters.

## 6.2 TCP Session Overview

We observe that many of the difficulties in the context of Web data transport arise due to the coupling of two distinct elements of TCP functionality into the entity that is a TCP connection:

1. Reliable, ordered byte-stream service with flow control.
2. Congestion control and loss recovery.

Only the former is visible at the application level. An application knows that each byte of data it sends over a TCP connection will be delivered reliably to the receiver without any reordering. A receiving application can effect flow control by controlling how fast it reads in data from a TCP connection. The latter part of TCP functionality happens under the covers, hidden from the application. However, it is critical both for supporting the reliable, ordered byte-stream service abstraction (because loss recovery is needed for reliable data delivery) and for good performance.

Mapping several logical data stream onto a single TCP connection (as in P-HTTP) causes ordering to be imposed on data that belongs to different data streams. As a result, flow control actions on

1. For ease of exposition, we use the term "TCP session" both to refer specifically to such an aggregate of connections and also to refer to the technique as a whole.

one data stream also impact the others, which is clearly undesirable. Mapping the data streams onto separate TCP connections avoids this problem, but introduces a new one — congestion control and loss recovery happen independently for each connection, which decreases their efficacy (as discussed in Section 4.3.2).

Our solution to these problems is to separate TCP functionality into two parts:

1. Reliable, ordered byte-stream service with flow control is provided on a per-TCP connection basis.
2. Congestion control and loss recovery are done in an integrated fashion across the set of TCP connections between a pair of hosts (such as a Web server and a client). We refer to this set of connections as a *TCP session*.

The rationale for integrating congestion control and loss recovery across the set of connections in a TCP session is that all of the connections are likely to traverse the same path through the network and experience similar levels of congestion. The choice of how many logical data streams (and hence TCP connections[2]) to use is purely an application-level decision, so it should not have any impact on the functioning of congestion control and loss recovery.

The TCP session abstraction enables applications to use as many TCP connections as they wish without impacting performance, either adversely or favorably[3]. So, for instance, an application may choose to break up an image file into well-defined sub-component that can be rendered independently of each other. By transferring and rendering the sub-components concurrently, the application can improve the performance perceived by a human user, although the performance of the underlying data transport may remain unchanged.

By default, the TCP session abstraction is transparent to applications. Just as before, applications open and use TCP connections using the standard sockets API. There is no need for applications to be modified or even re-linked. In addition, however, TCP session gives applications the opportunity to control the scheduling of connections within a TCP session. In this chapter, we confine our-

2. From the viewpoint of an application, a TCP connection is synonymous with a logical data stream.
3. A positive correlation between the number of TCP connections used and the performance achieved (as is the case with standard TCP in certain situations) could encourage applications to be greedy and launch several connections, to the detriment of the network as a whole.

selves to the default operation of TCP session, and defer discussion of application-controlled scheduling to Chapter 7.

The aggregation of connections into a TCP session can happen at several different levels of granularity, such as individual applications, all applications launched by an individual user, etc. However, since, in this chapter, we are focussing on the transparent operation of TCP sessions, we only discuss integration at the granularity of host-pairs (such as a client-server pairs). Many of the ideas discussed in this context carry over to other levels of granularity of integration, which we motivate and discuss in Chapter 7.

Finally, the operation of TCP session is confined to the sending side of data. So each of the communicating peers can independently choose whether or not to implement it. In practice, it would probably be more important to incorporate the algorithm in Web server hosts than in Web client hosts because the former is primarily a sender and the latter primarily a receiver of data.

## 6.3 Design and Operation of TCP Session

We now discuss the design and operation of TCP session in detail. We discuss four components of TCP session in turn:

1. Session Control Block
2. Integrated Congestion Control
3. Connection Scheduling
4. Integrated Loss Recovery

### 6.3.1 Session Control Block

The separation of TCP functionality that TCP session entails a corresponding separation of TCP state into two parts — that needed to support the reliable, ordered byte-stream abstraction and flow control, and that needed for congestion control and loss recovery. The latter is encapsulated into what we term as a *session control block* (SCB) maintained in the sender's protocol stack.

The SCB contains state variables that are shared among the connections in a TCP session. These variables, which are needed for congestion control and loss recovery, are no longer part of the TCP control block (TCB) of each individual connection. Only variables needed for the reliable, ordered byte-stream service and flow control provided by each connection are retained in the corresponding TCB.

The contents of the per-session SCB include:

1. Session congestion window (*session_cwnd*).
2. Session slow start threshold (*session_ssthesh*).
3. Amount of outstanding (i.e., unacknowledged) data across all the connections in the session (*session_ownd*).
4. Smoothed estimates for the round trip time and its variance (*srtt*, *rttvar*).
5. List of connections within the session (*connlist*)
6. List of unacknowledged segments sent on connections in the session, sorted by their time of transmission (*seglist*).

The contents of the per-connection TCB include:

1. Highest sequence number of data transmitted (*snd_max*).
2. Sequence number of the first unacknowledged byte of data (*snd_una*).
3. Receiver advertised flow control window (*rcv_adv*).

Note that we have only listed the subset state variables in the SCB and the TCB that are relevant to the discussion that follows.

A TCP session and the corresponding SCB come into existence automatically when the first connection between a pair of hosts is opened. These remain in existence so long as there is at least one connection open between the pair of hosts. When there are no open connections left, the TCP session is terminated and the SCB is de-allocated. However, in certain situations it may be desirable to retain the SCB even when there are no active connections. For instance, the SCB could hold persistent state for use by TCP fast start, which we present in Chapter 8.

Next, we discuss integrated congestion control.

### 6.3.2 Integrated Congestion Control

The key observation that motivates integrated congestion control is that it is *how much* data rather than *what* data that matters as far as the load and the state of congestion in the network are concerned. In other words, it is irrelevant from the viewpoint of congestion control whether the data being transferred between a pair of hosts is spread across multiple TCP connections or concentrated in a single connection. It only matters how much data is outstanding in the network. Accordingly, congestion control should not be impacted in any way by the number, duration, or other characteristics of concurrent connections launched by applications.

For this purpose, a sender implementing TCP session maintains a unified congestion window, *session_cwnd*, for the set of connections between a pair of hosts. This tracks the available capacity along the network path between the hosts. The sender also maintains a count of the total amount of outstanding data, *session_ownd*, across all the connections. The session congestion window places a limit on the total amount of outstanding data that the set of connections in the session can have, so the sender is entitled to send data so long as *session_ownd* is less than *session_cwnd*. Since there may be multiple connections within the TCP session, there is the question of which connection(s) should be picked when there is the opportunity to send data. We discuss this scheduling issue briefly in Section 6.3.3, and in more detail in Chapter 7.

The dynamics of *session_cwnd* are much like that of a *single* connection using standard TCP. At the time the first connection in a session is opened, *session_cwnd* is initialized to one segment. Since there is no outstanding data at this point, *session_ownd* is set to zero. As new data is sent out on any of the connections in the session, *session_ownd* is incremented. As acks indicating the successful delivery of this data are received, *session_ownd* is decremented and *session_cwnd* is grown. Initially, the sender grows *session_cwnd* at an exponential rate, in accordance with the slow start regime. When *session_cwnd* crosses the slow start threshold, *session_ssthresh*, the session switches to linear growth at the rate of one segment per RTT.

So long as there is at least one connection open in the session, the opening and closing of additional connections has no impact on the dynamics of the congestion window. This is in keeping with our goal to decouple the opening of a new connection (which is an application-level decision) from congestion control (which impacts the performance of the network as a whole). In particular, *session_cwnd* is not incremented just because a new connection has been opened. Instead, the new connection shares the existing session congestion window with other connections in the session, as we discuss in Section 6.3.3.

As in standard TCP, the loss of a segment is treated as a sign of congestion. In response, the sender backs off, except that, unlike in TCP, the entire session is backed off, not just the connection that experienced the loss. This is motivated by the observation that the loss of a segment of a connection is caused by the overall load imposed on the congested link by all the connections in the session (and also unrelated connections originating at other hosts). The "fate sharing" imposed by tail-drop gateways, that are predominant in the Internet, implies that there may be little correlation between the load that a particular connection imposes on the network and the amount of packet loss it experiences. So when it detects an implicit congestion signal in the form of a packet loss, the most that a sender can do to alleviate the congestion is to cut back the load offered by all the connections in the corresponding TCP session (which presumably all share the congested link).

Ideally, this cut back should be applied to all connections that contribute to congestion on that link, but practical difficulties make it difficult to do so:

- A sender that experiences a packet loss on a particular connection does not know which other connections of its to other hosts also share the same congested link.
- The sender does not know which other hosts have connections traversing the same congested link. Even if this knowledge were available, it would probably be too expensive (and perhaps also counter-productive because of the increase in traffic) to explicitly deliver this congestion signal to each individual host.

Therefore, our response to packet loss is confined to the corresponding session. The sender halves the congestion window of the corresponding session, provided the loss is detected via data-driven means. The session slow start threshold, *session_ssthresh*, is then set to this new, halved value of *session_cwnd*. On the other hand, if the sender is forced to time out, it sets *session_ssthresh* to half of *session_cwnd*, and resets *session_cwnd* back to one segment. These steps ensure that unlike in the case of independent TCP connections (Section 4.3.3), the congestion behavior of a TCP session is not impacted by the number of connections that applications choose to open.

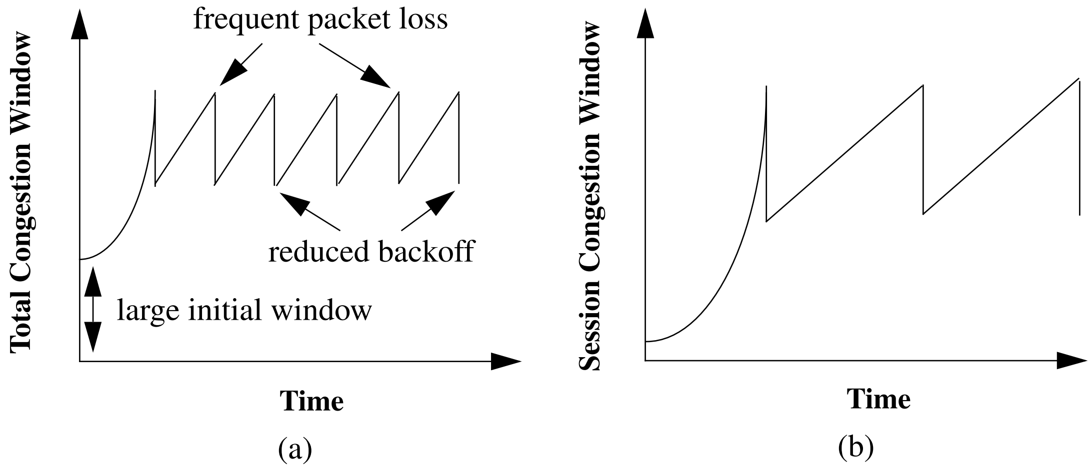



**Figure 6.1 A schematic depiction of the congestion window dynamics when several connections are launched simultaneously between a pair of hosts. (a) corresponds to the case of independent TCP connections, and (b) to TCP session. The total congestion window in (a) is obtained by adding up the windows of the individual connections.**

Figure 6.1 contrasts the congestion window dynamics of TCP session (Figure 6.1(b)) with that of independent TCP connections (Figure 6.1(a)). Note that these are schematics rather than actual traces. In the latter case, the total congestion window size is computed by adding up the window sizes of the individual connections. We note several significant differences that highlight the benefits of TCP session:

1. When *n* independent connections start up, the total initial congestion window size is *n* segments. In contrast, the initial window size with TCP session is one segment. The large initial window size can be advantageous if the network is able to absorb the burst of *n* packets, but, if not, it could lead to increased packet loss, and consequently degraded performance.

2. During the linear phase, the congestion window of *n* connections grows at the rate of *n* segments per RTT, compared to 1 segment per RTT for TCP session. This pushes the window size back more quickly to the point where packet loss occurs, causing more frequent packet losses.

3. With independent TCP connections, only the connection that experiences a packet loss halves its window. In contrast, with TCP session, the entire session window is halved. The reduced backoff in the former case, again, increases the frequency of packet loss.

The evolution of the congestion window in actual practice may be quite different from that depicted in Figure 6.1. This is so because of retransmission timeouts. As we shall see in Section 6.5, independent TCP connections are more prone to timeouts, especially if each of them is short in length.

Next, we turn to connection scheduling.

### 6.3.3 Connection Scheduling

Integrated congestion control entitles the sender to sent data so long as *session_ownd* is less than *session_cwnd*. When a new ack arrives, the sender decrements *session_ownd* to account for the data that has been acknowledged, thus clearing the way for new data to be sent. Unlike in TCP, where the sender is constrained to send new data only on the connection on which the ack arrived, the TCP session sender has the choice of possibly many connections (all within the same session) on which to send new data. In fact, it could choose to send new data on more than one connection. Picking one or more connections for this purpose is the *connection scheduling* problem.

By default, the operation of the TCP session is entirely transparent to applications. As such, all connections within the TCP session are treated equally from the viewpoint of connection scheduling. Connections are scheduled in a *round-robin* fashion. During each round, a connection is eligible to send up to one data segment (of size at most MSS). This ensures that each connection receives an equitable share of the available bandwidth in a consistent manner, i.e., each connection's share is spread out (roughly) evenly over time rather than being concentrated in bursts. The benefit of such scheduling becomes clear in Section 6.5.1 where we contrast it with independent TCP connections that compete with each other.

A connection may not be in a position to send any data when its turn comes because of two reasons — there is no more data left to send and/or the receiver-advertised flow control window (which is still maintained on a per-connection basis) does not permit sending any more data. The former could happen because the sending application is either slow in generating data or has finished sending all its data. The latter could happen because the receiving application is slow in consuming data. In either case, the connection scheduler simply skips over such connections until they are in a position to send data.

It is instructive to note that even if a subset of connections (i.e., data streams) in a TCP session stall, this has no impact on the rest of the data streams. This is in contrast to the head-of-line blocking problem that afflicts application-level multiplexing solutions such as P-HTTP.

In Chapter 7, we discuss more a sophisticated connection scheduling algorithm. We also discuss the benefits of exposing connection scheduling to applications via an appropriate API. used by applications to vary weights dynamically, in Chapter 7. Next, we discuss the final component of TCP session, integrated loss recovery.

### 6.3.4 Integrated Loss Recovery

We now discuss integrated loss recovery, the third component of TCP session. Integrated loss recovery is motivated by the observation in Section 4.3.2 the short length of typical Web connections makes timeouts the predominant means of loss recovery, thereby degrading performance. An important benefit of P-HTTP is that a persistent connection, which tends to be longer than individual HTTP connections, makes data-driven loss recovery more effective, thereby avoiding many of the timeouts. The goal of integrated loss recovery is to achieve a similar performance gain but in the context of TCP session, with its possibly numerous connections.

The key to effective data-driven loss recovery is having a long enough data stream that the sender has the opportunity to detect packet loss by observing the sequence of acks received in response to the (several) data segments that it sends out. For instance, consider the case of the TCP fast retransmission algorithm illustrated in Figure 6.2. It is only because the sender sends more segments beyond the one that happens gets lost that it receives multiple *duplicate acks* for the lost segment, and so is able to reliably infer and recover from the loss. The TCP sender waits for a threshold number (typically 3) of duplicate acks to provide a certain level of protection against packet reordering.

When a connection that is short in length suffers a packet loss, it often does not receive a sufficient number of duplicate acks. However, it is still possible for the sender to detect the packet loss by observing acks for data segments sent on other connections in the same session. This is the key to integrated loss recovery. Analogous to duplicate acks, we define the notion of a *later ack* as follows:

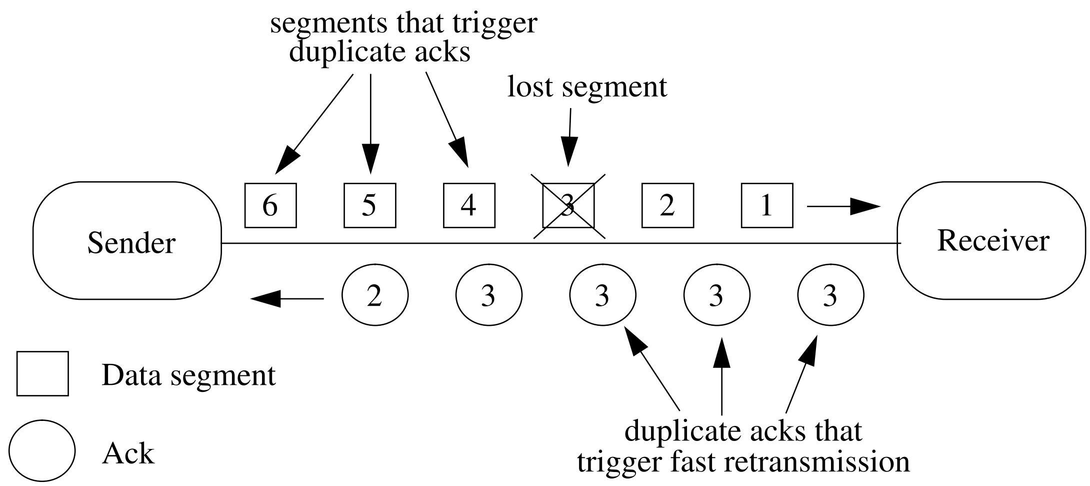



**Figure 6.2 An illustration of TCP fast retransmission. If data segments 4, 5 and 6 had not been sent, the duplicate acks would not have been received and fast retransmission would not have been triggered. Note that TCP acks specify the sequence number of the next segment expected, so in our illustration an ack has a sequence number that is one greater than that of the data segment that triggered it.**

**Definition:** An ack is termed as a *later ack* for an unacknowledged segment, *S*, belonging to a connection, *C*, if it acknowledges one or more segments sent after *S* on one of the other connections in the same TCP session as *C*.

Figure 6.3 illustrates the notion of later acks for the case of two connections in a TCP session. Acks numbered 2, 3 and 4 (shown shaded) are later acks for (the lost) segment 2 (shown in white). Later acks for an unacknowledged segment can be used in a way similar to duplicate acks to reliably infer packet loss while at the same time ruling out packet reordering as the underlying cause. This is because the segment(s) acknowledged by a later ack would presumably have (successfully) traversed the same path as the unacknowledged one. If a sufficient number of such *later* segments reach the receiver, chances are that the unacknowledged segment, which was sent prior to these later segments, was lost. The threshold for the number of later acks (plus duplicate acks) needed to reliably infer packet loss could be set to be the same as the TCP fast retransmission threshold, i.e., 3, to provide a similar level of protection against packet reordering[4].

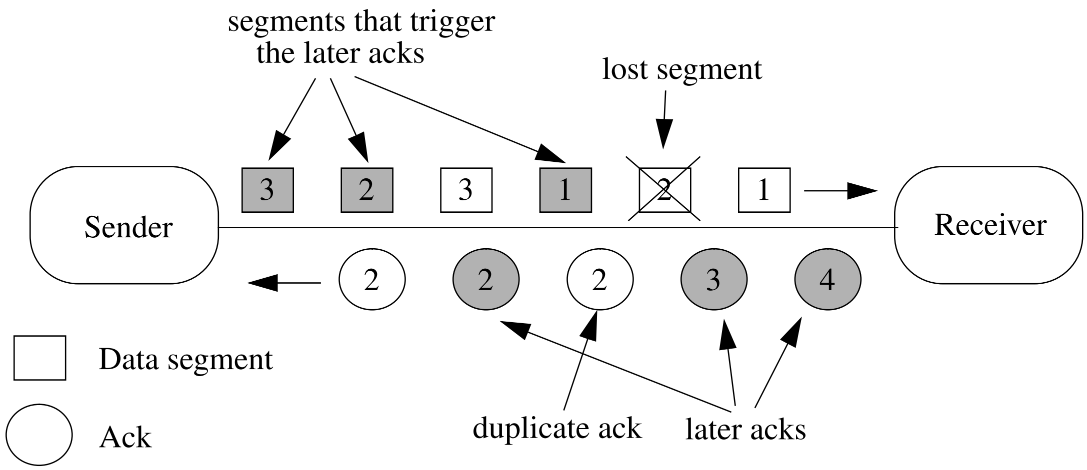



**Figure 6.3 An illustration of later acks in the context of two TCP connection (the white one and the shaded one) in a TCP session.**

There are two potential problems with using later acks:

1. TCP receivers often employ the delayed ack algorithm [101] where they only send an ack for every other data segment received. So the lack of an acknowledgement for a segment may simply be because the receiver is waiting for a second segment to arrive on the same connection before acknowledging them both. In the meantime, however, later segments on other connections may arrive in sufficient numbers that the receiver sends back acks for them. In such a situation, the sender may mistakenly assume that the missing segment is lost. We address this problem when we develop the loss recovery rules in Section 6.3.4.1.

2. The ack for a segment may get lost. When the sender subsequently receives later acks for this segment, it may mistakenly assume that the segment itself had been lost. This situation arises fundamentally because an ack on a connection does not explicitly convey any information about segments that have been successfully delivered on other connections. We defer the dis-

4. Links composed of multiple physical channels bundled together (e.g., a dual-channel ISDN line) may make reordering a more common occurrence. This may require revising the TCP fast retransmission threshold. We view this issue as orthogonal to integrated loss recovery and TCP session. A solution developed in the context of TCP can be translated to a corresponding one in the context of TCP session.

cussion of this issue to Chapter 7, where we evaluate the potential for false retransmissions, both via simulation and via experiments on the Internet, and propose a simple modification to the TCP receiver algorithm to alleviate this problem.

Next, we discuss the specifics of the integrated loss recovery algorithm, with the explicit goal of addressing the delayed ack problem.

### 6.3.4.1 Loss Recovery Rules

We now discuss in detail how the sender uses duplicate ack and later ack information to infer packet loss. For each unacknowledged segment in the session, the sender maintains an up-to-date count of the number of duplicate acks (*dupacks*) and later acks (*lateracks*). We discuss implementation details in Appendix B.

We specify a set of three rules that the server uses to infer packet loss. The first rule is the same as standard TCP fast retransmission. We denote the duplicate ack threshold by *thresh* (with a default value of 3).

***Rule #1:*** *If a segment is the unacknowledged segment with the smallest sequence number on its connection (i.e., the left-most one in the window) AND it has at least thresh duplicate acks AND it has not already been retransmitted, then it is eligible for retransmission.*

Note that the check for the segment being the left-most one in the window is superfluous in this case because only such a segment can possibly have a duplicate ack for it. However, we retain this clause for consistency across the three rules.

The second rule pertains to the case where a combination of duplicate and later acks have been received. As mentioned in Section 6.3.4, later acks by themselves do not conclusively point to a packet loss because of the possibility of the delayed ack algorithm being employed by the receiver. However, if at least one duplicate ack has been received, then this rules out the delayed ack algorithm as the reason why no ack has yet been received for the unacknowledged segment. Therefore, the second rule is:

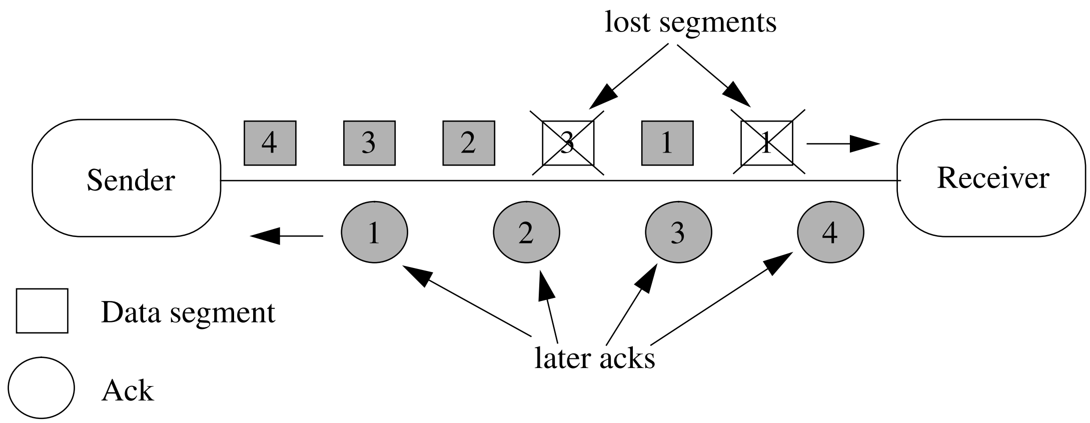



**Figure 6.4 An illustration of loss recovery rule #3. Based on the sequence of later acks received, the sender can reliably infer the loss of segment 1 of the white connection (which is the unacknowledged segment with the smallest. sequence number on that connection).**

***Rule #2:*** *If a segment is the unacknowledged segment with the smallest sequence number on its connection AND it has at least thresh duplicate and/or later acks combined AND it has at least one duplicate ack AND it has not already been retransmitted, then it is eligible for retransmission.*

This rule is illustrated in Figure 6.3, where segment 2 of the white connection receives one duplicate ack and three later acks (one more than necessary assuming thresh is set to 3), and so is eligible for retransmission.

The third rule pertains to the case where only later acks (and no duplicate acks) have been received. The key observation is that even if a receiver employs the delayed ack algorithm, it is required to at least send an ack for every other segment received. So if the sender receives the threshold number of later acks (or perhaps more) for two or more segments in a connection, it can safely conclude the oldest among them must have been lost. Therefore, the third rule is:

***Rule #3:*** *If a segment is the unacknowledged segment with the smallest sequence number on its connection AND each of two or more segments on that connection have at least thresh duplicate and/or later acks combined AND it has not already been retransmitted, then it is eligible for retransmission.*

**Rules for Integrated Loss Recovery:**

A segment, *seg*, belonging to connection, *conn*, is a candidate for retransmission if and only if at least one of the following conditions is true:

1. *seg->seq == conn->snd_una && seg->dupacks >= thresh && NOT_REXMT(seg)*

2. *seg->seq == conn->snd_una && (seg->dupacks + seg->lateracks >= thresh) && (seg->dupacks >= 1) && NOT_REXMT(seg)*

3. *seg->seq == conn->snd_una && (seg->dupacks + seg->lateracks >= thresh) && (seg2->dupacks + seg2->lateracks >= thresh) && NOT_REXMT(seg)*

**Note:**

a. *conn->snd_una* denotes the highest ack on the connection *conn*.

b. *seg2* is a second segment belonging to the same connection as *seg*.

**Figure 6.5 Rules for integrated loss recovery.**

Note that a particular ack may be a later ack for more than one segment on a connection. For instance, in Figure 6.4, ack 1 on the shaded connection is a later ack for just segment 1 of the white connection, but acks 2, 3 and 4 are later acks from both segments 1 and 2.

The three rules for integrated loss recovery are summarized Figure 6.5. If the sender is unable to reliably detect a loss using these rules, it falls back to a retransmission timeout.

We briefly discuss the implementation of TCP session in BSD/OS next. Further details with regard to integrated loss recovery are presented in Appendix B.

## 6.4 Implementation in BSD/OS

We have implemented TCP session both in the *ns* network simulator [81] and the BSD/OS 3.0 TCP/IP stack [16]. The implementation details outlined above form the basis for both implementations. Here, we briefly discuss our BSD/OS implementation from the viewpoint of how it interfaces with the existing TCP/IP code.

Our implementation of TCP session operates at the granularity of host pairs. It is implemented as a set of functions that are invoked from a small number of points in the TCP code. The set of functions includes those to create a new session, add a new connection to a session, process incoming acks, schedule the sending of data, and destroy a session.

When a TCP connection transitions to the *ESTABLISHED* state, *tcp_input()* calls a new routine, *session_addconn()*, to add the connection to the session between the two hosts. We need to wait until the *ESTABLISHED* state because only then are the identities of the two end hosts known. *Session_addconn()* adds a pointer to the connection's TCB in the *session PCB* (SCB), if it already exists. If not, it first calls *session_create()* to allocate and initialize a new SCB. Although *session_cwnd*, *session_ssthresh*, *srtt* and *rttvar* are maintained on a session-wide basis, we do not remove the corresponding variables from the TCBs of the individual connections. This is to minimize the amount of change needed to the existing TCP code. We simply ignore these variables in the individual TCBs.

When *tcp_output()* is called, it in turn invokes *session_output()*. *Session_output()* first makes sure that the *session_cwnd* does permit the sending of a new segment. Then, among the connections in the session that have data to send, it picks the one that should send a segment next according to the weighted hierarchical round-robin schedule. Note that this connection could, in general, be different from the one that invoked *tcp_output()* to begin with. However, an exception is made if the original connection wanted to send out a segment with a control flag, such as *FIN* or *RST,* turned on. When a segment is to be sent out on a particular TCP connection, *tcp_output()* is invoked with a pointer to the appropriate TCB passed as an argument. A special flag is used to indicate that *tcp_output()* is being invoked from *session_output()*, so it should *not* turn back and call *session_output()* again.

When an ack is received, *tcp_input(),* as usual, updates the highest ack information and clears out acknowledged data from its buffer. However, it does not do duplicate ack processing. Instead, it defers to *session_input()* which detects packet reordering by looking for both duplicate acks and later acks. If necessary, *session_input()* invokes *tcp_output()* to retransmit the appropriate segment.

The TCP timer function, *tcp_slowtimo()*, is invoked periodically to scan through the list of TCBs to determine whether a timeout is due for any connection. We augment this code in two ways. First, in addition to the list of TCBs, a separate list of SCBs is also scanned. For any session with a timeout pending, *session_timeout()* is invoked. Second, a retransmission timeout event for a connection (TCB) that is part of a session is simply ignored.

Finally, when the last connection in a TCP session is closed, *session_destroy()* is called to clean up state and free up the SCB. However, when TCP fast start (Chapter 8) is combined with TCP session, the SCB is retained to preserve state for future use.

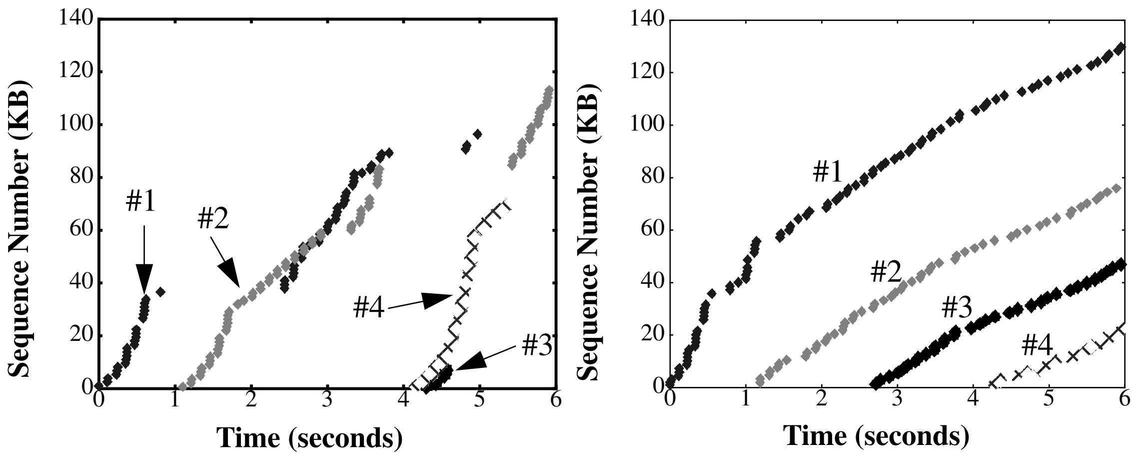
## 6.5 Performance Results

We conducted several experiments, using both our implementation in the *ns* simulator and that in BSD/OS, to evaluate the performance of TCP session under a variety of conditions. Much of our emphasis here is on the former because it enables us to conduct experiments on a larger scale and measure the various aspects of performance is greater detail. However, we begin with a simple experiment conducted using our implementation in BSD/OS.

### 6.5.1 Competition versus Sharing between Concurrent Connections

This experiment involves 4 TCP connections between a pair of hosts, launched about 1.5 seconds apart. This emulates a Web page download with a separate TCP connection for each of 4 inline images. We used our link emulator module (Chapter 3) to dial in a bottleneck link bandwidth of 1.5 Mbps (T1 speed), an RTT of 100 ms and a buffer size of 4 packets.

We conducted the experiment using two configurations — (a) independent TCP connections, and (b) TCP session. The sequence number plots for the two cases are shown in Figure 6.6. We observe a significant differences in the two cases. The performance of each connection in the case of independent connections is highly variable. There are times during which a subset of the connections makes progress while the rest stagnate. For instance, connection #3, launched at around 3 seconds, starts making progress only after connection #4 starts up. In a real setting, it may be that




(a) (b)

**Figure 6.6 The dynamics of four connections sharing an emulated 1.5 Mbps bottleneck link with (a) independent TCP connection, and (b) TCP session.**

connection #3 is the most important one from an application's viewpoint. But there is little that the application can do to prevent the performance degradation that this connection suffers.

In contrast, the performance with TCP session is much more consistent. Round-robin connection scheduling ensures that each connection gets an equal share of the available bandwidth. When a new connection starts up, the connections that are already active are slowed down a little to accommodate it.

In general, TCP session has three advantages:

1. Integrated congestion control eliminates competition between connections in the same TCP session, thereby reducing the total amount of competition in the network. This helps cut down the packet loss rate.
2. Explicit connection scheduling, perhaps with input from the application, removes uncertainty in how the bandwidth available to a TCP session is shared among its constituent connections. However, uncertainty in the share of the total bandwidth that a session is able to get remains, albeit at a reduced level because of 1 above.

3. Integrated loss recovery improves the chances of data-driven loss recovery, especially when connections are short in length. This decreases the occurrence of retransmission timeouts that lead to long periods of inactivity and poor performance (as evident for connection #1 in Figure 6.6(a)).

It may be argued that the first point listed above only amounts to scaling down the amount of competition in the network by a small factor. It does nothing about the competition between hosts. While this is true, we make two observations. First, there are many situations where the bottleneck link is not shared by multiple hosts. A dialup modem line connecting a client is a common example. In such situations, scaling down the competition by even a small factor helps, as our results in Section 6.5.2 show. Second, even when there are several hosts sharing the bottleneck link, the other two points listed above still stand. Our experimental results highlight this.

### 6.5.2 Simulation Results

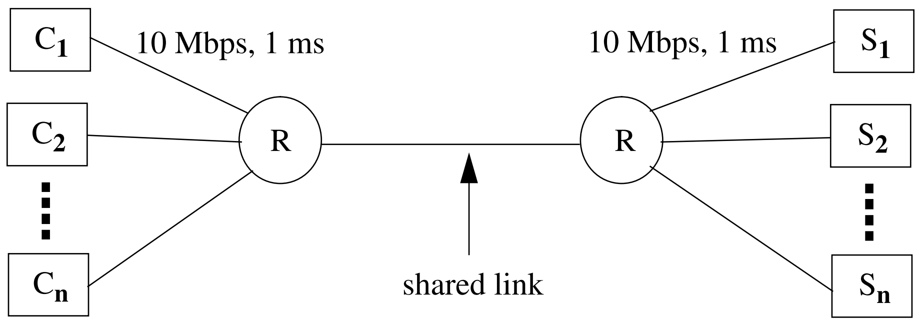
We now present detailed results from simulations experiments. The topology used for these experiments is shown in Figure 6.7. It consists of a large number of client-server pairs that share a common link. Depending on the speed of this link and the number of client-server pairs that are active simultaneously, this link may or may not be a bottleneck.

Traffic is generated using a simulated Web benchmark, which is designed to be representative of a "typical" Web transaction. It consists of four steps: the client sends a request for the HTML, the server responds with the HTML, the client then sends requests for each inline image, and finally the server sends back the images. In some experiments, background traffic in the form of bulk transfers is also introduced.

Figure 6.7 also summarizes the settings (including the default one) for the various simulation parameters.

We evaluated five different protocol configurations:

1. *httpseq*: HTTP/1.0 with sequential connections of the TCP NewReno flavor.
2. *httpconc*: HTTP/1.0 with concurrent connections of the TCP NewReno flavor. There are as many concurrent connections as inline images in the page being downloaded.




**Link parameters**

Bandwidth: 1.5 Mbps, 28.8 Kbps, 45 Mbps
Delay: 50 ms, 200 ms
Buffer size: 20 packets

**Web benchmark parameters**

HTML request size: 0.3 KB
HTML response size: 1 KB
Image request size: 0.3 KB
Image response size: 1-100 KB (10 KB by default)
# inline images: 1-10 (4 by default)
# simultaneous downloads: 1-30

**TCP parameters**

Max segment size (MSS): 1 KB
Max window size: 32 KB

**Figure 6.7 The simulation topology and parameter values. For parameters whose values are varied, the default setting is underlined.**

3. *phttp*: P-HTTP with a persistent connection of the TCP NewReno flavor.

4. *session*: TCP session

5. *session-noilr*: TCP session without integrated loss recovery (i.e., with only per-connection loss recovery as in standard TCP).

Of these, *httpconc* and *session* are of particular interest to us and are analyzed in the greatest detail. The former is representative of popular browsers such as Netscape Navigator that launch multiple concurrent connections with the hope of speeding up the Web transaction. As we shall see, this can often be highly counterproductive. The *httpseq* configuration is included to show that in spite of the drawbacks mentioned in Chapter 4, it could perform much better than *httpconc* in congested environments. The *phttp* configuration is included to see how close session is able to approach it in terms of performance, while at the same time avoiding the drawbacks of P-HTTP mentioned in Chapter 5. Many of these drawbacks (for example, the undesirable coupling of logically-separate data streams) are difficult to quantify and only manifest themselves in an actual implementation. They do not have an impact on the performance metrics listed below, so the graphs for *phttp* and *session* are very similar in all the cases. Finally, a comparison between *session* and *session-noilr* helps evaluate the benefits of integrated loss recovery.

We use several metrics to quantify performance. These include:

1. The *total download time*, averaged over several (up to 30) clients and also multiple runs.
2. *Fairness index* computed over the download time of each individual client and averaged over multiple runs.
3. The number of *packet losses* and the number of *retransmission timeouts*, both averaged over multiple runs.

Error bars represent 95% confidence intervals for the mean.

#### 6.5.2.1 Impact of the Number of Simultaneous Web Downloads

In this experiment, we evaluate the impact of the number of simultaneous downloads on performance. Each download is between a distinct client-server pair, so the $i^{th}$ download is between client $C_i$ and server $S_i$. All the downloads are initiated within a short, 100 ms interval. The bottleneck link bandwidth is 1.5 Mbps and delay 50 ms.

Figure 6.8(a) shows the average download time per client versus the number of simultaneous downloads for three protocol configurations — *httpseq*, *httpconc*, and *session*. As we would expect, the download time increases as the number of simultaneous downloads increases. However, the increase is the greatest for *httpconc* and the least for *session*. There is a speedup of up to 60% (i.e., 2.5X) in the download time with *session* compared to *httpconc*. The speedup over *httpseq* is smaller but still significant — up to 30%. The performance with *phttp* is quite similar to that with *session*, so we only show the curve for the latter.

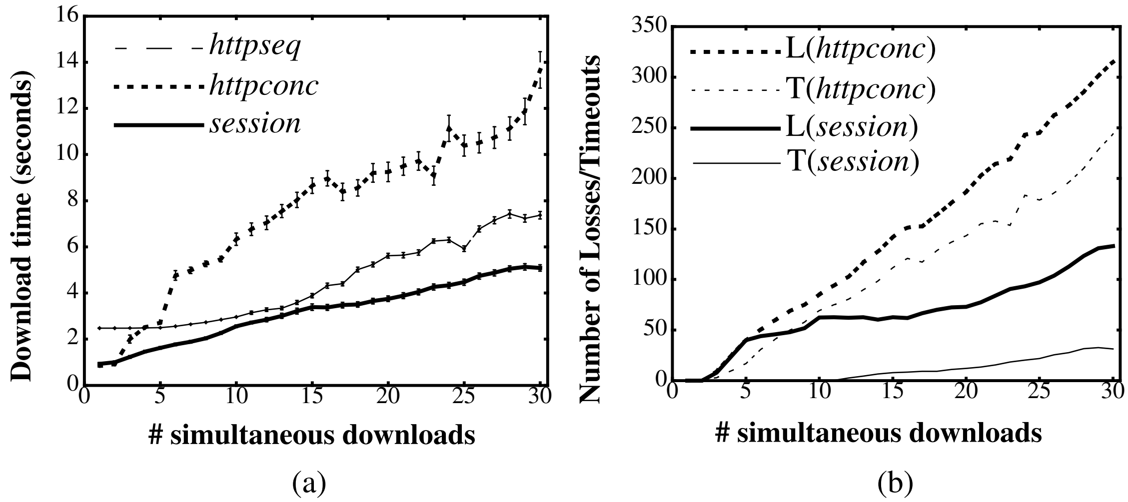
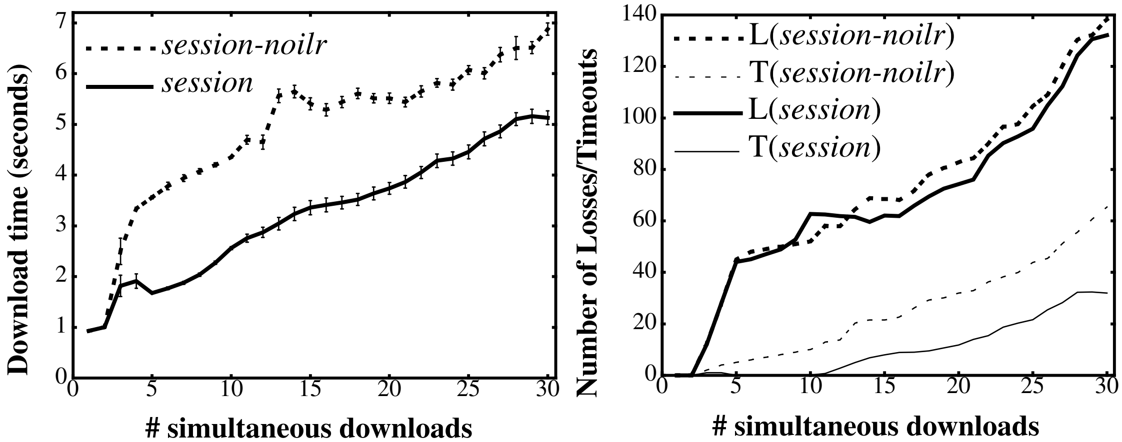



**Figure 6.8 Impact of the number of simultaneous downloads on (a) the average download time per client, and (b) the total number of packet losses (L) and retransmission timeout (T).**

Figure 6.8(b) goes into the reasons for the significant performance gains of *session* in more detail. It shows the total number of packet loss and retransmission timeout events for each of *httpconc* and *session*. Comparing the total number of losses, we observe that it increases much more sharply for *httpconc* than for *session*. Towards the right edge of the graph, it is over 2.3X larger for *httpconc* than for *session*. As discussed in Section 6.3.2 (and depicted in Figure 6.1), the independent connections (4 per download, in this case) in *httpconc* do not share congestion information with each other and push the network beyond the brink (thereby causing packet loss) much more frequently.

There is an even more dramatic difference when we compare the number of timeouts in the two cases. This is often more than 7X greater for *httpconc* than for *session*. Another way of viewing this is that whereas over 75% of packet losses result in a retransmission timeout with *httpconc*, under 25% do so with *session*. The short length of each connection (10 KB for each inline image) often forces it to wait for a timeout to recover from a packet loss. In contrast, integrated loss recovery enables data-driven loss recovery to succeed much more often in the case of *session*.




(a) (b)

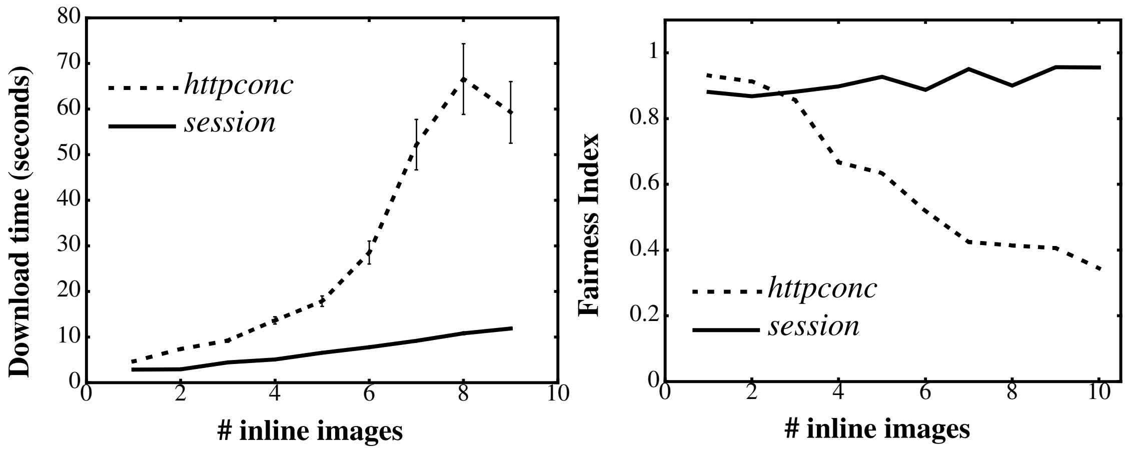
**Figure 6.9 Impact of integrated loss recovery on (a) the average download time per client, and (b) the total number of packet losses (L) and retransmission timeouts (T).**

Integrated loss recovery is also the reason why *session* performs better than *httpseq*. Although *session* suffers about 30% more losses than *httpseq* (on account of the latter operating with a very small effective window), it only experiences about half as many retransmission timeouts.

In summary, launching multiple connections concurrently can result in significant performance degradation. Part of this is because of increased packet losses. The rest is because of the predominance of timeouts. The latter also afflicts separate connections that are launched sequentially (*httpseq*). TCP session achieves significant performance gains compared to these because of integrated congestion control and integrated loss recovery. We analyze the latter in more detail next.

#### 6.5.2.2 Impact of Integrated Loss Recovery

To isolate the role of integrated loss recovery in the performance gains of *session*, we create a version of *session* minus integrated loss recovery. We call this *session-noilr*. Basically, *session-noilr* is identical to *session* except that it does not use later ack information for loss recovery. This is implemented by setting the *lateracks* count to zero for the purpose of loss recovery rules #2 and #3 in Figure 6.5.




(a) (b)

**Figure 6.10 Impact of the number of inline images per page on (a) the average download time per client, and (b) the fairness index computed over the download time for each individual client. The inline image size was set to 10 KB and the number of simultaneous downloads to 30.**

We conduct the same experiment as in Section 6.5.2.1. Figure 6.9(a) shows that the average download time degrades by amounts ranging from 35-100% (i.e., 1.35-2X) when integrated loss recovery is disabled. This is corroborated by Figure 6.9(b) which shows that although the total number of losses is about the same in both cases (because both employ the same integrated congestion control algorithm), the number of timeouts with *session* is only about half as much as with *session-noilr.*

Our results underscore the importance of integrated loss recovery. It is this that enables TCP session to perform data-driven loss recovery very effectively even when connections are short in length.

### 6.5.2.3 Impact of the Number of Inline Images

In this experiment, we investigate how the number of inline images in the page being downloaded impacts performance. We compare *httpconc* and *session*, both of which launch as many concurrent connections as there are inline images. Figure 6.10(a) plots the average download time per client as a function of the number of inline images. We observe that while the performance in the two

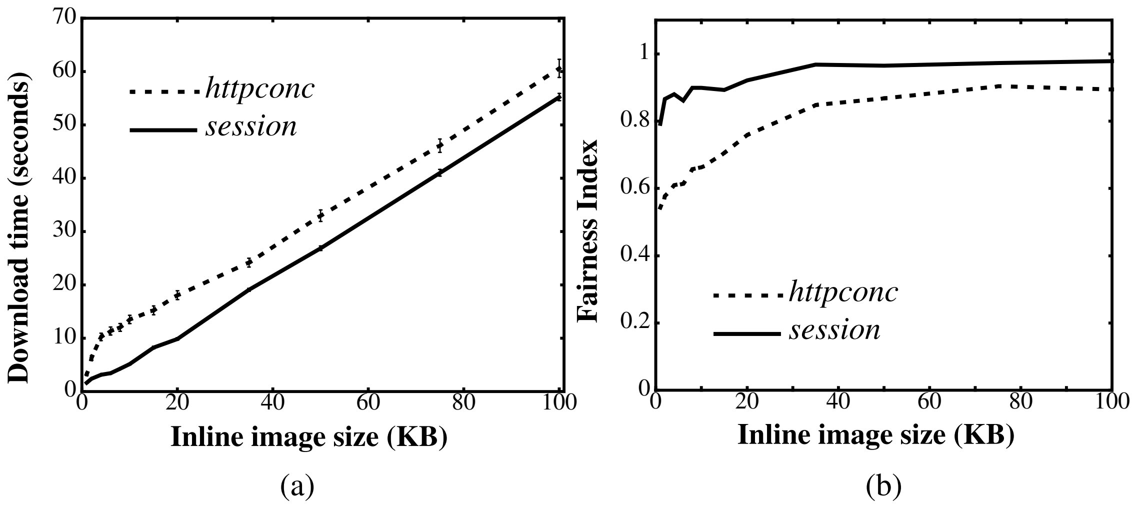



**Figure 6.11 Impact of inline image size on (a) the average download time per client, and (b) the fairness index computed over the download time for each individual client. The number of inline images per page is set to 4 and the number of simultaneous downloads to 30.**

cases is similar when there is only one inline image per page, there is a growing difference as the number of inline images increases, ending up with a difference of more than 6X. This trend arises because the benefit of integrated congestion control and loss recovery is greater when the integration happens over a larger number of (concurrent) connections.

Figure 6.10(b) plots the fairness index in the two cases. This index is computed over the download time for each client, and is a measure of how equitably the clients are treated. The figure shows the index starts off with a high value (around 0.9) in both cases, indicating that all clients receive similar performance. However, as the number of inline images increases, the fairness index for *httpconc* plunges, ending up at around 0.4 when there are 10 inline images per page. The reason for this is that as the number of inline images increases, the amount of packet loss in the case of *httpconc* tends to increase. (Part of this increase is because of the large initial burst mentioned in Section 6.3.2). In turn, this leads to an increase in the occurrence of timeouts. Since timeouts have a very significant impact on performance, there is a great disparity in the download time for clients whose connections experience a large number of timeouts, compared to those that experience fewer timeouts or even none. This causes the decline in the fairness index.

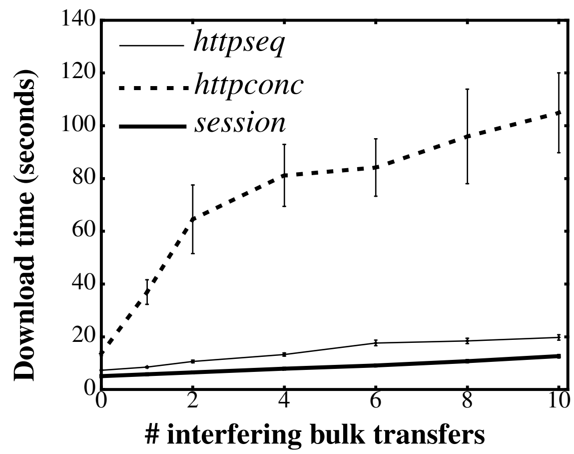
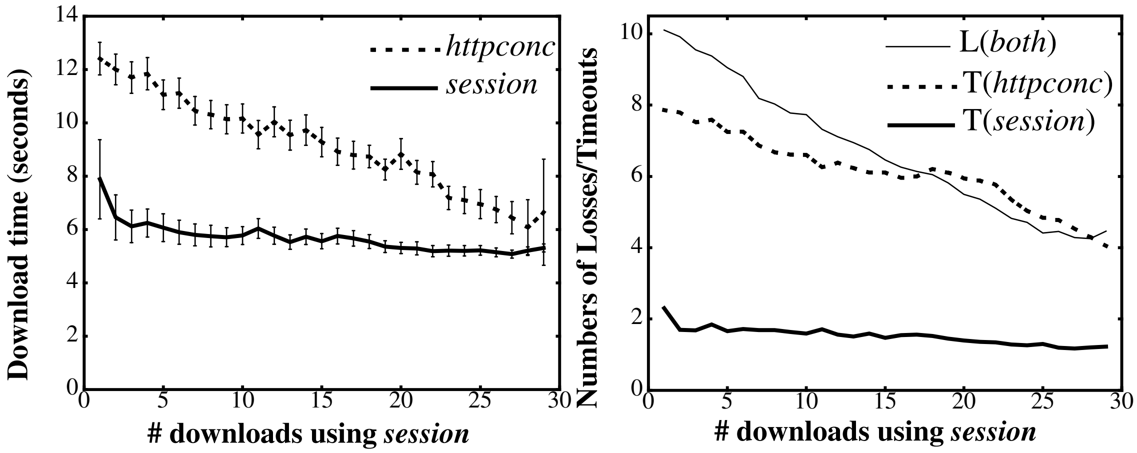



**Figure 6.12 Impact of interfering bulk transfers on the average download time per client. The number of inline images per page is set to 4 and the image size to 10 KB.**

On the other hand, *session* is able to avoid such degradation in the level of fairness by reducing the occurrence of timeouts.

### 6.5.2.4 Impact of the Inline Image Size

In this experiment, we study the impact of the size of each inline image on performance. Recall that the size of an inline image determines the length of the connection used to transfer the image. From Figure 6.11(a), we see that there is a steady increase in the average download time as the inline image size increases. This is as we would expect. Furthermore, the benefit of *session* over *httpconc* diminishes, in percentage terms, from over 50% (i.e., over a factor of 2) for 1 KB images to under 10% for 100 KB images. The reason for this decrease in relative benefit is that as the inline image size grows, each individual connection gets longer. So data-driven loss recovery becomes increasingly more feasible, even in the absence of integrated loss recovery. Given this, the trend in the fairness index (Figure 6.11(b)) is also as we would expect. As the inline image size grows and timeouts become less frequent, the degree of fairness also improves, more so for *httpconc*.




(a) (b)

**Figure 6.13 Impact of a heterogeneous traffic mix, where some downloads use the *httpconc* configuration and others *session*, on (a) the average download time per client, and (b) the average number of packet losses (combined for both configurations) and timeouts (separate for each configuration) per download. The number of downloads of each type is varied, while holding the total constant at 30. The number of inline images per page is set to 4 and the image size to 10 KB.**

### 6.5.2.5 Impact of Interfering Bulk Transfer Traffic

So far in our experiments, we have only considered Web-like traffic between clients and servers. We now introduce cross-traffic in the form of bulk transfers, in addition. These share the bottleneck link with the Web downloads. The number of bulk transfer connections is varied from 0 through 10. Figure 6.12 shows the impact of the bulk transfers on the Web download time. The most striking point in this graph is the wide disparity in the performance of *httpconc* and *session*. The download time with *httpconc* grows to be about 10X that with *session*. The reason for this is that the bulk transfers cause congestion in the network. When *httpconc* starts of with a large total initial window, there is heavy packet loss, leading to a large number of timeouts.

There is a smaller, but still significant, improvement compared to *httpseq*. *Session* cuts down the average download time by up to a factor of 2. This is because of its improved ability to do data-driven loss recovery.

#### 6.5.2.6 Impact of Heterogeneous Traffic Mix

In the same spirit as the previous experiment, we now evaluate *session* in an environment where there is a mixture of Web downloads that use *session* and those that use *httpconc*. The number of downloads using session is varied from 1 through 29 while the total number of simultaneous downloads is kept constant at 30 (so by implication, the rest use *httpconc*).

Figure 6.13(a) shows how the performance of the two categories of downloads varies with the traffic mix. We observe that both categories benefit as the mix shifts towards *session*. The benefit is greater for *httpconc*. The reasons for these trends is that when the traffic mix is dominated by *httpconc*, the network is more congested. The consequent packet losses adversely impact *httpconc* more than *session* because the former is more vulnerable to timeouts. As the share of *session* in the mix increases, the level of congestion in the network subsides, so *httpconc* suffers fewer timeouts and benefits substantially. *Session* also benefits, although not quite as much because timeouts were not a significant problem for it to begin with.

The trend in the frequency of timeouts is evident from Figure 6.13(b), which shows the average number of packet losses and timeouts per download. The frequency of timeouts with *httpconc* drops from being about 4.5X worse than *session* when the mix is dominated by *httpconc* to being about 3X worse when the mix is dominated by *session*. Note that in some cases, the average number of timeouts per download for *httpconc* actually exceeds the average number of losses per download. This is not an anomaly because the latter is computed by averaging over all downloads, including those that use *session*.

The results of this experiment and the previous one highlight the resilience of *session* in environments with a heterogeneous mix of traffic. It is advantageous to use *session* even when the majority of hosts in the network do not. This is important from the viewpoint of deploying it incrementally. The increasing use of *session* reduces the level of congestion in the network, which also benefits hosts that do not use *session*.

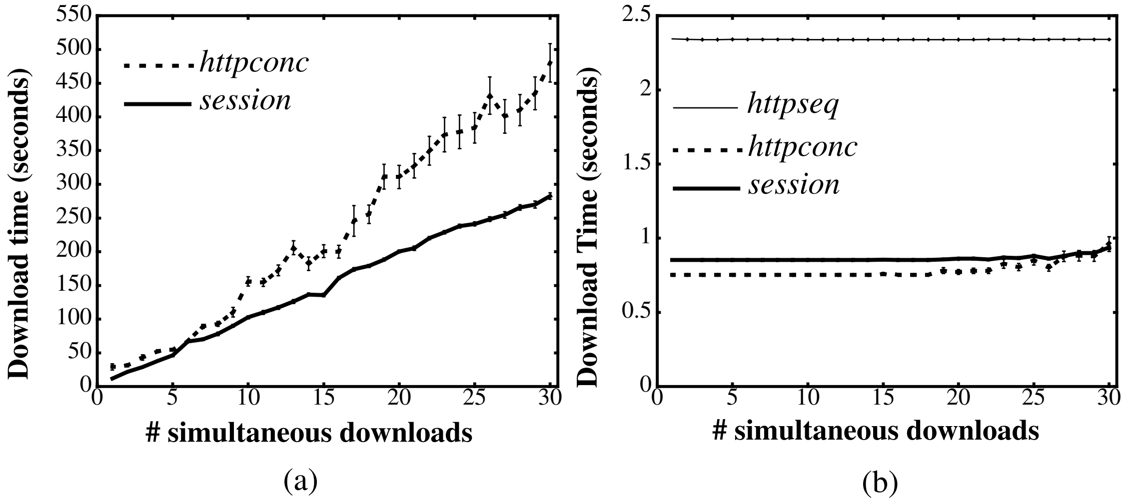



**Figure 6.14 Impact of link bandwidth, (a) 28.8 Kbps and (b) 45 Mbps, on the average download time per client. The number of inline images per page is set to 4 and the image size to 10 KB.**

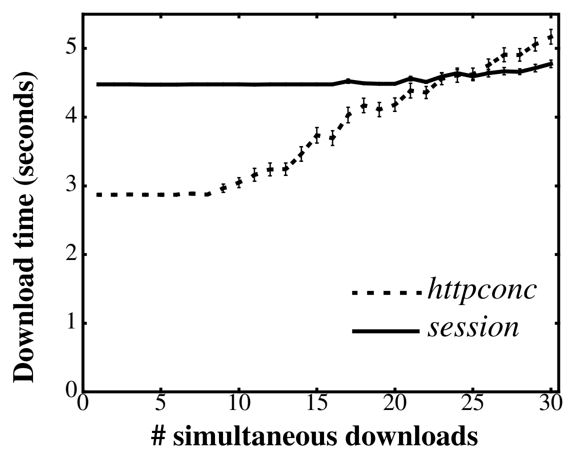



**Figure 6.15 The impact of a large RTT on the average download time per client. The link bandwidth is set to 45 Mbps and the delay to 200 ms (so that the RTT is about 400 ms). The number of inline images per page is set to 10 and the image size to 10 KB.**

### 6.5.2.7 Impact of the Link Speed and Delay

In all our experiments so far, the bottleneck link bandwidth has been held fixed at 1.5 Mbps, and its delay at 50 ms. We now vary these and repeat the experiment from Section 6.5.2.1, to study their impact on performance.

Figure 6.14(a) show the download time when the bottleneck link bandwidth is 28.8 Kbps. This extremely limited link bandwidth causes heavy congestion and packet loss. The consequent timeouts cause a sharp increase in download time as the load (i.e., number of simultaneous downloads) increases. The increase is greater for *httpconc* than for *session* because of the former's greater susceptibility to timeouts. Although the difference in performance in absolute terms is quite large (often over 100 seconds), the difference in relative terms is smaller than when the bottleneck link bandwidth was larger (Section 6.5.2.1). The reason is that with a link bandwidth of 28.8 Kbps, the transmission time for each packet is quite significant compared to the retransmission timeout (RTO) value. For instance, the transmission time for a 1KB packet over the 28.8 Kbps link is about 280 ms. In contrast, the transmission time over a 1.5 Mbps link is only about 5 ms. However, the difference in RTO in the two cases is not quite as much because it is computed at a rather coarse granularity (typically 500 ms). Consequently, the cost of a timeout (i.e., the RTO) is less significant in relative terms when other costs (e.g., transmission time) are large.

Figure 6.14(b) shows the situation at the other end of the spectrum where the link bandwidth is large (45 Mbps). We observe a markedly different behavior. The performance of *session* is slightly worse (about 13%) than that of *httpconc*. However, both of these perform much better (about 3X) than *httpseq*. These trends arise because there is only mild congestion given the high link bandwidth. Consequently, packet losses and timeouts are infrequent and only have a small impact on performance. The main issue becomes growing the congestion window to a large enough size to utilize the available network capacity. In this context, *session* and *httpconc* are in an advantageous position, the latter more so because the ensemble of concurrent connections starts of with a larger total initial window. As explained in Chapter 4, *httpseq* suffers, both because of repeated invocations of slow start and because of sequential operation.

These effects are even more pronounced in Figure 6.15, which shows the case of a 45 Mbps link with an RTT of 400 ms (akin to a geostationary satellite link). The number of inline images is set to 10 (instead of 4 in Figure 6.14). This means that *httpconc* launches 10 concurrent connections to retrieve the inline images. The consequent large initial window and faster window growth enable it to perform up to 35% better than session when the load is low. However, as the load increases, the concurrent connections launched by a large number of downloads overwhelm the network, causing more frequent packet loss. Consequently, the performance of *httpconc* begins to degrade, ending up worse than *session*.

The download time for *httpseq* under conditions of low load is around 21 seconds (not show in the figure), which is over 7X worse than for httpconc and a little under 5X worse than for session.

Thus, while *session* still maintains its performance advantage when the network bandwidth is very constrained, it performs worse than *httpconc* when capacity is plentiful and delay is large.

### 6.5.3 Summary of Results

Here is a summary of our key results:

1. *Session* outperforms *httpconc* under conditions of high load. In terms of download time, there is a speedup by up to a factor of 10. There are two reasons for this:

   - Integrated congestion control cuts down the packet loss rate significantly.
   - Integrated loss recovery cuts down the frequency of timeouts drastically.

   Improving performance when the network is heavily loaded is especially important because it is under these conditions that users perceive long delays, which leads to complaints.

2. The use of *session* is often advantageous even in a mixed environment where not all traffic uses *session*. This is significant because it facilitates incremental deployment. Furthermore, as the use of *session* grows, even other traffic in the network benefits because of the reduced level of congestion.

3. *Session* also performs better than *httpseq* though not quite as much. The performance difference is the greatest (up to 2X) when there is a significant amount of interfering traffic other than the Web traffic of interest. The performance benefits arise due to the ability of integrated congestion control to cut down the occurrence of timeouts.

4. When the bandwidth-delay product is large, *session* performs worse than *httpconc*. This is because, unlike the latter, it always starts off with a small initial session window size of 1 segment, so it is less able to fully utilize the available capacity. *Httpseq* performs much worse than both *session* and *httpconc*. *Phttp* performs similar to *session*.

5. Connection scheduling enables *session* to provide a consistent share of the bandwidth to each concurrent connection. In contrast, with *httpconc* the share obtained by each connection can be quite unpredictable. By comparison, neither *httpseq* nor *phttp* permits concurrent connections.

6. In spite of allowing applications to launch short and concurrent connections, *session* achieves as small a download time as *phttp*. Integrated congestion control and loss recovery blunt the adverse impact that short connections may have otherwise had. Furthermore, session eliminates the disadvantages of phttp mentioned in Chapter 5 — it enables having concurrent data streams without any unnecessary coupling between them, it is transparent to applications, and it confines changes to the sender side (typically, Web servers).

## 6.6 Limitations of TCP session

In spite of its advantages, TCP session has some limitations:

1. As mentioned above, the small initial session window size could lead to poor utilization of the available network capacity, given that total transfer size for a Web page may only be of the order of 20-30 KB. This is especially so when the network capacity is plentiful, i.e., the network bandwidth is large. Clearly, this is an undesirable situation.

   In Chapter 8, we present an orthogonal technique, *TCP fast start*, to alleviate this problem by reusing information about the network learned in the recent past.

2. It may be expensive to multiplex several data streams within a TCP session at a very fine granularity because of the overhead of the TCP/IP header (usually 40 bytes or more). If this is really an issue, header compression techniques similar to those used for low-speed dialup lines [52] could be used to reduce the overhead.

3. TCP session does not share any information across hosts. There are situation where this may be advantageous. For instance, two hosts on a LAN could benefit by sharing information about the network path to a common server. The SPAND system [97] points to a possible way of effecting such sharing.

## 6.7 Summary

In this chapter, we have presented the design and evaluation of a new technique, TCP session, which enables applications to open as many concurrent connections as they desire, without the attendant performance penalties. Connections share information rather than compete with each other. The three main components of TCP session are:

1. Integrated congestion control.

2. Connection scheduling.

3. Integrated loss recovery.

Our experimental results show that TCP session performs much better than HTTP with concurrent connections, and about the same as P-HTTP, while at the same time avoiding many of the drawbacks of the latter.

Since TCP session is transparent to applications (by default), it could be used in conjunction with an application-level multiplexing technique such as P-HTTP. If an application chooses to use P-HTTP, TCP session will essentially make no difference to performance. However, if another application on the same host chooses to launch several concurrent connections, TCP session would guard against performance degradation.

We note that although we have presented TCP session in the context of the Web, there is nothing in it that ties it to the Web. The technique would be useful in any situation where hosts exchange multiple concurrent data streams. An example is a shared whiteboard that uses TCP for communication between a pair of hosts [93]. Only pieces of data for which ordered delivery is desirable are mapped onto the same connection. Those corresponding to independent activities (for instance, writing on one end of the whiteboard while loading up an image at the other end) are mapped onto separate connections. TCP session would then be useful in ensuring that these connections do not compete with each other.

In the next chapter, we turn to some advanced issues pertaining to TCP session. Chief among these is exposing connection scheduling to the application. This enables the application to allocate a greater share of the bandwidth to certain connections based on application-level information, and even vary this allocation dynamically.

# Chapter 7

# TCP Session: Advanced Issues

In this chapter, we build on the TCP session technique we developed in Chapter 6 with a discussion of some advanced issues. First, we discuss the issue of connection scheduling in Section 7.1. Unlike in Chapter 6, where all connections in a session were assigned equal weight, we discuss instances where assigning unequal weights may be useful. Then, in Section 7.2, we discuss modifications to the integrated congestion control algorithm (Section 6.3.2) that may be necessary in certain contexts, for instance, when clients access the Web via proxy hosts. Finally, in Section 7.3, we evaluate the potential for false retransmissions due to integrated loss recovery (as mentioned in Section 6.3.4). Although our experimental results show no evidence of such a problem, we still discuss a possible solution if it ever were to be a problem.

## 7.1 Connection Scheduling

As discussed in Section 6.3.3, TCP session enables the sender to schedule connections within a session explicitly rather let each connection's share of the session bandwidth be determined by the vagaries of the network. To keep the operation of TCP session transparent to applications, we use round-robin scheduling, by default. However, as we discuss next, it may be advantageous to expose connection scheduling to applications (of course, sacrificing transparency in the process).

Exposing connection scheduling[1] to applications is motivated by arguments similar to those advocating the *application-level framing* (ALF) [21] principle for network protocol design. Basically, applications have knowledge of the semantics of the data being transferred that layers (such as TCP) lower down in the protocol stack do not. The data carried in certain connections within a TCP session may be of greater importance from an application's viewpoint than that in others. And the relative importance the individual components of the data (and hence that of the corresponding connections) may vary dynamically.

We discuss a couple of examples to illustrate these ideas. Consider a client downloading a Web page over a slow dialup line. Assume that page contains two images, one relevant to its content and the other a large advertisement banner. The client may request both images simultaneously. However, from the viewpoint of the server administrator, it may be desirable that the more important image be given higher priority, lest the user loses interest in the page (because the download of advertisement banner delays the other image significantly) and aborts the download. Meta-data information associated with the images can be used by the server *application* to assign the appropriate scheduling priorities to the two connections within their session.

As another example, consider a client downloading a large Web page with only a subset of the inline images lying in the visible region of the client's window. In such a case, it would be desirable, from the viewpoint of minimizing *user-perceived* latency, to speed up the download of this subset of the images even at the expense of other images that are currently outside the visible region. However, if the user were to scroll down the page, the some of the previously hidden images would become visible and vice versa. This would call for a change in scheduling priorities. However, unlike in the previous case, the scheduling priorities of the individual connections is determined by the client (the receiver), who needs to be convey it to the server (the sender).

Next, we discuss our connection scheduling algorithm to support such applications.

## 7.1.1 Hierarchical Round-Robin Scheduler

We use a *hierarchical round-robin scheduler* (HRR) [59] to schedule connections within a session. HRR allows assigning different levels of scheduling priority to individual connections. Just as

1. Note that we are only concerned with scheduling at the sender, no at routers within the network.

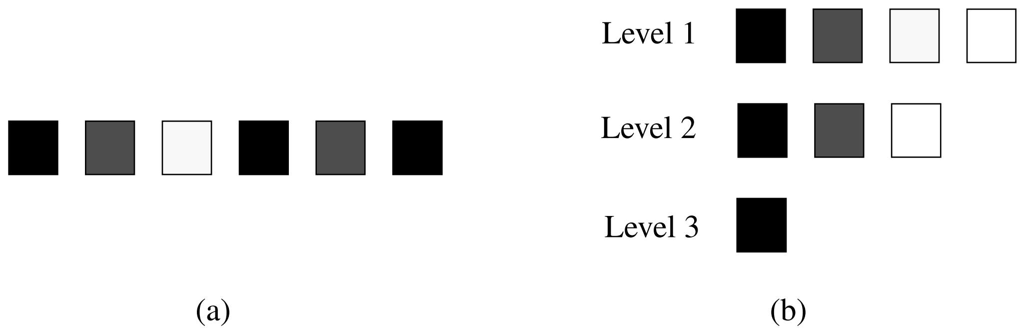



**Figure 7.1 (a) An illustration of weighted hierarchical round-robin scheduling with 3 connections, the first with a weight of 1/2, the second with 1/3 and the third with 1/6. The interleaving pattern shown keeps getting repeated. (b) A depiction of the schedule as a hierarchy of round-robin schedules. All the slots at a particular level has the same weight. A blank slot implies moving to the next level.**

plain round-robin scheduling does for the un-weighted case, HRR ensures that the transmission slots allocated to each connection are distributed as uniformly as possible given their relative weights. This is illustrated in Figure 7.1(a). In the context of HRR, "hierarchical" refers to the logical depiction of the schedule as a hierarchy of round-robin schedules, as in Figure 7.1(b).

By default, the weight of each connection is initialized to 1, so HRR degenerates to round-robin. We provide a simple interface for applications to control scheduling. It consists of two operations:

1. *setwt*(*conn*, *n*): set the weight for the connection *conn* to *n*.

2. *resetwt*(*session*): reset the weight for all connections in *session* to 1.

In practice, the argument to *resetwt* just specifies any one of the connections in the desired session. This is sufficient because the TCB of each connection contains a pointer to the corresponding SCB. This minimizes the change needed to the existing socket interface (since the session is not exposed to the application level explicitly) without sacrificing functionality.

This interface, though simple, is sufficiently powerful for use in the two scenarios discussed in Section 7.1. To speed up the download of a particular image, *setwt* is called, with a suitably large *n*, on the corresponding connection. If at a later point another connection is to be speeded up, then *setwt* is invoked with weight 1 on the old connection and weight $n$ on the new one. The *resetwt* operation is useful in case the weights on a large number of connections need to be set to 1.

To enable the client to convey scheduling information to the server (as in the second scenario discussed in Section 7.1), the HTTP protocol may be extended with new methods corresponding to *setwt* and *resetwt*. If a separate connection is used for each component of a Web page (as we advocate in the TCP session model), the client could implicitly specify the component whose download is to be sped up or slowed down by sending its *setwt* or *resetwt* request on the corresponding connection.

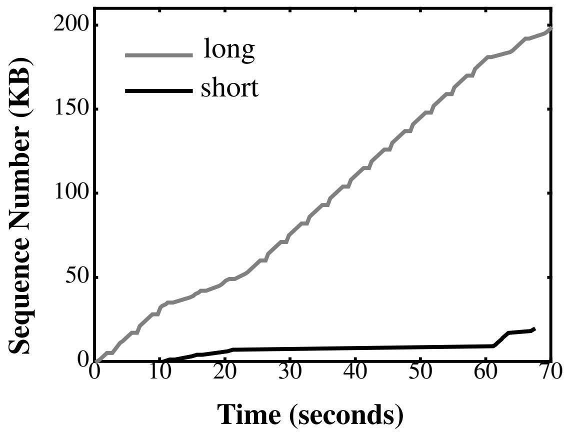
### 7.1.2 Comparison With Alternatives

We compare connection scheduling within a TCP session with two alternatives: (a) scheduling independent TCP connections, and (b) scheduling logical data streams within a single (persistent) TCP connection.

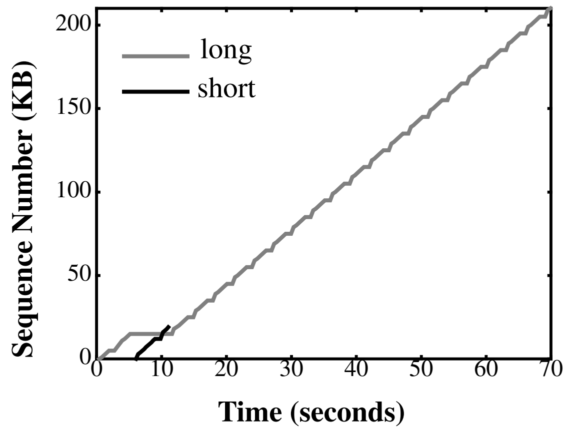
#### 7.1.2.1 Scheduling Independent TCP Connections

We first consider an *ns* simulation of a client downloading a Web page with two embedded objects, a large one (250 KB) and a small one (20 KB). The network bandwidth and delay are set to 28.8 Kbps and 50 ms, respectively. At first, the client only requests the large object. But at a later point (about 4 seconds into the download), its priorities get altered (for example, due to an action, such as scrolling, by the human user), and the client now wishes to download the small object as quickly as possible.

Figure 7.2 illustrates the progress of the connections in two cases: (a) with independent TCP connections for each object, and (b) separate TCP connections, within the same TCP session, for each object. In the latter case, the short connection is assigned a scheduling weight of 10. We note from the figure that in the case of independent connections (Figure 7.2(a)), the new (short) connection is forced to compete with the existing (long) connection. As a result, it performs poorly, resulting in a large download time of 64.32 seconds for the small object. In contrast, with TCP session (Figure 7.2(b)), the new connection is right away able to use a large share (on account of its weight being 10) of the session window build up by the original connection. As a result, the download time for the second object is much smaller, only 7.05 seconds.




(a)

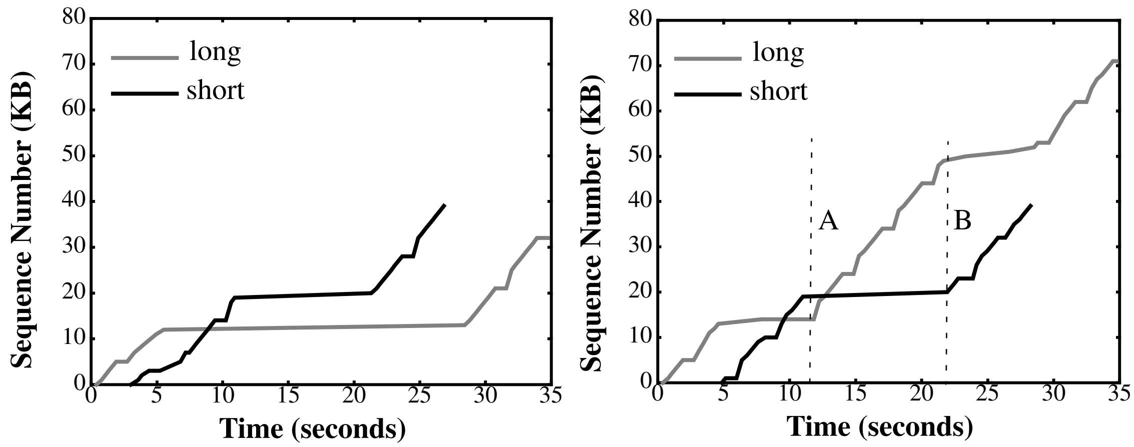



(b)

**Figure 7.2 Sequence number trace of the long and short connections over a 28.8 Kbps link. (a) corresponds to independent connections while (b) corresponds to TCP session. For ease of comparison, we use the same x-scale for both graphs.**

One possible way of speeding up the short connection in the case of independent connections is to explicitly slow down or halt the long connection. This may be accomplished either by having the receiver read in data slowly (thereby invoking the TCP flow control mechanism) or by having the sender send data slowly on the long connection. In either case, explicitly curtailing the long connection has a downside, as illustrated in Figure 7.3(a). We assume the short connection to be 40 KB long in this case. However, halfway through the transfer, the application sending data on the short connection is assumed to stall for several seconds. In practice, this could happen for several reasons, including disk contention or a temporary slowdown in the case of dynamically generated data. As a result, the short connection does not make any progress, but neither does the long connection even though it may have data ready to send. In general, such stalls may happen at unpredictable times, so it would be difficult to alter the level of flow control on the long connection at short notice. Another drawback in general is that when the long connection is slowed down, the short one does not immediately start using the bandwidth that is freed up, because there is no explicit communication between the independent connections. It discovers that more bandwidth is available only via the usual process of growing its window gradually.

In contrast, connection scheduling within the TCP session (Figure 7.3(b)) ensures that the long connection takes over immediately when the short connection stalls (time A), in spite of the weight




(a) (b)

**Figure 7.3 Sequence number trace of the long and short connections over a 28.8 Kbps link. (a) corresponds to independent connection while (b) corresponds to TCP session. The short connection stalls for several seconds after transferring 20 KB of data.**

of the latter being much higher. Likewise, the short connection reclaims its share immediately when it has data to send again (time B).

### 7.1.2.2 Scheduling Logical Data Streams within a Single TCP Connection

We now consider scheduling data streams within a single TCP connection, as might happen with P-HTTP. As explained in Section 5.7, both P-HTTP and its adaptation in HTTP/1.1 force the logical data streams to be ordered sequentially, i.e., a new one begins only after the previous one has terminated. As such, the short transfer would have to wait for the long one to complete first.

An alternative is to use a fine-grained multiplexing algorithm such as WebMUX [36]. However, we still run up against the fundamental problem mentioned in Section 5.7, that of coupling between the logical data streams introduced by ordered byte-stream service model of TCP. In the scenario discussed above, this implies the data belonging to the short transfer will be held up behind data of the long transfer that has already been sent by the application. The latter includes not only the data in transit in the network but also that queued up in the network socket buffer at the sender. This can be quite large because the socket buffer size is typically provisioned to handle a wide range of bandwidth-delay product settings. In our experiment, it is set to 32 KB, so the completion time for the short transfer (20 KB in length) is 15.90 seconds, of which about 9 seconds is spent waiting for the data already in the socket buffer to be cleared out. In contrast, the completion time with TCP session is only 7.05 seconds (Figure 7.2(b)).

### 7.1.3 Summary

In summary, connection scheduling enables the sender to explicitly control how a session's bandwidth is apportioned to each constituent connection. The scheduling policy can be controlled at the application level. Connection scheduling within a TCP session performs better and is more flexible than both scheduling independent TCP connections and scheduling logical data streams within a single TCP connection.

## 7.2 Adapting Integrated Congestion Control for Proxy Hosts

The second issue is the need to adapt the integrated congestion control algorithm in certain situations, such as clients accessing the Web via a proxy host or several users running Web clients on a single time-shared system. For ease of exposition, we confine ourselves to the proxy host case. The problem is that, with integrated congestion control, the server would place the proxy host, which may be serving a large number of clients, at par with a client host that connects to it directly, i.e., without going through a proxy. As a result, each of the clients behind the proxy may receive a much smaller share of the network bandwidth than the client that connects to the server directly.

This is illustrated in Figure 7.4(a), where client A connects directly to the server while clients B and C connect via a proxy. Initially, only clients A and B are active, and each receives a roughly equal share of the bandwidth. However, when client C also starts up, each of B and C receives only about half as much bandwidth as A.

To address this problem, we modify the integrated congestion control algorithm in such a way as to mimic the case where each client connects directly to the server, not via a proxy. The solution involves two steps. First, the server determines which connections are on behalf of the same client or user. It then applies the integrated congestion control algorithm on groups of such connections.

To facilitate the first step, the proxy application could explicitly inform the server application about the origin of each of its connections. Alternatively, the server application could deduce this information by itself, for instance, using *HTTP cookie* information [62].

As the next step, the TCP session implementation at the server groups together all connections on behalf of a client or a user into a *sub-session*. For each sub-session, it maintains an estimate of the amount of data that is outstanding in the network (*subsession_ownd*). The sub-session's share of the congestion window is then computed as *subsession_cwnd* = (*subsession_ownd*/*session_ownd*)**session_cwnd*. This quantity is then used to determine how *session_cwnd* is grown or shrunk:

1. When a new ack is received during slow start, *session_cwnd* is incremented by one segment, just as with the original algorithm (Section 6.3.2).

2. When a new ack is received during the congestion avoidance phase, *session_cwnd* is incremeted by a 1/*subsession_cwnd* fraction of a segment (i.e., *maxseg*/*subsession_cwnd* bytes, where *maxseg* is the maximum segment size). The idea is to increment *session_cwnd* by one segment per RTT per sub-session, just as would happen if each sub-session were a separate session. In contrast, the original algorithm would only increment *session_cwnd* by one segment per RTT for the entire session.

3. When the sender detects and recovers from a loss via data driven means, *session_cwnd* is decremented by *subsession_cwnd*/2[2]. So, in general, the decrease would be less than the halving done by the original algorithm. However, the cutback in window size could happen multiple times within a round trip, (at most) once for each sub-session. In contrast, with the original algorithm, it could happen at most once per RTT for the entire session.

4. If the sender experiences a retransmission timeout, *session_cwnd* is reset to one segment. Since a timeout is usually indicative of severe congestion, we retain the conservative response of the original algorithm.

As a result of this modified algorithm, the unfairness observed in Figure 7.4(a) is absent in Figure 7.4(b). Each of clients A, B and C receives a similar share of the bandwidth, as indicated by the roughly equal slopes of the three sequence number plots.

Note that the modified integrated congestion control algorithm does not alter two of the fundamental advantages of TCP session. First, connections within the same session share information grace-

2. This is appropriate because *subsession_cwnd* is tied to the amount of outstanding data, and hence the load, that the sub-session places on the network.

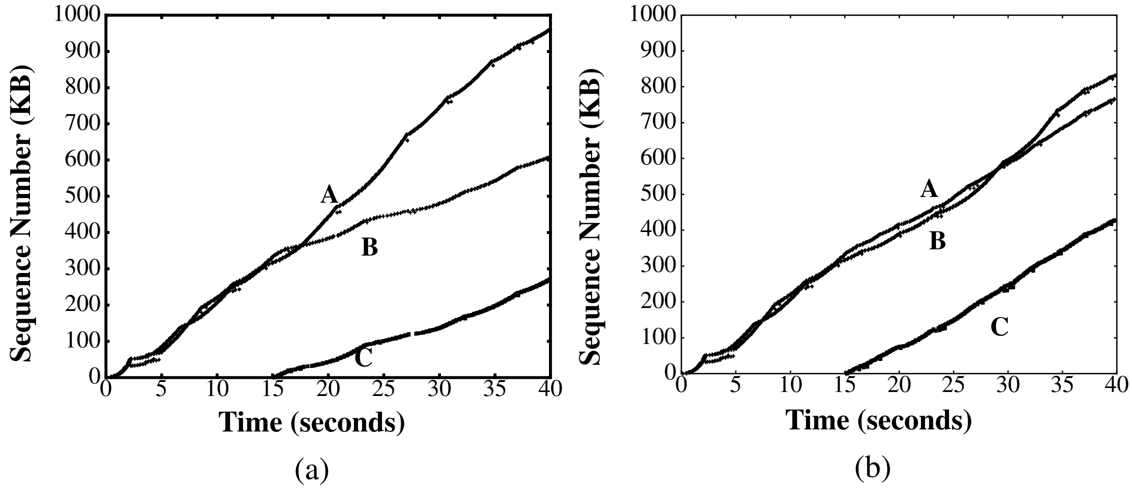



**Figure 7.4 The progress of connections between a server and three clients, A, B, and C. A connects directly to the server while B and C connect via a proxy host. (a) corresponds to the original integrated congestion control algorithm from Section 6.3.2, and (b) to the modified algorithm presented in Section 7.2.**

fully rather than compete with each other. This is so even among connections belonging to different sub-sessions. Among other things, this means that any cutback in the window size is shared by all connections within the session, so the performance of each is more consistent. Second, the integrated loss recovery procedure is unaffected by these modifications.

Although we have defined a sub-session as corresponding to one client or user, it is certainly possible to define it differently. Of course, the server would need some way knowing how to group connections into sub-sessions. It is also possible to build a hierarchy of sub-sessions and use a scheme such as Class-Based Queuing (CBQ) [34] to schedule among them. But this may not be worthwhile in practice because of the complexity involved.

## 7.3 Potential For False Retransmissions

Finally, we evaluate the potential for *false retransmissions* (i.e., unnecessary retransmissions) that integrated loss recovery may introduce. As explained in Section 6.3.4, the loss of acks on a connection could cause the sender to receive a sufficient number of later acks (on other connections) to trigger unnecessary retransmission of the corresponding packet(s). While even standard TCP is vulnerable false retransmissions when acks are loss, the use of later acks adds a new dimension to the problem.

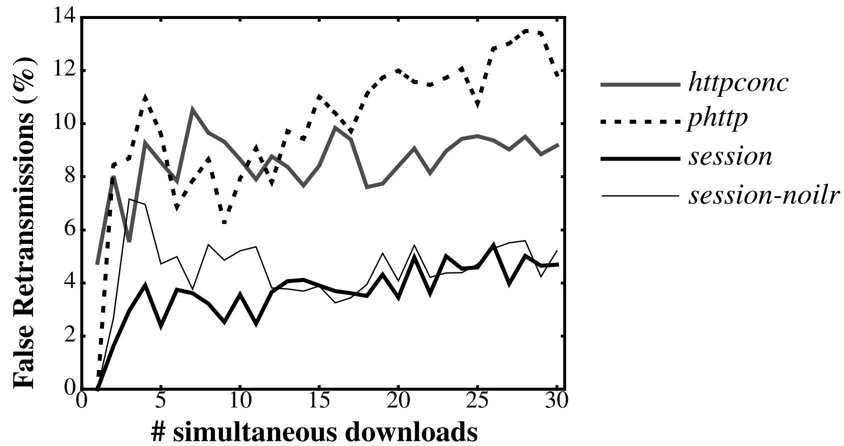



**Figure 7.5 The fraction of all retransmissions that are false (expressed as a percentage). We do not show error bars for the sake of clarity.**

We repeat the experiment from Section 6.5.2.1 in which a variable number of clients download a Web page simultaneously over a 1.5 Mbps/50 ms bottleneck link. Each Web page includes four 10 KB inline images. The one difference in this experiment is that heavy cross-traffic (in the form of ten bulk transfer connections) is introduced in the direction from the clients to the servers (i.e., in the direction in which acks flow). This often causes acks to be dropped in the network.

In Figure 7.5, we plot the fraction of all retransmissions that are false for four different protocol configurations — *httpconc*, *phttp*, *session* and *session-noilr*. Recall that *session-noilr* corresponds to TCP session minus integrated loss recovery. As such, it is not exposed to the additional risk of false retransmissions as *session* is. In spite of this, as Figure 7.5 shows, the likelihood of a retransmission being false is about the same (4-5%) with both configurations. This shows that at least in experiment, integrated loss recovery does not aggravate the problem of false retransmissions in spite of the frequent loss of acks. We believe this is because ack losses tend to be spread across connections making the pathological scenario outlined in Section 6.3.4 less likely.

However, it may still be desirable to have an explicit mechanism to guard against false retransmissions. One possibility is to introduce a new TCP option to enable acks on one connection to carry acknowledgement information for other connections in the same session as well. The option would specify the source and destination port numbers of the other connection and its cumulative ack (i.e., a total of 8 bytes), so a single ack could carry acknowledgement information for up to three other connections[3]. Since the sender would have a multiplicity of sources for acknowledgement information, false retransmissions would be less likely. However, the downside is that this requires support at the receiver (client) as well, whereas the operation of the rest of TCP session is confined to the sender.

It is interesting to note in Figure 7.5 that both *httpconc* and *phttp* are significantly more likely to suffer false retransmissions that session. With *httpconc*, this happens because of the greater dependence on retransmission timeouts as a means of loss recovery. Because of the limited information provided by a cumulative ack, a connection is susceptible to false retransmissions during the slow start phase following a timeout. With *phttp*, the frequency of timeouts is about the same as with *session*. However, it tends to suffer a greater number of false retransmissions because the cumulative ack on the single connection provides less information than that on each of a set of concurrent connections, as is the case with *session*.

## 7.4 Summary

In summary, we have discussed several advanced issues pertaining to TCP session. First, we discussed the flexibility provided by connection scheduling within a session. Then we described a modified version of integrated congestion control for use with in the context of Web proxies. Finally, we showed that integrated loss recovery does not make false retransmissions more likely even when acks tend to get lost.

With this we have concluded our discussion of TCP session. In the next chapter, we develop *TCP fast start*, a technique to address the second challenge from Chapter 1, namely enabling short transfers to utilize the network bandwidth effectively.

3. When SACK is used, an ack would carry this new option only if there is space left after the SACK blocks, in any, have been included.

# Chapter 8

# TCP Fast Start

Thus far we have developed two different solutions — P-HTTP and TCP session — to address the challenges of Web data transport. The *shared state* used by these techniques offers significant performance benefits compared to independent TCP connections. Furthermore, TCP session offers several advantages over P-HTTP in terms of flexibility. However, as we noted in Section 6.6, both TCP session and P-HTTP have suboptimal performance when the network has a large bandwidth-delay product.

In this chapter, we develop a new solution, *TCP fast start*, which addresses this problem. The basic idea is to use *persistent state* to enable even short transfers to fully utilize the available network capacity. TCP fast start is orthogonal to both P-HTTP and TCP session, and hence can be used in conjunction with either. The rest of this chapter is organized as follows. First, we motivate the problem that we are solving and point out the specific challenges involved in doing so. Then in Section 8.2 we present the design of TCP fast start in detail. We describe our BSD/OS implementation of fast start in Section 8.3. In Section 8.4, we detailed performance results obtained using our implementation and using simulation. We discuss various issues pertaining to fast start in Section 8.5, and present a summary in Section 8.6.

## 8.1 Motivation

TCP uses the slow start procedure, starting with an initial window size of one segment, to gradually probe the network for bandwidth. This procedure is invoked at the time a new connection begins and when an existing connection (such as a P-HTTP connection) resumes after being idle for longer than a threshold. These procedures are specified in [51,54], which recommend an idle period threshold of one round trip time (RTT) or one retransmission timeout value (RTO). The TCP implementation in BSD/OS sets the threshold to the latter, and so does our implementation of TCP session.

There are two reasons why slow start is invoked under these conditions:

1. Since no data has been sent (at least) during the past RTT, ack clocking would have died down. Slow start provides a convenient way to get ack clocking going again for the new transfer.
2. Since the sender has not probed the network in the recent past (during which time the load in the network could have changed unpredictably), it should be conservative in making an assumption about the available network capacity. Consequently, it starts off with a small initial congestion window size of one segment.

Such invocations of slow start interact adversely with the typical patterns of Web access, which involve periods of activity interspersed by idle periods that are usually at least several seconds long [65]. The idle periods would cause the congestion window to be reset and slow start to be invoked each time activity resumes. This happens even when the same connection is reused, as in P-HTTP. Slow start leads to poor bandwidth utilization, and hence increased latency, especially when the transfer length is short and the bandwidth-delay product is large.

A possible solution is to cache and reuse the congestion window size from the recent past, rather than invoke slow start with an initial window size of one segment. This possibility has been mentioned, for instance, in the T/TCP functional specification [13] and TCP Control Block Interdependence [103] documents. However, such a scheme runs afoul of point #1 made above: a large initial window in the absence of ack clocking could lead to a large burst of packets and, consequently, heavy packet loss. Neither [13] nor [103] presents an implementation or a performance analysis of this problem.

Rate-based Pacing [105], developed contemporaneously with our work, attempts to resolve this problem by using the TCP Vegas estimate of a connection's rate to pace out packets smoothly. While this avoids bursts of back-to-back packets, it could still run into problems if the network load has increased significantly in the interim. In this case, even pacing packets out based on the old rate leads to congestion, and hence packet loss. As we show in Section 8.4.2.2, this leads to poor performance for the connection itself as well as other competing connections.

Next, we discuss our solution, *TCP fast start*, and point out how it addresses the problems mentioned above, while achieving very good performance.

## 8.2 Design of TCP Fast Start

Like the solutions mentioned above, TCP fast start exploits *persistent* state. The sender caches and reuses information gathered about the network conditions in the recent past, rather than repeating the slow start discovery procedure each time. The cached information[1] includes the congestion window size (*cwnd*), the slow start threshold (*ssthresh*), and the smoothed round-trip (*srtt*) time and its variance (*rttvar*).

Before getting into the details, we mention two important objectives in accordance with the robustness principle of Section 1.8.

1. If the cached information is still valid, fast start should help improve performance. However, if this information is stale (for instance, because of a sudden surge in network load), fast start should not lead to worse performance than if standard slow start had been used in the first place. In other words, slow start places a lower bound on the actual performance.
2. The performance gains of fast start should not be at the expense of other connections. It is okay for the other connections to suffer to the extent that the fast start connection tries to use its share of the bottleneck bandwidth, but not because the fast start connection is being too aggressive.

It is probably quite difficult to meet these goals exactly without introducing a lot of complexity in the network (such as per-connection state in the routers). TCP fast start tries to do its best with

1. Throughout this chapter, the phrase "cached information" refers to cached protocol variables (such as *cwnd*), and not cached Web page contents. Also, we use the phrases "cached information" and "persistent state" interchangeably.

only a simple and easy-to-implement *priority dropping* mechanism in the routers. Our experimental results (Section 8.4) show that fast start is quite successful in meeting these goals.

While fast start is equally applicable to a standard TCP connection as well as to a TCP session, for ease of exposition we present our discussion in terms of a single TCP connection.

### 8.2.1 Basic Idea

Various studies (e.g., [7], [88]) have shown that the available network bandwidth tends to be stable for periods ranging from a few minutes to tens of minutes. Since the length of a typical pause during an active Web browsing session tends to be much shorter (for instance, [65] reports a median of 15 seconds), it may be reasonable to assume that in the common case the network conditions would not have changed much during such a pause. Consequently, a sender employing fast start caches information about the network conditions and reuses it to expedite start up when the connection resumes activity. To avoid sending out a large burst of packets (due to the absence of ack clocking), the sender uses the cached values of *cwnd* and *srtt* to smooth out the burst.

However, even the smoothed out rate may be too much in (the presumably less common but still important) case of an increase in the level of network load that renders the cached information invalid. This could cause heavy packet loss, and hence degraded performance, for the connection attempting fast start as well as for other connections. This is undesirable given our robustness objectives. To avoid such degradation, packets sent during the fast start phase are accorded a high *drop* priority[2]. When faced with congestion, routers in the network preferentially drop fast start packets. This limits the adverse impact that an over-aggressive fast start attempt could have on other connections.

The heavy packet loss suffered by a connection during fast start could result in worse performance than standard slow start. To avoid this, we augment the sender with new loss recovery algorithms that quickly detect the failure of a fast start attempt and fall back to standard slow start.

Thus, TCP fast start has two components — the sender algorithm and the router algorithm. We discuss each in turn.

2. In the rest of this chapter, we use the term “low priority” to refer to “high *drop* priority”.

## 8.2.2 Sender Algorithm

Incorporating fast start at the end-hosts involves adding new algorithms to the TCP sender but none to the receiver. The sender algorithm has four components:

1. Initiation and termination of the fast start phase.

2. Initialization of TCP state variables.

3. Use of timers to clock out packets during the first RTT.

4. Quick detection of and recovery from a failed fast start attempt.

### 8.2.2.1 Initiation and Termination of Fast Start

The sender initiates TCP fast start whenever a connection resumes activity after being idle for longer than an RTO but shorter than a maximum threshold (set to 5 minutes by default). We defer the question of having an adaptive maximum threshold or aging the cached information gradually to future research.

Packets sent during the fast start phase are marked as low priority. These include all packets sent in the initial window after the connection resumes, except for the first one (which would have been sent in any case by standard slow start). Packets beyond the initial window, i.e., those sent after ack clocking sets in, do not belong to the fast start phase and hence are not marked.

Fast start could terminate prematurely if the sender detects multiple packet losses during fast start (Section 8.2.2.4).

### 8.2.2.2 Initialization of State Variables

When fast start is initiated, TCP state variables are initialized using their most recent values. The variables of interest are *cwnd*, *ssthresh*, *srtt* and *rttvar*.

*Cwnd* is set to the most recent congestion window size at which an entire window of data was transmitted successfully, i.e., without any packet loss. This window size depends on whether the connection was in the slow start phase or in the congestion avoidance phase just before the pause. In the former case, *cwnd* is set to half its old value. In the latter case, it is set to its old value minus one segment.

The cached values of *ssthresh* (which is a "safe" estimate of the size of the network data pipe), and *srtt* and *rttvar* (which together determine the retransmission timeout value) are left unchanged[3].

### 8.2.2.3 Clocking Out Data During Fast Start

The potentially large congestion window at the time fast start is initiated can result in a large burst, which is clearly undesirable. Since it will take at least one RTT for ack clocking to set in, we need an interim method of clocking out packets during fast start.

Our solution is to have the sender use a fine-grained timer to clock out data until ack clocking sets in. The sender has a parameter, *maxburst*, that can be configured to limit the size of any burst it sends out. The sender spaces apart such *maxburst*-sized bursts in time by *maxburst*srtt/cwnd*. This ensures that the bursts are spaced apart uniformly over an RTT, which is ideally how long the fast start phase should last. We set *maxburst* to 4 segments in our experiments[4].

There is evidence to suggest that the overhead of software timers is not likely to be significant in modern computer systems with fast processors. For instance, [29] reports an overhead on the order of a few microseconds. Furthermore, the timer overhead is unlikely to be a significant addition to the cost of taking interrupts and processing acks that goes with ack clocking. Finally, timers are required only during the fast start phase, which is about one RTT in duration.

### 8.2.2.4 Quick Detection/Recovery From Failed Fast Start Attempt

The information cached by the sender could be stale. A particularly dangerous situation is when several new connections become active between the time when the network parameters are cached and when the sender tries to reuse them. In this case, a fast start attempt could be too aggressive. The priority dropping mechanism shields other (non-fast start) connections in such a situation. However, the connection that is attempting fast start could itself suffer heavy packet loss, yielding worse performance than if it had used standard slow start. We augment the TCP loss recovery procedure as follows (to avoid this undesirable situation):

3. It is possible to assign a larger weight to fresh RTT samples during fast start to enable quick adaptation in case of a significant change. However, we have not experimented with this.
4. Note that even standard TCP with delayed acks can burst out 3 segments in a row during slow start.

- ***Fine-grained reset timer:*** The TCP timestamp option is used to obtain accurate RTT samples, from which accurate estimates of *srtt* and *rttvar* are computed. These are used to set a fine-grained timer[5] to detect the loss of fast start packets. We believe that the additional overhead of fine-grained timers is acceptable because it is only incurred for fast start packets. Such packets have a higher likelihood of being dropped because of their low priority.

- ***Slow start without penalty:*** If the fine-grained reset timer expires and the earliest unacknowledged packet is a fast start packet other than the first one (which, as discussed in Section 8.2.2.1, does *not* have the drop priority bit set), the TCP sender cuts *cwnd* down to one segment and initiates slow start. However, it does not cut down *ssthresh*, back off RTO, or discard selective ack (SACK [66]) information (if available). The idea is to make the behavior as close as possible to the case where the connection just did standard slow start in the first place (and did not attempt fast start at all).

  Unlike the loss of a regular packet, the loss of a fast start packet is an indication that the (optimistic) assumption made about network conditions is invalid rather than an indication of congestion.The low priority assigned to the original fast start packets is of critical importance. This makes it okay for the sender to fall back to the default behavior (i.e., slow start), but not incur the penalty associated with a standard retransmission timeout. If the network is truly congested, the connection will discover it during the ensuing slow start.

- ***Recovery from multiple losses:*** When there is packet loss during fast start, the sender (as usual) attempts to recover from it without resorting to a timeout. However, in some cases such a loss recovery procedure could be slow and could lead to significantly worse performance than if fast start had not been used at all. For instance, TCP NewReno recovers at the rate of one loss per RTT, which is equivalent to operating with a window size of one segment until all the packets lost in the burst have been retransmitted (this is assuming there is no more new data to send, as is often the case with short Web transfers).

---

5. The timeout value is still computed as *srtt* + 4**rttvar*, with fine-grained estimates of *srtt* and *rttvar* instead of coarse-grained ones.

We use the following heuristic to limit such performance degradation. If selective ack information (SACK) is available, the sender uses it to recover from multiple losses within one RTT. If not and if the sender receives a partial new ack (indicating the loss of multiple packets [45]) during fast recovery, it cuts *cwnd* down to one segment and initiates slow start right away, without waiting for a timeout (and without imposing any other penalty, as discussed earlier).

- ***Capitalizing on successful transmission during a failed fast start attempt:*** Although a fast start attempt may have failed because of multiple packet drops, some packets may actually have been delivered successfully. Therefore, the sender uses cumulative ack and selective ack information, both from the fast start phase and from the subsequent slow start phase, to avoid retransmitting such packets. In contrast, the default TCP behavior is to discard selective ack information obtained prior to a timeout [66].

As our results in Section 8.4.2.2 show, these techniques are of critical importance in limiting performance degradation when the cached information is stale.

Next, we briefly discussed the router mechanism needed to support fast start.

### 8.2.3 Router Mechanism

The router mechanism we use is simple, packet-level priority dropping. It distinguishes between packets based on a one-bit priority field. When its buffer fills up and it needs to drop a packet, it first picks a low-priority packet, if available. Since fast start packets are assigned a low priority, this algorithm ensures that an over-aggressive fast start does not cause (non-fast start) packets of other connections to be dropped[6]. This mechanism does not require routers to do per-connection processing or maintain per-connection state. The drop priority mechanism applies equally well to drop-tail and RED routers [33].

We retain standard FIFO scheduling for all packets. This keeps the operation of the router simple when there is no need to drop a packet (presumably, the common case). Of course, FIFO scheduling means that fast start packets could increase the queuing delay for other connections, but typically queuing delay has a minor impact on TCP performance compared to packet drops.

6. As we discuss in Section 8.5, such a mechanism when used in conjunction with FIFO scheduling does *not* fully shield the high-priority (non-fast start) packets.

The notion of packet drop priority is also of interest in other contexts. The cell loss priority mechanism (CLP) in ATM provides the same functionality at the granularity of cells. There is a growing interest in using the IP type-of-service (TOS) mechanism to support differentiated services (including packet drop priority) in the Internet [25]. This effort is being supported by the major router vendors. For our purposes it is sufficient if drop priority is supported just by the bottleneck routers.

In spite of this, the lack of widespread support for the drop priority mechanism in today's Internet would impede the deployment of fast start on a large scale. However, as we discuss in Section 8.5, even a partial deployment could yield significant performance benefits.

Next, we briefly describe our implementation of fast start in BSD/OS.

## 8.3 Implementation in BSD/OS

We have implemented fast start both in the *ns* simulator and in the BSD/OS TCP/IP stack. We describe the latter here.

The fast start phase is initiated by the *tcp_output()* function. When it is called to send out data, *tcp_output()* checks to see if the connection has been idle for longer than the retransmit timeout period (RTO). If it has and if fast start has been enabled for the connection, a new value of the congestion window (*cwnd*) is computed. This is either half the old *cwnd* (if old *cwnd* < *ssthresh*) or *cwnd* minus *maxseg* (if old *cwnd* > *ssthresh*). Here *maxseg* is the maximum segment size for the connection. Thus, *cwnd* is set to the most recent congestion window size at which an entire window of data was transmitted successfully. The values of *ssthresh*, *srtt*, and *rttvar* are left unchanged.

The quantities *fs_startseq* (= *snd_nxt* + *maxseg*) and *fs_endseq* (= *snd_nxt* + *cwnd*) are computed to mark the start and end sequence numbers of the fast start phase. *Snd_nxt* is the next packet in sequence to be sent, and *maxseg* accounts for the segment that the TCP sender would have sent in any case when invoking slow start after an idle period. The current value of *ssthresh* is also saved for possible use later should the fast start attempt fail.

All segments that contain data bytes in the range (*fs_startseq,fs_endseq*) have the fast start bit set[7]. This identifies them to routers as low priority packets. However, the use of this bit is not standardized, so only routers with our *linkemu* module (Appendix A) installed will recognize it.

To avoid bursting out data during the fast start phase, *tcp_output()* breaks up potentially large bursts into smaller ones, each at most *maxburst* (by default, 4) segments in size, and spaces them apart according to the rate of the connection (*cwnd/srtt_exact*). The quantity *srtt_exact* is a smoothed and fine-grained RTT estimate computed using the TCP timestamp option with timestamps of millisecond granularity. The default timestamp resolution of 500 ms is too coarse-grained to be useful here.

Having computed the rate, *tcp_output()* computes the length of time (*delta*) over which *maxburst* segments should be spread out. It also records the times at which the last *maxburst* packets were transmitted. Using these, it ensures that no more than *maxburst* data packets are sent out in a time interval of length *delta*. If more packets are waiting to be sent, a software timer is used to schedule *tcp_output()* at a later time. Note that the software timer needs to be invoked only when bursts larger than *maxburst* segments are generated. This is likely to happen only during fast start phase (which lasts for about one RTT).

While in the fast start phase, the TCP sender enables the fast start *reset timer* instead of the standard retransmit timer. The timeout value for the reset timer is computed as *srtt_exact* + *4*rttvar_exact*, where the "exact" quantities are computed as discussed above. Rather than using a separate software timer, the firing of the reset timer is tied to the fast TCP timer (200 ms period, typically). This is in contrast to the slow TCP timer (500 ms period, typically) that the retransmit timer is tied to.

If the reset timer expires while fast start packets are still outstanding[8], the sender immediately *resets* its state to what it would have been had fast start not been attempted. *Cwnd* is reset to one segment, *ssthresh* is restored using the value saved at the time fast start was initiated, and slow

7. It is possible to use a generic "drop priority" bit in the IP header (such as one of the IP type-of-service bits), but we decided to pick an *unused* bit just to be safe. We could not find any in the IP header, so we picked one in the TCP header.
8. Note that if even the first packet sent without the fast start bit set has not yet been acknowledged, the sender treats it as a standard retransmission timeout.

start is initiated. However, none of the other penalties of a retransmit timeout (such as timer back-off) is imposed.

As explained in Section 8.2.2.4, fast start is also aborted if the sender detects the loss of multiple packets in a window during fast start phase. This detection is done in *tcp_input()* by looking for a partial new ack during TCP fast recovery, just as in TCP NewReno [45].

We now present detailed performance results.

## 8.4 Performance Results

We first present performance results based on our BSD/OS implementation of TCP fast start. A limitation of these experiments is the lack of priority dropping in the Internet routers today. For this reason, we also present detailed performance results based on simulation.

The protocol configurations we evaluated include:

1. *tcp*: TCP NewReno with standard slow start.
2. *fs*: TCP fast start without priority dropping.
3. *fs-pdrop*: TCP fast start with priority dropping.
4. *fs-pdrop-noflr*: TCP fast start, but without any of the fast loss recovery techniques from Section 8.2.2.4.

Each of these was used in conjunction with a scheme such as P-HTTP or TCP session.

### 8.4.1 BSD/OS Measurements

In this experiment, we have a client download Web pages of different sizes from a server. The client is the one we used for our experiments with P-HTTP (Section 5.5). The server is Apache version 1.3b5 using either the *tcp* or *fs* configurations. In each run of the experiment, the client first downloads a 100 KB Web page. This enables the server to obtain and cache information about the network path between it and the client. Then after an idle period of several seconds (much longer than the RTO), the client downloads another page from the server. In the *tcp* configuration, the sec-

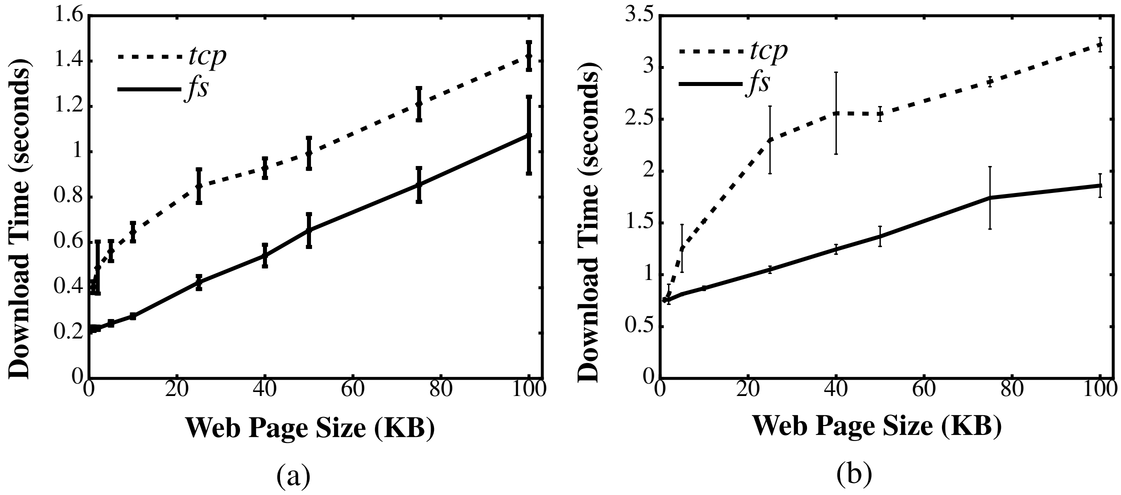



**Figure 8.1 Download time versus Web page size for the (a) terrestrial WAN network, and (b) DirecPC network.**

ond download would start off in slow start, but in the *fs* configuration it invokes fast start based on the information cached during the first download.

We conducted the experiment over two network paths — a terrestrial wide-area network and one including the DirecPC satellite network. These are the same as the ones used in our P-HTTP experiments (Section 5.5).

Figure 8.1 shows the download time as a function of the Web page size. Since the HTML file is only 1 KB in size, the Web page size is determined primarily by the sum of the sizes of the individual inline images. The mean and a 95% confidence interval on the mean are computed over 15 runs of the experiment.

We observe that *fs* performs significantly better than *tcp* in both the WAN and the DirecPC networks. The performance improvement, both in relative and absolute terms, tends to be larger in the latter than in the former. This is because the RTT of the DirecPC network is much larger (400 ms versus 70 ms). This means that each RTT saved with *fs* is more valuable in the DirecPC case. Also, a larger RTT implies a larger bandwidth-delay product, so *fs* can potentially cache and reuse a larger initial window size.

We also observe that the benefit of *fs*, in relative terms, tends to diminish as the Web page gets larger. For instance, in the WAN case, it drops from over 50% (2X) for pages under 10 KB in size

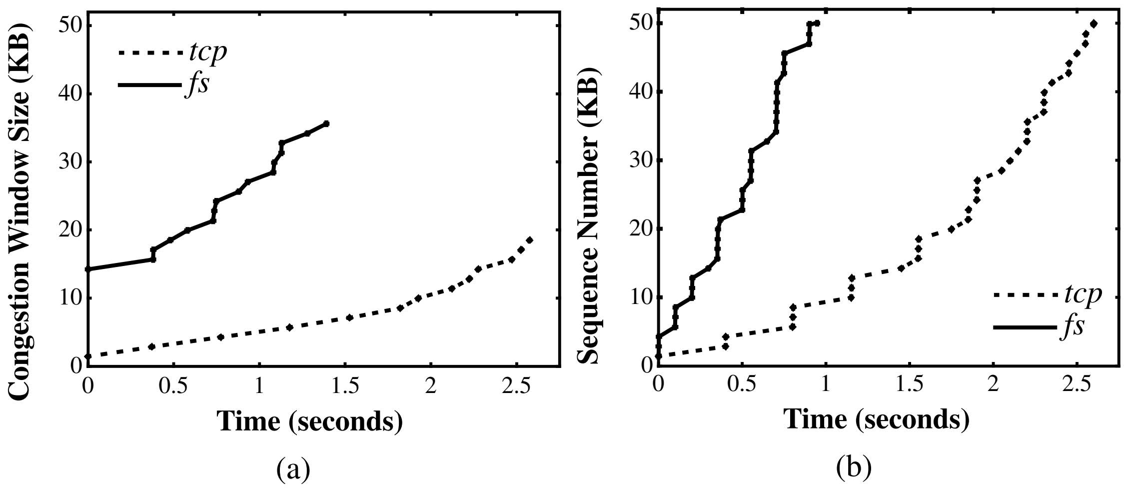



**Figure 8.2 (a) The congestion window plot, and (b) the sequence number plot for the second of two 50 KB Web page downloads.**

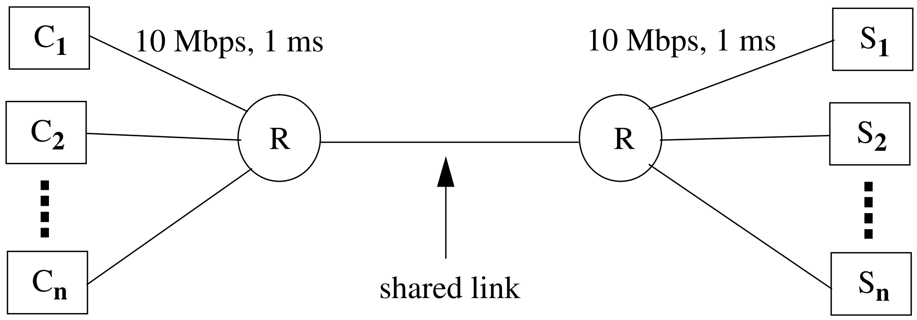
to about 25% for 100 KB pages. This is because the longer a transfer, the less important the slow start phase at start up time. However, Web page transfers tend to be short in length (for instance, [65] reports an average of around 30 KB), so fast start is quite valuable in practice.

Figure 8.2 shows the operation of fast start in greater detail for the second of two 50 KB downloads. Figure 8.2(a) depicts the growth of the congestion window over time. We observe that *fs* starts off with an initial congestion window size of about 15 KB, derived from the value cached during the first download. This translates into a much quicker completion time (about 2.75X) for the second download with *fs* compared to *tcp*, as shown in Figure 8.2(b).

A shortcoming of experiments over the Internet is the lack of support for priority dropping in the routers. To address this limitation, we now turn to simulation experiments.

## 8.4.2 Simulation Results

We present performance results using our implementation of TCP fast start in the *ns* network simulator. The topology used for the simulation experiments is shown in Figure 8.3. Depending on the experiment, one or more of the client-server pairs initiate Web page downloads across the shared link[9]. For each such pair, two downloads are done, with an intervening idle period. The first download, 100 KB in size, gives the server the opportunity to cache information about the network path to the client. We measure and report the performance of the second download.




**Simulation parameters**

Maximum TCP window size: 100 KB
Router buffer size: 20 packets

**Web download parameters**

Request size: 300 bytes
HTML size: 1 KB
Total size of inline images: 40 KB

**Figure 8.3 The network topology and parameter settings for the simulation experiments.**

Again depending on the experiment, cross-traffic in the form of TCP bulk transfers may be introduced between other servers and clients contending for the shared link. We consider several settings for the bandwidth and the delay of for the shared link:

1. 28.8 Kbps, 50 ms (i.e., an RTT of 100 ms)

2. 1.5 Mbps, 50 ms

3. 1.5 Mbps, 200 ms (i.e., an RTT of 400 ms)

4. 45 Mbps, 200 ms

The protocol combinations we consider are *tcp*, *fs*, *fs-pdrop*, and *fs-pdrop-noflr*. Note that the implementation of these in *ns* does not use the three-way handshake procedure at connection setup time. In effect, each is using T/TCP-style accelerated open that avoids consuming an RTT during connection setup.

9. We prefer to use the term "*shared* link" rather than "*bottleneck* link" because there are situations where this link is fast enough that it is not a bottleneck. However, we do use the term "bottleneck link" in a few instances where it is appropriate.

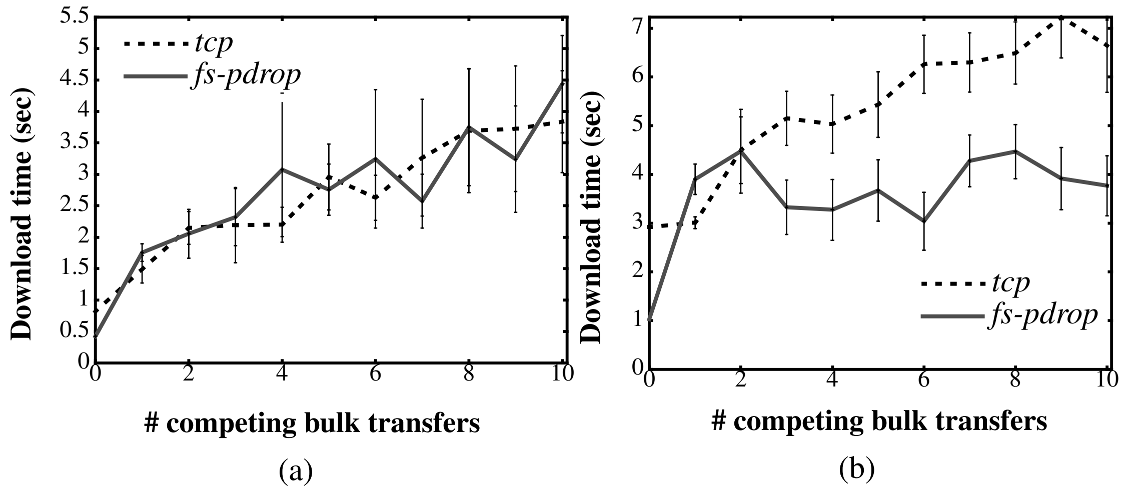



**Figure 8.4 The completion time for the second of two Web downloads under conditions of constant load. The bottleneck link bandwidth is 1.5 Mbps and the delay is (a) 50 ms, and (b) 200 ms.**

First, we consider the performance of a single Web download in the presence of cross-traffic in the form of bulk transfers.

#### 8.4.2.1 Single Web Download with Constant Load

In the constant-load experiment, the bulk transfers constituting the cross-traffic are left running throughout the experiment to provide a roughly constant background load. Under such conditions, it is likely that cached values of TCP parameters remain valid after an idle period. By varying the number of bulk transfer connections from one experiment to the next, we can evaluate fast start under different levels of cross-traffic load.

After the cross-traffic has reached its steady-state level, a single Web download, 100 KB in length[10], is initiated between client $C_1$ and server $S_1$. After a long pause, a second download, 40 KB in length, is initiated between the same pair of hosts. This mimics a Web browsing session in which a user downloads multiple pages from the same server, with intervening idle periods. We report the time taken for the second download (which is in a position to exploit cached information, if fast start is used).

10.We pick a larger size, 100 KB, for the first download to emulate the combined effect of multiple downloads that may have happened in the past.

Figure 8.4(a) and Figure 8.4(b) show the results for a 1.5 Mbps bottleneck link with two different settings of delay — 50 ms and 200 ms. We make several observations. In the 50 ms case, the performance of both *tcp* and *fs-pdrop* is similar. Furthermore, *fs* (not shown) performs only about 15% better than *fs-pdrop*. The reason for these is that the bandwidth-delay product of the network is rather small (1.5 Mbps*100 ms = 18.75 KB). And since this is shared among multiple connections, the share of each is even smaller. Under these conditions, the Web connection does not build up a large window at the time of the first download. Consequently, the initial window size during the subsequent fast start is not much larger than the default (i.e., one segment), so fast start only has a limited impact. The relatively small performance difference between *fs* and *fs-pdrop* arises because in the latter case, priority dropping makes the Web connection more prone to losing packets.

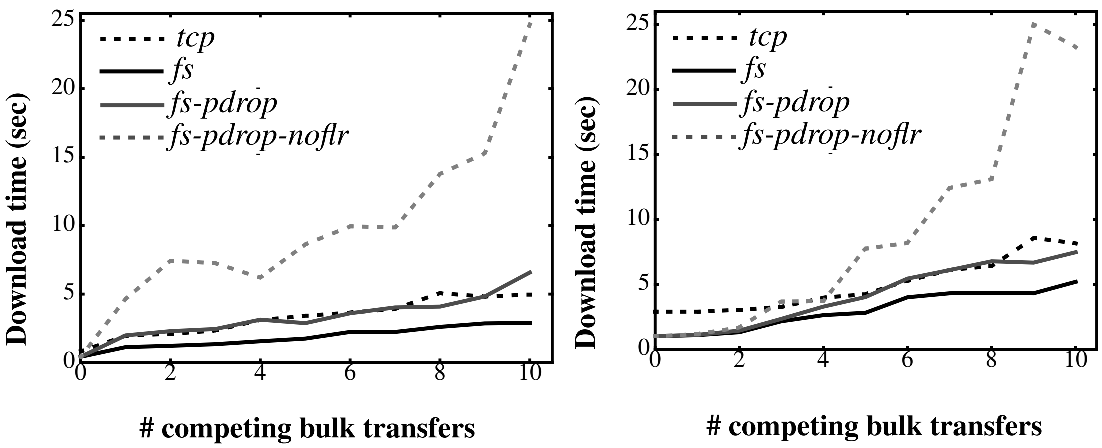
When the delay is 200 ms, the bandwidth-delay product is four times as large, about 75 KB. Consequently, fast start has a far more significant performance impact. As Figure 8.4(b) shows, *fs-pdrop* improves download time by 30-50% (1.43-2X) compared to *tcp*. Also, *fs* (not shown) performs about 5% better than *fs-pdrop*.

When the bandwidth is set to 45 Mbps and delay to 200 ms, the bandwidth-delay product is very large, about 2.25 MB. This makes fast start very beneficial. Regardless of the level of cross-traffic (in the range that we considered), the download time with *tcp* is about 2.85 seconds and that with *fs* or *fs-pdrop* is about 0.96 seconds, a 3X improvement.

In summary, we make two observations. First, the benefits of fast start are greater when the bandwidth-delay product is larger. Second, under conditions of constant load, there is not much of a difference in performance between *fs* and *fs-pdrop.* This is because the cached information used by fast start remains fairly accurate, so fast start causes few packet drops. As a result, the use of priority dropping does not make much of a difference in this case.

#### 8.4.2.2 Single Web Download with Changed Load

This experiment is similar to the previous one except for one significant difference — the cross-traffic is absent at the time of the first Web download, but is introduced into the network during the idle period between the two downloads. Therefore, the network conditions will have changed for




(a) (b)

**Figure 8.5 The completion time for the second of two Web downloads under conditions of changed load. The bottleneck link bandwidth is 1.5 Mbps and the delay is (a) 50 ms, and (b) 200 ms.**

the worse between when the Web connection caches various network parameters (at the time of the first download) and when it tries to reuse them (at the time of the second download). Figure 8.5(a) and Figure 8.5(b) show the results for a 1.5 Mbps bottleneck link with a delay of 50 ms and 200 ms, respectively. For the sake of clarity, we do not show the error bars.

In the 50 ms case, the performance of *fs-pdrop* is similar to that of *tcp*. When fast start is initiated using stale information, it causes the already loaded network to drop several packets. In spite of this, *fs-pdrop* suffers little performance degradation because the augmented loss recovery procedure (Section 8.2.2.4) quickly detects and recovers from the failed fast start attempt. The value of the augmented procedure is further underscored by the performance of *fs-pdrop-noflr*, which only uses the standard TCP loss recovery procedure. The heavy loss of packets results in a download time that is up to 5X worse than that in the case of *tcp*.

When *fs-pdrop* is used with a coarse-grained reset timer instead of one that is fine-grained, the performance degradation relative to *tcp* is only 1.35X or less (not shown in the figure). Much of the benefit of the augmented loss recovery procedure is due to the three other techniques discussed in Section 8.2.2.4 rather than fine-grained timer.

The performance of *fs* is 40-50% (1.67-2X) better than that of *fs-pdrop* and *tcp*. In the absence of priority dropping, packet drops tend to be spread across the Web connection and the bulk transfers.

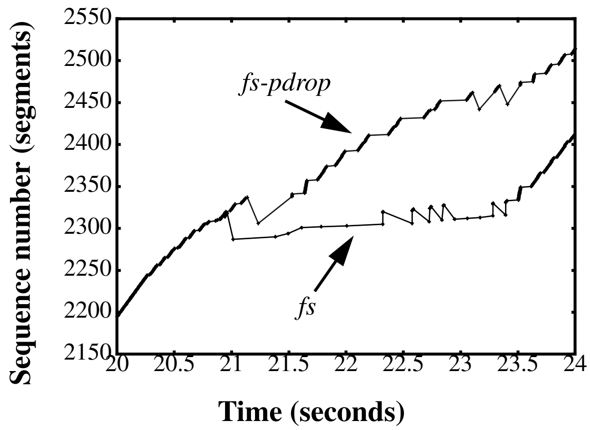



**Figure 8.6 A section of the sequence number trace of a bulk transfer connection in the immediate aftermath of simultaneous fast start attempts by ten Web connections. The bottleneck link bandwidth is 1.5 Mbps and the delay is 50 ms.**

The Web connection, whose fast start attempt is primarily responsible for the packet drops, does not suffer much. On the other hand, with *fs-pdrop* the Web connection bears the brunt of the packet drops.

Fast start without priority dropping (*fs*) achieves better performance at the cost of the ongoing bulk transfer connections. This is clear from Figure 8.6, which shows the sequence number trace of a bulk transfer connection in the immediate aftermath of a fast start attempt by ten Web connections. Due to the absence of priority dropping, *fs* causes the bulk transfer to lose several packets, and consequently, stall for several seconds. On the other hand, *fs-pdrop* has a much smaller impact on the bulk transfer. Its impact is primarily in the form of increased queuing delay.

When the bottleneck link delay is 200 ms, the performance trends are similar to the 50 ms case. However, the larger bandwidth-delay product places the network in a better position to absorb the burst of packets due to fast start. This leads to fewer packet losses. Therefore, the impact of both the augmented loss recovery procedure and that of priority dropping is smaller than in the 50 ms case.

In summary, fast start coupled with priority dropping is resilient under adverse conditions, i.e., when fast start is attempted based on stale information. The augmented loss recovery procedure

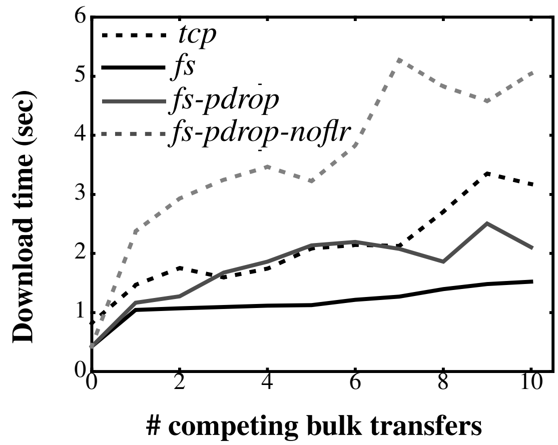



**Figure 8.7 The completion time for the second of two Web downloads under conditions of changed load. The bottleneck link bandwidth is 1.5 Mbps and the delay is 50 ms. The bottleneck router uses RED buffer management with a buffer size of 20 packets, a minimum marking threshold of 3 packets, a maximum marking threshold of 12 packets, and a weight of 0.02 for computing an exponentially smoothed estimate of the average queue length.**

limits performance degradation for the connection attempting fast start while priority dropping protects other traffic.

### 8.4.2.3 Impact of RED Buffer Management

So far, we have only considered drop-tail buffer management. We now briefly consider the impact of Random Early Detection[11] (RED) buffer management [33]. RED enables high link utilization while at the same time maintaining free buffer space to absorb bursts. So there are fewer packet losses due to fast start, even when it uses stale information. Consequently, the download time improves compared to the drop-tail case. The across-the-board improvement in performance is evident when one compares Figure 8.7 to Figure 8.5(a).

Packets sent during the fast start phase are just as likely to get marked as other packets. Also, if a fast start packet gets marked but the corresponding (short) transfer terminates before the sender invokes congestion control action, the sender preserves this notification by caching the new

11.We consider the variant of RED that marks packets with an ECN notification rather than drop them.

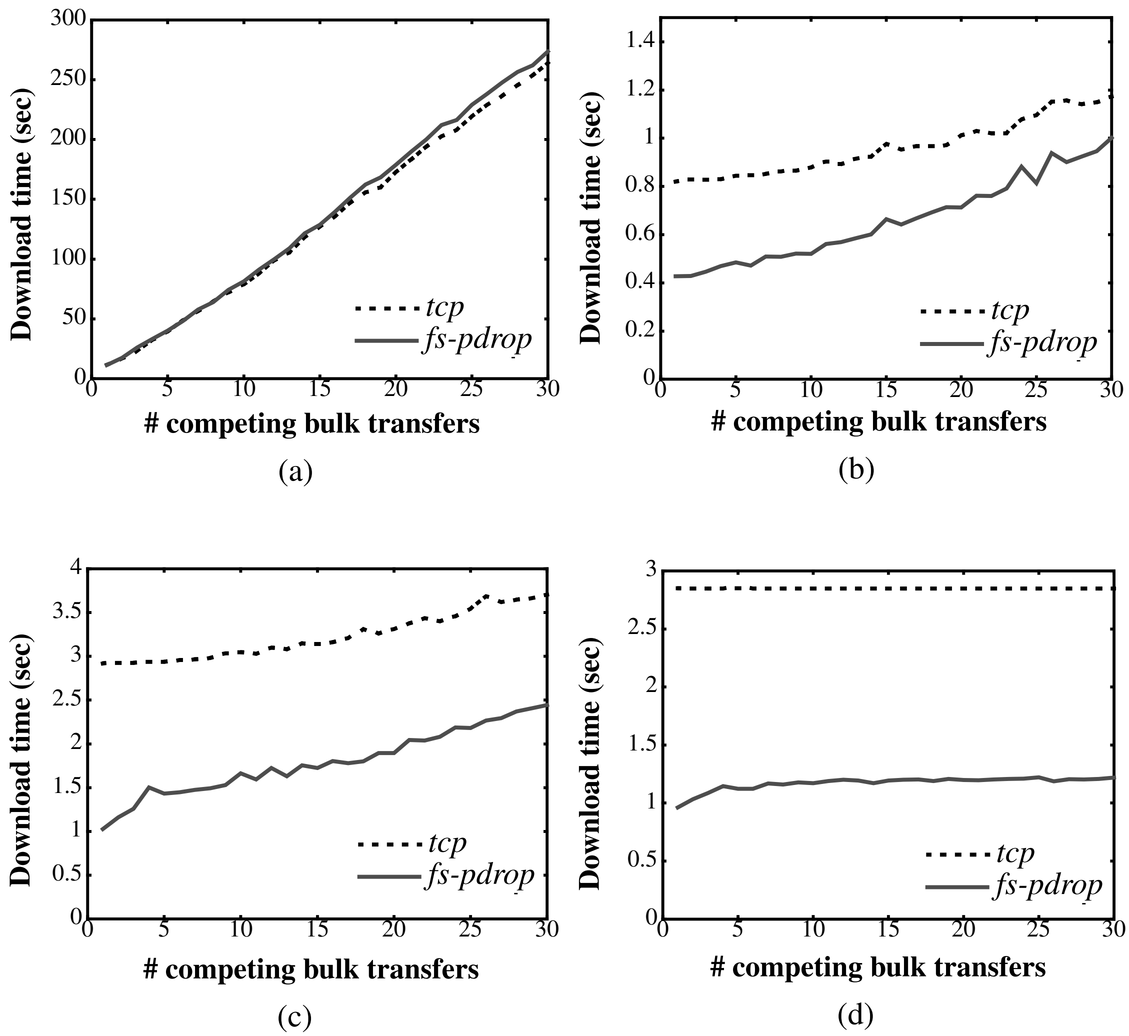



**Figure 8.8 The average download time for competing Web downloads under conditions of constant load. The bandwidth and delay of the shared link are (a) 28.8 Kbps/50 ms, (b) 1.5 Mbps/50 ms, (c) 1.5 Mbps/200 ms, and (d) 45 Mbps/200 ms.**

(halved) congestion window. Then the next fast start attempt will be less aggressive. Thus, the use of persistent state makes an explicit congestion notification scheme such as RED effective even when the individual transfers are short in length.

### 8.4.2.4 Competing Web Downloads with Constant Load

We now consider performance in the presence of competing Web downloads, but without any bulk transfer cross-traffic. Each run of the experiment involves between 1 and 30 client-server pairs initiating a download that traverses the shared link. The initiation times are spread uniformly at random over a 10-second interval. Then, after a long idle period, each client-server pair does a second download, again spread over a 10-second interval. So this experiment is similar to that discussed in Section 8.4.2.1 in that the overall level of load in the network (i.e., the number of concurrent downloads) remains roughly constant between the time of the first download and that of the second.

In Figure 8.8, we report the time taken for the second download (averaged across clients) for various configurations of the shared link. We make two observations. First, the performance benefit of fast start is larger when the bandwidth-delay product is larger. In particular, there is little performance gain when the bandwidth is as low as 28.8 Kbps. As explained in Section 8.4.2.1, a larger bandwidth-delay product provides fast start with the opportunity to cache and reuse a larger initial window size.

Second, the performance benefit of fast start, in relative terms, tends to decrease as the number of concurrent downloads increases. For instance, with a 1.5 Mbps/50 ms bottleneck link, the reduction in download time of *fs-pdrop* relative to *tcp* decreases from 50% to 15% (2X to 1.18X) as the number of concurrent downloads increases from 1 to 30. Likewise, the performance benefit drops from 65% to 35% (2.85X to 1.5X) in the case of a 1.5 Mbps/200 ms bottleneck link. The reason for these trends is that an increased level of load diminishes the opportunity for an individual connection to cache and reuse a large window. It is only in the case of a 45 Mbps/200 ms link that this trend is not evident. The levels of load that we consider (i.e., up to 30 concurrent downloads) is not large enough to impact performance, given the very large bandwidth-delay product.

In summary, a significant improvement in performance is possible even when there are multiple connections that initiate fast start concurrently. As the load in the network increases, the performance benefit decreases gradually.

#### 8.4.2.5 Competing Web Downloads with Changed Load

This experiment is similar to the previous one expect that the second download by each client is initiated within a 0.1-second interval (as against a 10-second interval for the first download).

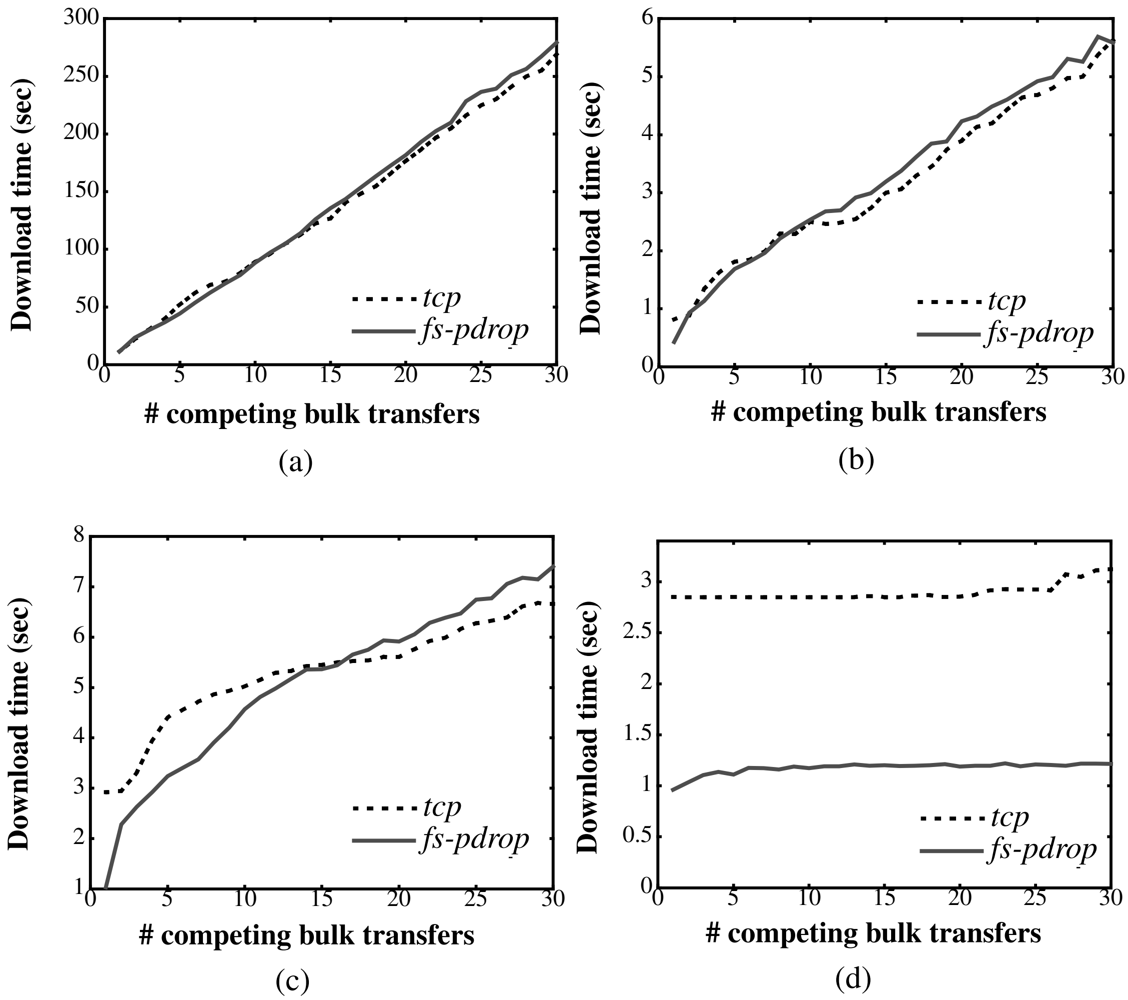



**Figure 8.9 The average download time for competing Web downloads under conditions of changed load. The bandwidth and delay of the shared link are (a) 28.8 Kbps/50 ms, (b) 1.5 Mbps/50 ms, (c) 1.5 Mbps/200 ms, and (d) 45 Mbps/200 ms.**

Therefore, as in the changed-load experiment (Section 8.4.2.2), a fast start attempt at the time of the second download would be based on stale information.

Such a change in load mimics the *flash crowd* phenomenon often observed in the Web, where a large number of users (clients) converge on a Web server almost simultaneously, often because of some extraordinary news event. In such a situation, information cached by the server during a (normal) period of relatively modest activity becomes stale.

Figure 8.9 shows the results for various configurations of the bottleneck link. Comparing this to Figure 8.8, we observe that the performance benefit of fast start disappears in most cases. This is because the use of stale information causes fast start to invariably fail at the time of the second download. The only exception is the 45 Mbps/200 ms case, where the large bandwidth-delay product prevents significant packet loss in spite of fast start being over-aggressive. Even in the cases where fast start fails, the augmented loss recovery procedure limits performance degradation of *fs-pdrop* to less than 10%.

In summary, this experiment again underscores the resilience of fast start in the face of adverse conditions, including those that mimic flash crowds.

#### 8.4.2.6 Competing, Heterogeneous Web Downloads with Changed Load

In this experiment, we consider a mixture of connections, some that use fast start and others that do not. The experiment involves 20 client-server pairs doing downloads in two phases. The first phase is spread over a 10-second interval and the second over a 1-second interval. One half of the client-server pairs (i.e., 10 of them) use fast start while an equal number do not.

| Protocol | Without fast start | With fast start |
|---|---|---|
| *tcp* | 3.37 (0.20) | n/a |
| *fs* | 4.09 (0.17) | 3.32 (0.19) |
| *fs-pdrop* | 3.38 (0.16) | 3.20 (0.18) |

**Table 8.1 Completion time (in seconds) for downloads that use fast start and those that do not. A 95% confidence interval for the mean is reported in parentheses. Note that in the case of *tcp*, there are no downloads that use fast start.**

Table 8.1 summarizes the results. We observe that both with *fs* and with *fs-pdrop*, the use of fast start results in little performance gain (under 5%) compared to *tcp*. This is as expected because the change in the network load renders the cached information stale, so fast start tends to fail. The more important point is that when fast start is used by a subset of the connections in conjunction with priority dropping (*fs-pdrop*), connections that do not use fast start experience little performance degradation. In contrast, when fast start is used without priority dropping (*fs*), the download time for the latter degrades by over 20% compared to the case where fast start is not used at all (*tcp*). This, again, underscores the importance of using priority dropping in conjunction with fast start.

### 8.4.3 Summary of Results

We now summarize our key results both simulation-based and implementation-based:

1. Fast start has the potential to cut down latency significantly (up to 3X in our experiments) under a variety of conditions. The performance gains tend to be largest when the bandwidth-delay product is large, the network load is low and when the Web page size is neither too small nor too large.
2. Even when the conditions are not favorable (e.g., high network load, or worse still, changing network state that renders the cached information stale), fast start at worst performs similar to standard TCP. The augmented loss recovery procedure is effective in quickly detecting and recovering from a failed fast start attempt.
3. The use of priority dropping in conjunction with fast start is critical in avoiding performance degradation for other connections in case of an over-aggressive fast start attempt based on stale information.
4. RED buffer management improves the performance of fast start.

Next, we turn to a discussion of some general issues pertaining to fast start.

## 8.5 Discussion

We discuss issues pertaining to the performance benefits of fast start, its robustness, and incremental deployment.

TCP fast start is complementary to the P-HTTP and TCP session techniques that we have developed in previous chapters. In fact, there is a synergy between those techniques and fast start. P-HTTP and TCP session make the congestion control parameters more stable and reliable compared to when there are lots of short, independent and competing connections. Therefore, fast start is in a better position to cache and successfully re-use this information.

Both Web clients and Web servers benefit from fast start. From the viewpoint of a client, fast start means faster Web page downloads. From the viewpoint of a server, it means increased capacity because resources (such as the user-level process/thread, socket buffers, etc.) associated with a particular download are freed sooner and can be used to serve other requests.

The robustness of TCP fast start comes from it use of priority dropping and its dependence on past information. If the network becomes congested, a fast start attempt is likely to fail. If the connection is only able to build up a small window during the subsequent slow start, its next fast start attempt will be much less aggressive. This self-regulating mechanism ensures that senders do not keep flooding the network with packets bound to get dropped. Of course, this assumes that hosts are cooperative, just as TCP congestion control does. To be safe, the network should have mechanisms to detect and penalize malicious behavior (e.g., [32]).

While priority dropping is a useful mechanism to exercise control on a time-scale finer than a round trip time, it may not, however, completely shield high-priority packets from losses due to low priority packets. This is because of a combination of FIFO queuing, which could delay the servicing of high-priority packets when there are low-priority packets ahead of it in the queue, and the policy of dropping packets only when the queue fills up. To address this problem, it is possible to use a more sophisticated (but still stateless) priority dropping scheme such as the one discussed in [3], which preemptively drops packets even before the queue fills up.

Support for priority dropping is not widespread in the Internet today. However, this situation is set to change with major router vendors including such mechanisms in their newer products. This is change is being driven by the Internet's *differentiated services* effort [25], which is standardizing priority dropping as one in a set of per-hop behaviors. The underlying philosophy is that packets that do not conform to a profile (*out-of-profile*) should be dropped preferentially with respect to packets that do conform (*in-profile*). Our use of drop priority in conjunction with fast start exactly matches this philosophy if we define the "profile" to be standard TCP slow start. Fast start packets are out-of-profile and therefore dropped preferentially.

In spite of this, we cannot depend on universal support for priority dropping at this time. A proxy-based approach provides a path for incremental deployment in certain situations, while still obtaining significant performance benefits. For instance, consider the DirecPC satellite network. If all client Web accesses are routed via a proxy at the earth station, much of the benefits of fast start can be obtained by having the proxy implement fast start and by deploying priority dropping in the network domain encompassing the proxy and the clients. Neither the Web servers nor the routers in the rest of the Internet need be modified. (And in any case, fast start does not require any modification to the clients.) An added advantage is that, with all communication routed via it, the proxy would tend to have better estimates of the bandwidth available to each client than any individual server would. In Chapter 9, we discuss another instance, namely an asymmetric-bandwidth access network, where such a partial deployment would be feasible and beneficial.

Finally, although the focus of this chapter has been the Web, TCP fast start could also be useful in other situations involving bursty data transfer. Examples include wide-area transactions, RPC, and sensor data collection.

## 8.6 Summary

In this chapter, we have presented a new technique, *TCP fast start*, to complement the P-HTTP and TCP session techniques that we developed earlier. The basic idea behind fast start is to use cached information (i.e., persistent state) to decrease the overhead of probing using TCP slow start. If fast start succeeds, it cuts down latency significantly, especially for short transfers. However, to avoid performance degradation in adverse circumstances, priority dropping and augmented loss recovery procedures are used in conjunction with fast start.

Thus far in this thesis, we have considered performance issues that arise due to bandwidth and delay limitations, in general. In the next chapter, we consider the performance issues that arise specifically due to asymmetry in network characteristics in a new class of access networks. Although it only form a small part of the end-to-end network path, an access network could have a significant impact on performance. As we shall see, TCP fast start could help improve performance significantly in such networks even though their inherent delay may not be very large.

# Chapter 9

# Asymmetric Access Networks

In previous chapters, we considered performance problems that arise due to bandwidth limitations and/or large delays. In some cases, the limitations arise due to the characteristics of the *access* network, such as the low bandwidth of a dialup modem line or the large delay of a satellite network.

In this chapter, we consider a special class of access networks, namely *asymmetric access networks*, where performance problems arise not so much due to the inherent bandwidth or delay limitations as due to asymmetry in the network characteristics. We begin by motivating why asymmetric access networks are important to study, especially in the context of Web performance. We also provide a general definition of network asymmetry. Then in Section 9.2, we analyze the performance problems in detail via experiments in our network testbed. In Section 9.3, we discuss various solution techniques that we have developed. An evaluation of these solutions appears in Section 9.4. In Section 9.5, we present a discussion of how the techniques we have developed may be useful in other contexts as well, and how they may be extended. Finally, we present a summary in Section 9.6.

## 9.1 Motivation

The explosion in the popularity of the Internet (and the Web in particular) in recent years has driven increasing numbers of users to get connected to the Internet. This has lead to a growing demand for bandwidth. Advances in fiber-optic and switching technologies hold the promise of solving part of the problem by making the core of the network extremely fast. So the performance bottleneck has tended to shift towards the periphery of the network, in particular to access networks that provide "last-mile" connectivity to users in homes and small offices.

The predominant access network technology in use today is dialup phone lines. However, the limited bandwidth of this technology has lead to the development and deployment of alternative access network technologies of much higher speed. These include networks based on cable modem, ADSL and satellite technologies (Section 1.6). These share the characteristic that bandwidth is *asymmetric*, i.e., it is 10-1000 times larger in the *downstream* direction (i.e., towards the end user) than in the *upstream* direction (i.e., away from the end user). There are two reasons for the growing popularity of such asymmetric networks:

1. Due to technological reasons, it is easier and more cost-effective to provide high bandwidth from a central location (such as a the headend of cable network plant or a satellite earth station) towards end users than in the opposite direction. The reasons include the poor quality of wiring and ingress noise in the upstream direction in cable networks, and upstream channel access and power limitations in satellite networks.

2. Applications, in particular Web browsing, tend to have asymmetric communication requirements. An order-of-magnitude more data flows in the downstream direction than in the upstream direction. This coupled with the predominance of Web access among applications has motivated service providers to deploy asymmetric access networks as a cost-effective way of meeting the demand for bandwidth.

However, in spite of the asymmetry in traffic, asymmetry in bandwidth has significant performance implications for TCP transfers. This is because a steady stream of acknowledgements is needed to sustain the flow of data. A disruption in the flow of acks in the upstream direction could adversely impact the downstream bandwidth utilization. Furthermore, Web requests need to traverse the constrained upstream link, which adds to the potential for there being a performance problem.

Although we primarily focus on *bandwidth asymmetry*, we present a general definition that also encompasses other forms of asymmetry as discussed below:

**Definition:** *A network is said to exhibit network asymmetry with respect to TCP performance, if the throughput achieved is not solely a function of the link and traffic characteristics in the direction of data transfer, but depends significantly on those in the opposite direction as well.*

In addition to asymmetry in bandwidth, this definition extends to other types of asymmetry, such as those arising in media-access and packet error rate. As we point out in Section 9.5, a similar set of solutions is effective in addressing both bandwidth and media-access asymmetry. However, we focus on bandwidth asymmetry in this chapter, and present only a brief discussion of media-access asymmetry. A more detailed treatment of the latter appears in our paper [5] and in a contemporaneous thesis [4].

## 9.2 Analysis of Performance Problems

We now analyze the problems that arise due to bandwidth asymmetry. The experiments were conducted using the wireless cable modem network in our testbed (Section 3.1.2 and Figure 3.2). The cable modem network has a downstream bandwidth of 10 Mbps. The upstream link is much slower, usually a 28.8 Kbps dialup phone line. In some of our experiments, we used an Ethernet segment as the upstream link. While such a configuration is probably unrealistic, it provides a useful data point corresponding to when there is no asymmetry.

We consider the cases of unidirectional and bidirectional transfers separately. We restrict ourselves to bulk transfers here. When we present experimental evaluation of our solutions (Section 9.4), we also consider short, Web-like transfers.

### 9.2.1 Unidirectional Transfer

In this case, we assume that data transfer happens only in the downstream direction. This closely approximates the case where clients download Web pages but do not send out any data (other than

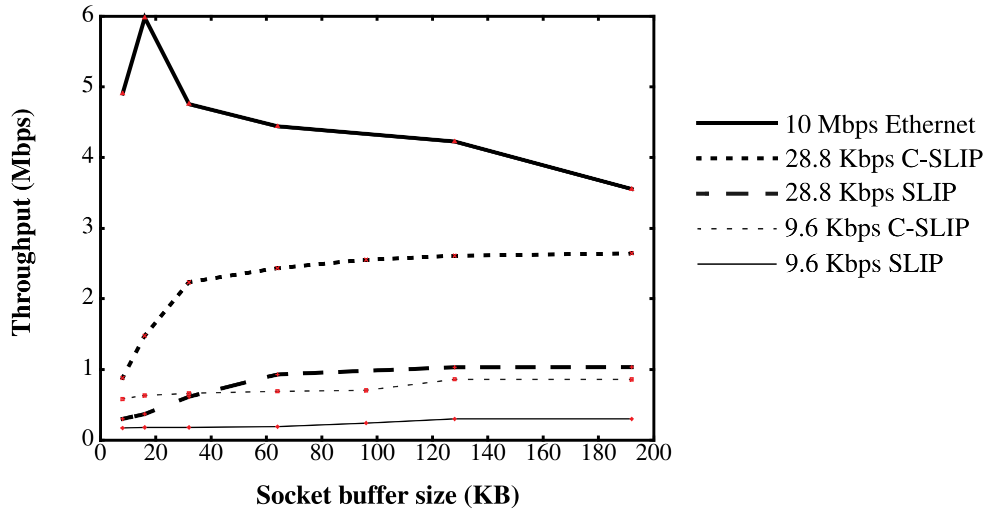



**Figure 9.1 Measured performance of the Hybrid wireless cable network using different upstream links, across a range of socket buffer sizes. Each run of the experiment involved the transfer of 1 MB of data in the downstream direction.**

short Web requests) in the upstream direction. For ease of exposition, we restrict ourselves to a single TCP transfer in the downstream direction.

As in [63], we define the following:

**Definition:** *The* **normalized bandwidth ratio***,* k*, between the downstream and upstream directions is the ratio of the raw bandwidths divided by the ratio of the packet sizes used in the two directions.*

For example, with bandwidths of 10 Mbps and 100 Kbps in the downstream and upstream directions, respectively, the raw bandwidth ratio is 100. With 1000-byte data packets and 40-byte acks, the ratio of the packet sizes is 25. So, $k$ is 100/25 = 4. This implies that if there is more than one ack for every $k = 4$ data packets, the 100 Kbps bottleneck link in the upstream direction will get saturated before the downstream bottleneck link does, possibly limiting downstream throughput, as we now discuss

The main effect of bandwidth asymmetry is that TCP ack clocking could break down. Consider two data packets transmitted by the sender in quick succession. While in transit to the receiver, these packets get spaced apart (in time) according to the bottleneck link bandwidth in the downstream direction. The principle of ack clocking is that the acks generated in response to these packets preserve this spacing all the way back to the sender, enabling it to clock out new data packets with the same spacing.

However, the constrained upstream bandwidth and consequent queuing effects could alter the inter-ack spacing. When acks arrive at the upstream bottleneck link at a faster rate than the link can support (i.e., when $k > 1$, assuming every data packet is acknowledged), they get queued behind one another. The spacing between them when they emerge from the link is *dilated* with respect to their original spacing. (This is in contrast to the widely-studied *ack compression* phenomenon ([110], [73]), which happens when acks get queued at a fast link, i.e., $k < 1$). Thus, the sender clocks out new data at a slower rate than if there had been no queuing of acks. Another consequence is that the growth of the sender's window size slows down.

This is part of the reason why the downstream throughput with a dialup upstream link running SLIP [95] is so low (Figure 9.1). SLIP header compression (C-SLIP) [52] reduces the sizes of acks and hence decreases $k$, thereby improving performance. For instance, consider bandwidths of 10 Mbps and 28.8 Kbps in the two directions, and a data packet size of 1 KB. With the TCP timestamp option enabled, the ack size is 52 bytes with SLIP and 18 bytes with C-SLIP. So $k$ is 18.05 with SLIP and is 6.25 with CSLIP. With TCP delayed acks (i.e., one ack for every two data packets), throughput is limited to 10*2/18.05 = 1.1 Mbps and 10*2/6.25 = 3.2 Mbps, with SLIP and C-SLIP respectively. These numbers closely match the measured throughputs reported in Figure 9.1.

In contrast, the performance with an Ethernet upstream channel is much better because of the absence of bandwidth asymmetry ($k$ is 0.052) and a much smaller link delay than the dialup lines. As an aside, the dip in throughput beyond a socket buffer size of 16 KB is because the increased burstiness causes router queue(s) along the downstream path to overflow.

In practice, the upstream bottleneck link will also have a finite amount of buffer space. If the TCP transfer lasts long enough, this buffer can fill up and cause acks to be dropped. If the receiver acknowledges every packet, then $(k-1)$ out of every $k$ acks get dropped, on average, at the upstream buffer. Since, in effect, only one ack traverses the upstream bottleneck link for every $k$ data packets, acks may not directly limit downstream throughput. However, the decreased frequency of acks could cause other problems, as we now explain.

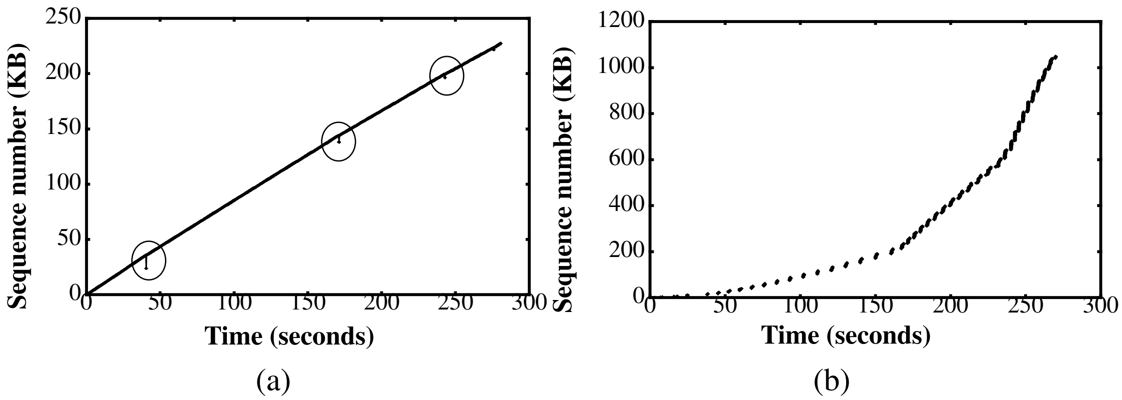



**Figure 9.2 Sequence number traces for (a) the upstream transfer and (b) the downstream transfer over the 10 Mbps cable modem network with a 9.6 Kbps dialup upstream link. Sharp upswings in the data-rate of the downstream transfer happen precisely when the upstream transfer loses packets (indicated by the circles in (a)) and consequently slows down.**

First, the sender becomes bursty. If the sender receives only one ack in *k*, it ends up sending out data in bursts of *k* packets. This increases the chance of data packet loss, especially when *k* is large. Second, since conventional TCP senders tie the growth of their congestion window size to *how many* acks have been received rather than *how much* data has been acknowledged, fewer acks imply slower growth of the window. Third, the receipt of fewer acks could disrupt the sender's fast retransmission algorithm when there is a data packet loss. The sender may not receive the threshold number of duplicate acks although the receiver may have sent out more than the required number. And finally, the loss of the (now infrequent) acks elsewhere in the upstream path could stall the sender for long periods of time.

Next, we turn to bidirectional transfers.

## 9.2.2 Bidirectional Transfers

We now consider the case when TCP transfers[1] simultaneously occur in the downstream and upstream directions. This is representative of the case where a user is sending out data (for instance, an e-mail message) while simultaneously receiving other data (for example, Web pages)[2]. We restrict our discussion to the case of one connection in each direction.

1. We use the terms "transfer" and "connection" interchangeably in our discussion.

In this scenario, the effects discussed in Section 9.2.1 are more pronounced, because some of the upstream bandwidth is used up by the upstream transfer, leaving only the balance for the acks of the downstream transfer. This increases the normalized bandwidth asymmetry for the downstream transfer.

In addition, there are other effects that arise due to the interaction between data packets of the upstream transfer and acks of the downstream transfer. Assume that the upstream transfer is initiated first and that it saturates the upstream link and its buffer with its data packets. When the downstream transfer is initiated at a later time, there is a high probability that many its acks will encounter a full upstream buffer and hence be dropped. Even after these initial problems are overcome, acks of the downstream connection could often get queued behind one or more (large) data packets of the upstream connection. The queuing delay could be large. For instance, it takes about 280 ms to transmit a single 1 KB data packet over a 28.8 Kbps line. This causes the downstream transfer to stall for long periods of time.

Figure 9.2(a) and Figure 9.2(b) show the progress of concurrent upstream and downstream transfers, respectively, in the wireless cable modem network. As discussed above, the upstream transfer, which is initiated first, causes the downstream transfer to make very slow progress initially. This is clear from Figure 9.2(b) which shows large idle periods until about 170 seconds into the transfer. It is only at times when the upstream connection loses packets (due to buffer overflow) and slows down temporarily that the downstream connection gets the opportunity to make rapid progress and quickly build up its window. This is evident from the sharp upswings in the data rate of downstream connection at the times when the upstream connection loses (and retransmits) packets.

The situation is quite different when the downstream transfer is initiated before the upstream one. The acks of the downstream connection could quickly fill up the upstream buffer[3]. So when the upstream connection is launched, its data packets may encounter a full buffer and be dropped. Such early packet losses would cause the sender of the upstream connection to suffer retransmission timeouts and back off its timer. In the extreme case, this could cause the upstream connection

2. When more than one user/host shares the asymmetric access network, the downstream and upstream transfers may be initiated by different users.
3. The capacity of a router buffer is often limited by the number of packets it can hold, rather than the number of bytes. This is the case with BSD/OS-based router.

to make little or no progress so long as the downstream transfer is active. This interaction between the transfers in the two directions is the opposite of that discussed above (i.e., when the upstream transfer is initiated first).

In summary, performance in the presence of bidirectional traffic is quite unpredictable, with the possibility of transfers in one direction completely shutting out those in the other direction.

### 9.2.3 Summary of Performance Problems

Here is a summary of performance problems caused by bandwidth asymmetry:

1. In the case of a unidirectional transfer, the ack stream could get disrupted due to queueing and/or packet drops. While header compression helps, it does not eliminate the problem entirely.
2. In the case of bidirectional transfers, performance is quite unpredictable, with the possibility of either the downstream transfer or the upstream one starving out the other.

Having identified the basic performance issues, we now present our solution.

## 9.3 Solution to the Problems Caused by Bandwidth Asymmetry

Our solution has two components: (a) reducing disruption in the ack stream due to congestion in the upstream direction; and (b) having the sender adapt to disruptions in the ack stream, or alternatively, masking such disruptions from the sender. We discuss each in turn.

### 9.3.1 Reducing Disruption in the Ack Stream

As noted in Section 9.2, when there is a high degree of asymmetry in bandwidth, acks could congest the upstream link. Depending on the size of the upstream link buffer and the length of the downstream data transfer, this congestion could cause acks to be dilated and/or fill up the buffer (and then be dropped). The solution is to decrease the frequency of acks. We discuss two ways of doing so, either end-to-end or locally at the upstream router.

#### 9.3.1.1 Ack Congestion Control (ACC)

The idea here is to extend congestion control to TCP acks, since they consume a non-negligible fraction of resources in the bandwidth-constrained upstream direction. For this purpose, the TCP receiver needs to be made aware of the congestion caused by acks. In general, the TCP receiver (usually the client) may not be on a host that is directly attached to the upstream link[4]. Therefore, we employ the RED algorithm [33] on the queue for the upstream link to track the average queue size and to generate explicit congestion notifications (ECN) when appropriate. Such notifications are carried in upstream packets and are reflected back by downstream packets of corresponding connection. Since, in our BSD/OS platform the buffer capacity is limited by the *number* of packets it can hold, data and ack packets are equally likely to be marked by the RED algorithm.

When it receives an ECN, the TCP receiver reduces the frequency of acks. It does so by employing a generalized version of the TCP delayed ack algorithm. The receiver maintains a dynamically varying *delayed ack factor*, $d$, and sends one ack for every $d$ data packets received. By default, $d$ is set to 2, corresponding to the standard delayed ack policy of sending an ack for every other data packet received. When it receives an ECN, the receiver increases $d$ multiplicatively, thereby decreasing the frequency of acks also multiplicatively. Then, for each subsequent round-trip time (determined using the TCP timestamp option) during which it does not receive an ECN, it linearly decreases the factor $d$, thereby increasing the frequency of acks. Thus, the receiver mimics the standard congestion control behavior of TCP senders in terms of the frequency with which it sends acks.

There are bounds on the delayed-ack factor $d$. Obviously, the minimum value of $d$ is 1, since at most one ack is sent per data packet. The maximum value of $d$ is determined by the sender's window size, which is conveyed to the receiver in a new TCP option[5]. The receiver should send at least one ack (preferably more) for each window of data from the sender. Otherwise, it could cause the sender to stall until the receiver's delayed-ack timer (usually set at 200 ms) expires and forces an ack to be sent.

---

4. For instance, both the downstream and upstream link may terminate at a cable modem box that then connects to hosts in a residential or small-office environment via an Ethernet LAN.
5. It may be possible to do away with such an option, and instead have the receiver decide that $d$ is too large when there is data that it has still not acknowledged and it has not received any new data for longer than a threshold period. However, we have not experimented with such an algorithm.

In summary, ACC has the advantage that the TCP receiver has control over which acks get sent and which do not[6]. Also, it does not require routers in the network to explicitly identify packets belonging to a connection, so it will work in conjunction with an end-to-end security mechanism such as IPSEC [2]. On the downside, though, the RED algorithm may not respond quickly enough to congestion when transfers are short in length. We discuss this issue in Section 9.4.1.

#### 9.3.1.2 Ack Filtering (AF)

Ack filtering, based on a suggestion by Karn [60], is an alternative to ACC. It is a router-based technique that decreases the number of acks sent over the bandwidth-constrained upstream link by taking advantage of the cumulative nature of TCP acks.

When an ack is about to be enqueued, the router (or the end-host's routing layer, if the host is directly connected to the constrained link) checks if any older acks belonging to the same connection are already in the queue. If there are any, it removes some fraction (possibly all) of them, depending on how full the queue is. The removal of these "redundant" acks reduces the delay due to queuing of acks, and frees up space for other data and ack packets. The policy that the router uses to pick the acks to remove (i.e., how many and which acks to remove) is configurable. However, in our experiments we use the simple policy of removing all preceding acks belonging to a connection whenever a new ack with a larger cumulative ack arrives on the same connection.

AF has the advantage that, unlike ACC, it does not involve any changes to the end-to-end TCP protocol. Also, since the filtering is done locally at the upstream router, there is no delay in responding to congestion. However, AF requires the router to identify acks that belong to the same connection, which may be difficult to do if TCP headers are encrypted as in IPSEC. This would also make it difficult to avoid filtering out acks that have special significance (e.g., duplicate acks). Finally, AF does not impact the rate at which acks *arrive* at the queue. As we discuss in Section 9.4.2.1, this, when combined with acks-first scheduling, can lead to starvation of upstream connections.

6. Some acks (such as duplicate acks) may be more important than others depending on the transport-layer information they carry.

#### 9.3.1.3 Acks-first Scheduling

Acks-first scheduling attempts to resolve the contention between acks and data, which could lead to unpredictable performance in the presence of bidirectional traffic (Section 9.2.2). It schedules acks at a strictly higher priority than data over the upstream link.

Acks-first scheduling is motivated by the observation that ack packets (with header compression enabled) are much smaller in size than typical data packets. The transmission time of acks is usually small enough not to have a significant impact on the upstream transfer. On the other hand, the large data packets of the upstream transfer could have a very significant impact on the downstream transfer. Acks-first scheduling reduces this impact substantially.

Note that as with the RED algorithm used in conjunction with ACC, acks-first scheduling does not require the router to explicitly identify or maintain state for individual TCP connections. The higher scheduling priority of ack packets could be conveyed to routers using the ToS (type-of-service) bits in the IP header.

### 9.3.2 Adapting to Disruption in the Ack Stream

Bandwidth asymmetry could cause a severe disruption in the ack stream of a downstream transfer (in terms of the frequency of acks and the spacing between them), even if ACC, AF and/or acks-first scheduling is used. Since the TCP sender depends on acks to clock out new data, such a disruption in the upstream direction could have a significant impact on the flow of data in the downstream direction. There are two ways of addressing this problem —having the sender adapt to disruptions in the ack stream, or shielding the disruptions from the sender. We discuss each in turn.

#### 9.3.2.1 Sender Adaptation (SA)

The loss of acks and/or a change in the inter-ack spacing could cause the sender to become more bursty and could slow down the growth in its window size.

To avoid sending out large bursts, the sender places a limit, *maxburst*, on the burst size, just as in TCP fast start (Section 8.2.2.3). By default, we set *maxburst* to 4 packets. If there are more than *maxburst* packets waiting to be sent, the sender groups the packets into smaller bursts (each of size at most *maxburst*) and spaces them apart in time.

The sender avoids a slowdown in window growth by simply taking into account the *amount* of data acknowledged by each ack, rather than the *number* of acks received. So, if an ack acknowledges $s$ segments, the window is grown as though $s$ separate acks had been received. Such a policy is appropriate because the window growth should only be tied to the available bandwidth in the *downstream* direction, so the number of acks received is irrelevant.

#### 9.3.2.2 Ack Reconstruction (AR)

Ack reconstruction is an alternative to sender adaptation that *shields* the sender from disruptions in the ack stream, thereby obviating the need for the sender to adapt. AR reconstructs the ack stream after it has traversed the constrained upstream link. Reconstruction involves inserting new acks to bridge large gaps in the ack sequence, and smoothing out the inter-ack spacing. In this sense, AR can be viewed as the converse of AF.

A detailed description of AR appears in our paper [5]. However, we deprecate it in favor of SA for two reasons. First, SA builds on some of the same mechanisms that we had developed for TCP fast start, while AR introduces new mechanisms. Second, SA is easier to deploy. In particular, it can be deployed at a proxy on the premises of the *downstream* service provider. In contrast, AR would need to be deployed on the premises of the *upstream* service provider (such as an ISP providing standard dialup service), who may be completely unaware of the asymmetry in the client's network connectivity.

## 9.4 Performance Evaluation

Having discussed our solution techniques in detail, we now present a performance evaluation of the techniques. Our evaluation is based on our implementation of the solution techniques in the *ns* network simulator. While we have also implemented the techniques in the BSD/OS protocol stack, as of the writing of this dissertation, the wireless cable modem network in our testbed has been dismantled. So we are unable to conduct experiments on the real network.

The topology used for our simulation experiments is shown in Figure 9.3. It consists of a client and server communicating via an asymmetric network. The link speed and delay settings are chosen to be representative of the Hybrid wireless cable modem network.

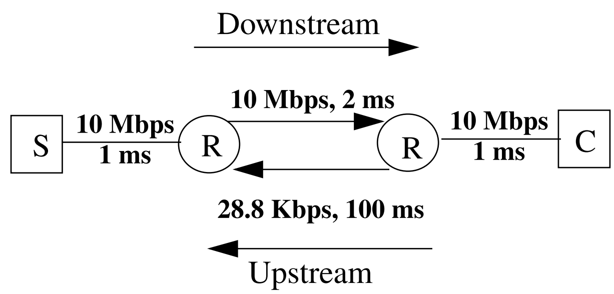



**Figure 9.3 The topology for our simulation experiments.**

We consider the following basic protocol configurations for the downstream transfer and the upstream router:

1. *tcp*: TCP NewReno
2. *af*: TCP NewReno with sender adaptation, and ack filtering at the upstream router.
3. *acc*: TCP NewReno with sender adaptation, and ack congestion control at the receiver.
4. *noasym*: TCP NewReno, but without any asymmetry in bandwidth (10 Mbps upstream link). This serves as a benchmark for comparison.

We selectively combined these configurations with other options, including acks-first scheduling at the upstream router (*afirst*) and TCP fast start at the sender of the downstream connection (*fs*). So, for example, the configuration *acc-afirst-fs* refers to ack congestion control with acks-first scheduling and fast start.

We assume that C-SLIP header compression is in use. The size of a compressed ack is assumed to be 18 bytes (this matches the size of compressed acks with the timestamp option that we measured on an actual dialup link). The data segment size is set to 1 KB.

ACC, when enabled, operates in conjunction with RED at the upstream router. The router buffer size is set to 20 packets. A weight of 0.02 is used to compute an exponentially smoothed estimate of the average queue length. The minimum and maximum marking thresholds are set to 3 and 12 packets, respectively.

The upstream connection, when present, uses TCP NewReno.

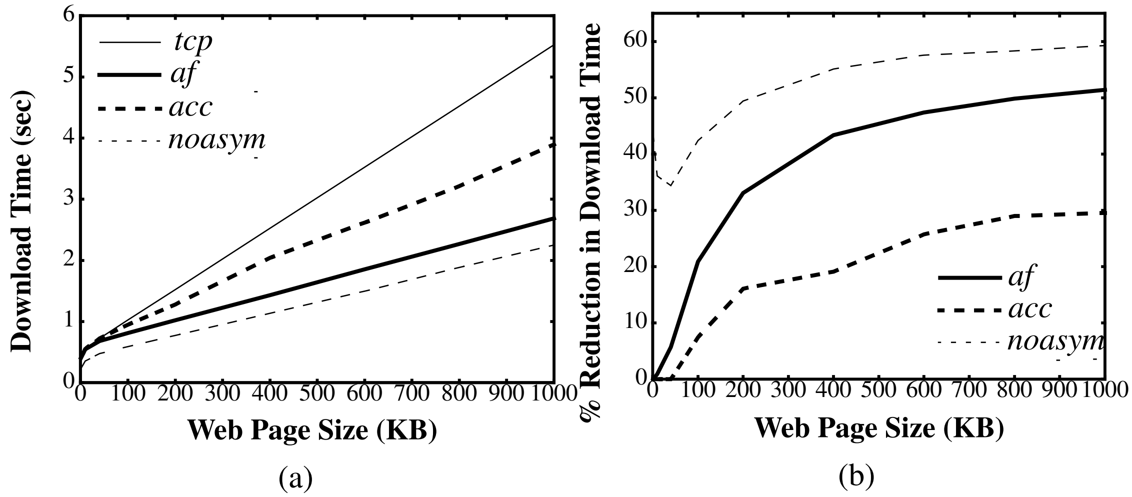



**Figure 9.4 (a) The download time as a function of the Web page size, and (b) the improvement in download time, in percentage terms, relative to *tcp*.**

We now present out experimental results.

## 9.4.1 Unidirectional Traffic

We consider the download of a Web page from the server to the client. The Web page consists of a 1 KB HTML file with one inline image of variable size. The download largely involves unidirectional traffic, with the only data traffic in the upstream direction being two 300-byte HTTP requests, one for the HTML file and the other for the inline image.

Figure 9.4(a) shows the download time as a function of the Web page size (basically the size of the inline image, since the HTML is only 1 KB in size). Figure 9.4(b) shows the reduction in download time, in percentage terms, relative to *tcp*. Asymmetry causes a significant increase in the download time, as evident from the large gap between the *tcp* and *noasym* curves. Both *af* and *acc* bridge this gap to varying degrees. The former is more effective than the latter. In fact, *af* performs quite similar to *noasym*, i.e., the configuration with no bandwidth asymmetry. This is because it is able to avoid upstream congestion by completely eliminating the queuing of acks. In contrast, the RED algorithm, which triggers *acc*, is slower in responding to queuing (because of the exponential smoothing used to compute the average queue length), and moreover, does not eliminate queuing entirely.

It is evident from Figure 9.4 that the performance gains of *af* and *acc,* in absolute terms, grow with the size of the Web page. The percentage gains show a similar trend, although they eventually flatten out. The reason for these trends is that the problem of upstream congestion due to acks becomes significant only when the sender's window size is large enough to cause acks to be generated at a pace faster than the upstream link can support.

This implies that, in the absence of bidirectional traffic, bandwidth asymmetry does not cause significant performance degradation for typical Web page downloads that are only a few tens of kilobytes in length. So *af* and *acc have* little impact in such cases. It is only in the less common case of a large transfer (for example, the download of an MPEG video clip) that these techniques yield significant performance benefits.

Next, we turn to the case of bidirectional traffic.

## 9.4.2 Bidirectional Traffic

We first consider the case of bulk transfers in the two directions, and then turn to Web downloads.

### 9.4.2.1 Bulk Transfers

In this experiment, a bulk transfer is first initiated in the downstream direction, and after it has reached steady state, another transfer is initiated in the upstream direction. Both transfers last for 50 seconds. In Table 9.1, we report the throughput achieved by each transfer during the period that both are active.

| Protocol Configuration | Downstream Throughput (Mbps) | Upstream Throughput (Kbps) |
|---|---|---|
| *tcp* (80%) | 1.80 | 22.7 |
| *tcp* (20%) | 9.93 | 0.00 |
| *af* | 0.54 | 24.0 |
| *af-afirst* | 9.93 | 0.00 |
| *acc* | 1.50 | 19.4 |
| *acc-afirst* | 2.38 | 25.9 |

**Table 9.1 The throughput from the simulation of simultaneous bulk transfers in the downstream and upstream.**

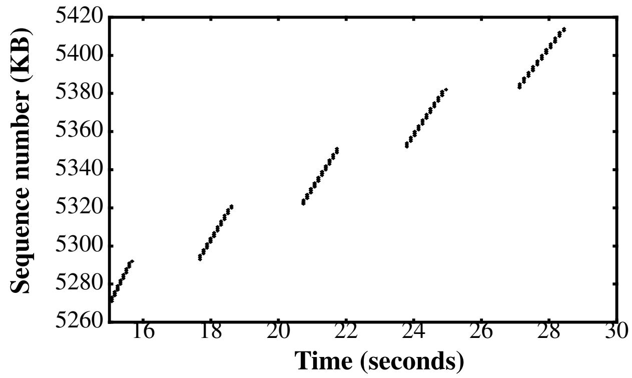



**Figure 9.5 A section of the sequence number trace of the downstream bulk transfer while the upstream connection is active. The configuration in use is *af*. The multi-second idle periods are caused by acks getting queued behind large data packets.**

We make several observations. As discussed in Section 9.2.2, performance in the case of *tcp* is unpredictable. In 20% of the runs of our experiments (11 times out of 50), the acks of the downstream connection quickly fill up almost the entire upstream buffer. Therefore, when the upstream transfer starts up, there is a high probability that one or more of its initial packets get dropped. When this happens a few times in succession, the TCP retransmission timer backs off to such an extent that the upstream connection makes no progress (i.e., it achieves zero throughput).

However, the situation is quite different in the remaining 80% of the runs. While the acks of the downstream connection do queue up at the upstream buffer, they do not fill it up completely at all times (because packets are dequeued regularly for transmission, thereby creating vacancies in the buffer). It is possible for a small number of data packets of the upstream connection to get into the queue. Once this happens, subsequent acks of the downstream connection suddenly experience a much longer delay because they are queued behind one or more large data packets. This causes the downstream transfer to slow down, making it unlikely that its acks will fill up the upstream buffer in the future. The net result is the upstream transfer makes good progress while the downstream one performs poorly.

With *af*, downstream throughput is poor, but upstream throughput is close to optimal. This is so because *af* ensures that no more than one ack of the downstream connection is queued at the upstream buffer at any point in time. The upstream transfer is able to make steady progress, even better than it could with *tcp*. This, in turn, impacts the downstream transfer adversely, both because of significant queuing delay and because of the loss of acks. The downstream connection makes progress in short bursts interspersed by multi-second idle times, as shown in Figure 9.5. The end result is that downstream throughput is much worse than with *tcp*.

With *acc*, the throughput of the upstream transfer is reasonably good (19.4 Kbps as against the maximum possible of 28.8 Kbps). At the same time, the throughput of the downstream transfer (1.5 Mbps) is much better than with *af*. This is because unlike with *af*, *acc* allows there to be multiple acks (belonging to one connection) in the queue. So there are correspondingly fewer data packets in the queue. This, in turn, decreases the impact that the data packets can have on the acks and hence the downstream transfer.

Our discussion thus far shows that neither *af* nor *acc* is, by itself, effective in addressing the performance problems that arise due to bidirectional traffic. The main stumbling block is FIFO scheduling at the upstream router. We now evaluate the effectiveness of acks-first scheduling in this context.

From Table 9.1, we note that *af-afirst* leads to starvation of the upstream transfer. This is because ack packets arrive at the queue at a rate faster than they can be drained out, so there is always an ack waiting to be sent in the queue[7]. Therefore, data packets of the upstream transfer never get the opportunity to be transmitted. In the absence of competition from the upstream transfer, the downstream transfer achieves a high throughput (9.93 Mbps).

The starvation of the upstream transfer with *af-afirst* points to the need to regulate the frequency of acks even before they enter the upstream queue, which is precisely what ack congestion control does. With *acc-afirst*, the upstream throughput (25.9 Kbps) is close to optimal while the downstream throughput (2.38 Mbps) is significantly better than with *tcp*. Ack congestion control decreases the frequency of acks, and header compression makes them small in size, so scheduling them ahead of data packets does not impact the progress of the upstream transfer in a significant way.

---

7. Note that an ack that is in the process of being transmitted is no longer in the queue, so it is not considered by the ack filtering algorithm.

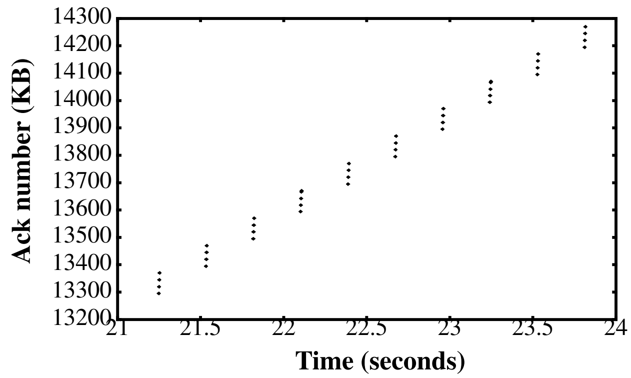



**Figure 9.6 A section of the ack trace for the downstream transfer while the upstream transfer is active. The configuration in use is *acc-afirst*. There is an idle time of about 280 ms between bursts of acks because of the 1 KB data packets belonging to the upstream transfer.**

A simple calculation shows that with our choice of parameter settings, the downstream throughput cannot be better than 2.85 Mbps while sustaining optimal upstream throughput. While a data packet of the upstream connection is undergoing transmission on the 28.8 Kbps link (a period about 280 ms in length), the sender of the downstream connection does not receive any new acks. This is illustrated in Figure 9.6. So it can send at most one window's worth of data in 280 ms. With our maximum window size setting of 100 KB, the downstream throughput can be no more than 100*8/280 = 2.85 Mbps. The throughput of 2.38 Mbps achieved by *acc-afirst* is fairly close to this limit.

In summary, bidirectional transfers cause performance problems due to the competition between data and ack packets for buffer slots and for access to the upstream link. Ack congestion control coupled with acks-first scheduling goes a long way in alleviating these problems.

Next, we consider the performance of a Web download in the presence of bidirectional traffic.

### 9.4.2.2 Web Download

As in the previous experiment, we consider bidirectional traffic. But the bulk transfer in the downstream direction is replaced with a Web page download. The upstream transfer is initiated first. Then, two Web pages, 100 KB and 40 KB in size, are downloaded. A long idle period separates the two downloads. This enables us to evaluate TCP fast start in this environment. In Table 9.2, we report the completion time of the second download for a variety of protocol configurations. In all

| Protocol Configuration | Download Time (seconds) |
|---|---|
| *tcp (without RED)* | 43.07 (0.64) |
| tcp | 12.40 (1.88) |
| tcp-afirst | 3.67 (0.53) |
| *af* | 15.77 (1.88) |
| *af-afirst* | 4.35 (0.45) |
| *af-afirst-fs* | 3.07 (0.34) |
| *acc* | 12.15 (1.34) |
| *acc-afirst* | 4.12 (0.42) |
| *acc-afirst-fs* | 2.80 (0.38) |
| *noasym* | 0.42 (0.00) |

**Table 9.2 The download time for a 40 KB Web page in the presence of bidirectional traffic. The standard error is given in parentheses.**

except the first configuration, the upstream router employs the RED algorithm, which provides ECN feedback to the sender of the upstream connection.

The overall observation is that the constrained upstream bandwidth, coupled with the presence of bidirectional traffic, has a very significant impact on Web page download time. This is clear from the 100X larger download time with *tcp* (without RED) compared to *noasym*.

Much of the increase in the download time comes about because of the large queuing delay experienced by acks at the upstream router, which in turn stalls the Web download in the downstream direction. When this router uses the RED algorithm to provide ECN feedback to the sender of the upstream connection, the number of data packets in the queue decreases considerably. The consequent decrease in the queuing delay reduces the Web download time by 3.5X.

Another significant performance improvement comes when the upstream router uses acks-first scheduling in place of FIFO scheduling. This further decreases the queuing delay experienced by acks and reduces the download time by another factor of 3.5X.

In spite of the above improvements, acks would still get queued at the upstream router when a data packet is in the process of being transmitted. Since this takes about 280 ms in the network we studied, acks would experience a queuing delay equal to about half that, i.e., 140 ms. This adds to the inherent 100 ms RTT of the network, making the effective RTT quite large. This makes TCP fast start an attractive proposition. There is a 30% reduction in download time when fast start is used (*af-afirst-fs* and *acc-afirst-fs*). It is interesting to note that fast start may be useful in an asymmetric-bandwidth network even if the *inherent* delay is small.

In summary, the use of ECN at the upstream router, acks-first scheduling, and fast start help improve performance significantly. The regulation of the ack stream using AF or ACC is not play a critical role in the performance of the downstream transfer in this case.

### 9.4.3 Summary of Results

Here is a summary of our key results:

1. In the case of unidirectional transfers, congestion of acks in the upstream direction could impact downstream throughput. The impact is more significant when the downstream transfer is long. Each of ack congestion control and ack filtering, in conjunction with sender adaptation, helps alleviate the problem. Ack filtering tends to be more effective since it is a local scheme and completely eliminates queuing of acks behind other acks of the same connection, while ack congestion control takes longer to adapt, and in any case, does not eliminate the queuing of acks entirely.

2. In the case of bidirectional transfers, the dominant effect is the interaction between the large data packets of the upstream transfer and the small acks of the downstream one. The data packets can increase the queuing delay for acks very significantly, thereby slowing down the progress of the downstream transfer. The impact can be large even in the case of short Web

transfers. The use of ECN to regulate the flow of the upstream transfer, acks-first scheduling to limit the impact of data packets on acks, and fast start to overcome the large RTT due to queuing delay, all help improve performance significantly.

3. Using acks-first scheduling without regulating the flow of acks could lead to starvation of data packets. Ack congestion control, used in conjunction with acks-first scheduling, avoids this problem.

## 9.5 Discussion

We now discuss how some of the techniques we have developed in this chapter may be improved and how they are useful in addressing problems that arise due to asymmetry in aspects other than bandwidth.

Our performance evaluation in Section 9.4 points out the strengths and limitations of ack filtering and ack congestion control in alleviating congestion due to acks. The advantages of AF are that it operates locally and is effective even when transfers are short in length. On the downside, though, it does not give the receiver the opportunity to decide which acks to discard, and could lead to starvation of upstream transfers when used in conjunction with acks-first scheduling. On the other hand, ACC has the advantage that it gives the receiver control over which acks to keep and which to discard. By regulating the flow of acks end-to-end, it avoids adverse interaction with acks-first scheduling. However, the inherent hysteresis in the RED algorithm and the end-to-end feedback delay make this technique less effective for short transfers.

It may be possible to combine the two techniques to obtain the benefits of each while avoiding their drawbacks. AF would be effective in responding immediately when there is congestion due to acks. At the same time, the history of acks filtered in the past would be used fed into the RED algorithm. This would ensure that ECN notifications would be generated even though there is no actual queuing of acks (because of filtering). If a transfer lasts for long enough, the ECN notifications would enable the receiver to decrease the frequency of acks end-to-end, thereby diminishing the need for filtering acks in the future.

The use of persistent state could also make ACC more effective, especially for short transfers. This is akin to the use of such state in TCP fast start, except that the state would be maintained at the receiver rather than at the sender. The receiver would cache and reuse the delayed-ack factor instead of starting with a default setting of 2 each time.

Finally, as we have discussed in [5], AF and ACC are quite effective in combating problems due to media-access asymmetry that are common in many wide-area wireless networks. For instance, in a cellular data network, mobile hosts experience a significantly larger and more variable media-access latency than the base station. In the multi-hop Ricochet network, the half-duplex nature of the radios makes it quite expensive to change the direction of packet flow frequently. In both these instances, the common case of data packets flowing towards mobile hosts and (equally frequent) acks flowing in the opposite direction poses problems. As we report in [5], decreasing the frequency of acks using AF or ACC alleviates the problems due to such asymmetry. It is interesting to note that the same techniques help in the context of both bandwidth and media-access asymmetry, albeit for different reasons.

## 9.6 Summary

In this chapter, we have discussed performance problems that arise in asymmetric access networks, which are increasingly being deployed to meet the demand for bandwidth fueled by the growth in popularity of the Internet, and in particular the Web. We have focussed on asymmetric-bandwidth networks, such as those based on cable modem technology. The performance problems arise primarily because congestion in the bandwidth-constrained direction could disrupt the ack stream and hence the performance of data transfer in the opposite direction. The presence of bidirectional traffic further compounds this problem.

We have developed a set of techniques to address these problems. These include ack congestion control and ack filtering to decrease the frequency of acks, sender adaptation to enable the sender to cope with disruptions in the ack stream, and ack-first scheduling to minimize disruptions in the ack stream in the presence of bidirectional traffic. Our performance results show that these techniques are very effective in combating the problems arising due to asymmetry. In some instances, performance improves by up to 15X.

With this chapter, we have concluded the main body of this thesis. In the next and final chapter, we summarize the work presented in this thesis, discuss our contributions, and give pointers to ways in which this work may be extended in the future.

# Chapter 10

# Conclusions and Future Work

We begin this chapter by recapitulating the challenges of Web data transport and the solutions we have developed. We then discuss the contributions of this thesis. We re-visit the design principles laid out in Section 1.8, and evaluate to what extent we have conformed to them. Finally, we point out directions for future work, both in addressing some of the shortcomings of this work and in exploring new avenues.

## 10.1 Challenges and Solutions

This work has been motivated by the rapid and continued growth of the World Wide Web, and the performance problems that have arisen because of a mismatch between the characteristics of the Web application and the capabilities of the TCP protocol, the *de facto* standard protocol for reliable unicast data transport in the Internet.

Two characteristics of the Web application make it quite different from traditional applications, such as bulk data transfer. First, Web page data is composed of multiple distinct components, each in itself of some value to the user. So it is desirable to transfer the components concurrently and independently to minimize user-perceived latency. Second, data transfer happens in relatively short bursts, with long intervening idle periods.

These characteristics of the Web application cause a mismatch with TCP. There are two problems. First, there is a tight coupling between TCP's ordered byte-stream service model, and its congestion control and loss recovery mechanisms. So mapping each Web transfer to a separate TCP connection, as in HTTP/1.0, is inefficient. Second, it is difficult to utilize network bandwidth effectively when a transfer is short in length.

The objective of this thesis has been to resolve this mismatch while minimizing changes to the existing Internet protocol architecture. Specifically, our goals were to:

1. Provide efficient support for multiple, concurrent data streams between a server and a client.
2. Enable effective utilization of network bandwidth even when data transfer is short in length.

In addition, the growth in popularity of the Web has led to a proliferation of access network technologies. A new and growing subset of these are asymmetric networks, such as those based on cable modem technology. Improvements in fiber-optic and switching technologies promise to make the core of the network extremely fast, so it is increasingly more likely that access networks will play a critical role in determining end-to-end performance. So our third goal was to:

3. Enable efficient data transport over a wide range of access network technologies, including those with asymmetric bandwidth, large bandwidth-delay product, and very limited bandwidth.

We have developed several techniques to meet these goals. These include *persistent-connection HTTP*, *TCP session*, *TCP fast start*, and *TCP for asymmetric networks*. We discuss these techniques in the context of general paradigms presented in the next section.

## 10.2 Contributions

The contributions of this thesis are a set of general paradigms to evolve unicast data transport from the current state-of-the-art, and specific realizations of these that address the goals listed above. We discuss these paradigms next.

### 10.2.1 Separation of Protocol Service Model from Protocol Mechanisms

The service that a network protocol provides and the mechanisms used to support it should not be coupled together tightly in a one-to-one correspondence. In the context of TCP, the ordered byte-service provided by a TCP connection should not require that the congestion control and loss recovery procedures needed to support this service model also operate within the confines of a connection.

Our initial attempt in this direction resulted in *Persistent-Connection HTTP* (*P-HTTP*). The key idea in P-HTTP is sharing a single TCP connection to support multiple logical data streams, each providing a reliable, ordered byte-stream service (just like a TCP connection). The key ideas in P-HTTP have been adopted by the HTTP/1.1 protocol. However, P-HTTP has the limitation that it imposes an ordering across the logical data streams. This drawback led us to an alternative approach, which we discuss next.

### 10.2.2 Shared State

The resources of the Internet are shared by many users, hosts and applications. To achieve fair and effective utilization of the resources, the communicating entities should share among themselves the information they learn about the availability of these resources. Such sharing of information should be explicit, not be just implicit (as it is in TCP).

Our specific realization of this paradigm, in the context of communication between a pair of hosts (such as a Web server and a client), is *TCP session*. By integrating congestion control and loss recovery across multiple TCP connections, TCP session decouples these activities from the individual connections and instead uses shared information for the benefit of all.

This paradigm fits in particularly well in the context of multicast data transport. A good example is the use of join-experiments in Receiver-driven Layered Multicast [70] to share information about the level of congestion in the network across multiple receivers.

### 10.2.3 Persistent State

While the availability of resources in a shared network varies with time, information learned in the recent past may be useful in estimating availability of resources in the near future. Indeed, several studies (e.g., [7], [88]) have concluded that the availability of resources, such as bandwidth, in the Internet tends to remain quite stable for significant lengths of time.

We realize this paradigm, both in the context of an individual TCP connection and that of a TCP session, via the *TCP fast start* algorithm. By caching and reusing information such as the congestion window size and the round trip time estimate, TCP fast start enables even short transfers to utilize the network bandwidth effectively, thereby decreasing latency.

There is however, the possibility of the information learned in the past being rendered stale by significant changes in the network conditions. This requires that precautions be taken, which TCP fast start does, when reusing information from the past. Among the mechanisms employed by TCP fast start for this purpose is priority dropping, which we discuss next.

### 10.2.4 Exploiting Support for Differentiated Services

When appropriate, protocols and/or applications on end-hosts should exploit support for service differentiation that the network may provide. In the context of the Internet, there is an effort underway [25] to define light-weight mechanisms to support service differentiation at the granularity of packets. Such mechanisms provide the end-host protocols and applications the means to enforce control on a finer time-scale than a round trip time (which is what standard TCP is limited to).

We realize this paradigm in the context of TCP fast start by exploiting a *priority dropping* mechanism. This gives the sender the freedom to reuse information about the availability of network resources that it learned in the past while limiting possible adverse consequences in case this information is stale.

### 10.2.5 Redefining the Coupling Between the Data and Control Paths

A data transport protocol can be thought of as having two components — the *data path*, which is concerned with the actual process of sending data, and the *control path*, which determines when, how much and which data gets sent. The coupling between the data and control paths need to be defined with care.

In TCP, the flow of acks represents the control path, and its coupling with the flow of data is defined by the ack clocking mechanism. While this mechanism has several advantages, there are situations (as the investigations in this thesis have revealed) where it leads to performance limitations. For instance, dependence on ack clocking forces a connection to go through slow start to get the clock started. *TCP fast start* loosens this coupling by substituting software timers for the ack clocks during the first RTT of a connection. Software timers have been used to clock out data in transport protocols, such as NETBLT [22], that use rate-based flow control.

Another instance where the tight coupling between the flow of acks and the flow of data hurts is in asymmetric networks. The high degree of asymmetry in network characteristics in opposite directions means that the flow of acks often does not reflect a true picture of the network characteristics in the data path, so it should not control the flow of data. We have developed several techniques, including a*ck congestion control*, *ack filtering*, *acks-first scheduling*, and *sender adaptation*, to loosen the coupling between the two, thereby improving performance substantially.

### 10.2.6 Summary

In summary, by exploiting the paradigms discussed above, we have greatly enhanced the performance and flexibility of unicast data transport within the framework of TCP. Applications are free to use as many TCP connections as they wish without running the risk of degraded performance. Even data transfers that are short in length are able to utilize the network resources effectively. Good performance is obtained over a wide variety of access networks, including asymmetric ones.

Our contributions also include mathematical and trace-based performance analysis, simulation, implementation, and performance evaluation of the ideas developed in this thesis.

## 10.3 Design Principles Re-visited

We re-visit the design principles laid out in Section 1.8 to determine the extent to which we have succeeded in conforming to them.

### 10.3.1 Robustness

We believe that our work has benefited from being constructed on the robust TCP/IP protocol framework. In places where we have deviated from this framework, we have included specific mechanisms to guarantee robustness. For instance, TCP fast start uses the priority dropping mechanism to avoid penalizing other traffic in the network in case the cached information is stale.

Another example is the use of software timers to smooth out the sending of data when acks are infrequent, as in asymmetric-bandwidth networks.

### 10.3.2 Wide Applicability

The techniques we have developed are not tied to specific applications. Also, in large part, they are not tied to specific networking technologies. We believe these make the techniques applicable in a wide variety of situations.

The one possible exception is the set of techniques developed in Chapter 9. While these have focussed on asymmetric-bandwidth networks, they are also useful in combating other forms of asymmetry, such as that arising due to media-access characteristics. Furthermore, some of the techniques, such as sender adaptation, find uses in other contexts, such as TCP fast start, as well. Other mechanisms, such as ack congestion control, are triggered automatically only when there is congestion along the ack path. They are essentially harmless in other situations.

### 10.3.3 End-to-End Principle

The techniques we have developed are confined to hosts in the periphery of the network. The two exceptions are ack filtering and priority dropping, both of which require router support. However, ack filtering is confined to the upstream router in an asymmetric-bandwidth network. Also, since such networks tend to be at the periphery of the Internet rather than at its the core, the overhead on the router tends to be quite low.

The requirement for priority dropping may impact a larger fraction of the routers in the Internet. However, the mechanism itself is very simple. It does not require any state to be maintained or any complex computations to be performed in the routers. It just requires that packets be classified based on a single bit in the packet header. We believe that the robustness that this simple mechanism helps provide in the context of TCP fast start is well-worth its cost. And as we discuss next, this mechanism is being deployed for other reasons anyway.

### 10.3.4 Incremental Deployment

The operation of a majority of the techniques that we have developed are confined to the data sender. These include TCP session and much of TCP fast start. Confining operation to the sender facilitates deployment in the context of the Web, because only Web servers need to be modified and the (larger number of) clients are left untouched. Also, our experiments show that performance often improves even if the techniques are deployed only on a subset of the servers.

The main stumbling block to incremental deployment of TCP fast start is its dependence on the priority dropping mechanism. While such a mechanism is being deployed as part of the differentiated services effort [25], support for it is far from ubiquitous at this time. However, in certain situations, it is possible to derive much of the benefit of fast start with only a partial deployment, for instance, at a Web proxy located at a satellite earth station or at the headend of a cable network plant.

### 10.3.5 Summary

In summary, we believe we have been quite successful in conforming to the design principles laid out in Section 1.8. The shortcomings are mainly due to the dependence on router mechanisms such as priority dropping. While this does impede deployment to an extent, we believe that it also opens the door to a new paradigm for transport protocols, namely, exploiting differentiated services mechanisms in routers.

## 10.4 Directions for Future Work

We now discuss several directions for future work, both in addressing the limitations of this work and in exploring new avenues.

### 10.4.1 Sharing State Across Hosts

While TCP session shares state across connections between a pair of hosts, there are situations where it may be advantageous to extend such sharing across multiple hosts. For instance, client hosts on a local area network that are communicating with the same server may benefit from sharing information about the availability of network resources. Another example is a single client communicating with multiple server nodes in a cluster. There are several new issues that need to be addressed in this context — when is it appropriate to share state and on what time-scales should such sharing be done. The SPAND system [97] is an instance where such sharing is being investigated, albeit at the level of applications than the data transport protocol.

### 10.4.2 Comprehensive Use of Persistent State

TCP fast start adopts a very simple model for persistent state. To first order, it only reuses the most recent values of the congestion window size[1] and the round trip time estimate. It may be possible to improve the chances that a fast start attempt succeeds by making comprehensive use of information from the past. For instance, the evolution of the congestion window size over several round trip times in the past, suitably filtered, could be used to determine an appropriate setting for the congestion window. Another possibility, suggested by Mogul [75], is to consider the history of packet losses to decide whether to be aggressive or conservative in using persistent state.

### 10.4.3 Guarding Against Malicious Users

A shared network such as the Internet is vulnerable to malicious users who may appropriate an unfair share of the network resources by being aggressive in their use of the network. There have been proposals such as [32] to incorporate mechanisms in routers to detect and penalize such behavior. While not altering the situation fundamentally, the techniques that we have developed in this thesis pose new challenges to such policing. For instance, with TCP session, bundles of connections would need to be policed as a whole rather than each connection separately. The use of priority dropping as a protocol mechanism raises the question of whether different yardsticks should be used to police different classes of packets.

### 10.4.4 Improved Metrics for Quantifying Perceived Performance

Traditional network performance metrics such as throughput and latency, while useful, do not accurately capture the performance *perceived* by a human user. This is particularly so with multimedia data, as in the Web. Even a partially downloaded Web page may be useful from a user's

1. This is not strictly true. As discussed in Section 8.2.2.2, the old congestion window size may be modified before reuse.

viewpoint. And, for instance with progressively coded images, having the base layer of two images may be more valuable to a user than the having entire contents of one image and none of the other. So the figure of merit may well be how quickly the perceptually-important data is downloaded, not necessarily all of the data[2]. Quantifying perceived performance accurately is a difficult task. It requires accounting for several factors, including the perceived value of a partially downloaded object (the signal-to-noise ratio may serve as a crude measure in the case of images), the distinction between information that is visible to the user versus that which remains hidden, etc. There has been some initial effort in this direction, for instance, in [39] and [43].

### 10.4.5 Multicast-Based Dissemination of Web Data

There have been quite a few proposals for multicast-based dissemination of Web data (e.g., [104], [109]). The basic idea is to use multicast to avoid transmitting a given piece of Web data repeatedly through the network. While this is certainly desirable, there are several impediments to the general use of multicast in the context of the Web. These include the asynchronous nature of Web accesses, the updating of data at unpredictable times, and the limited overlap between accesses by different users (as evident from the relatively low hit rates of Web caches). Therefore, the interesting question is how multicast could be used to complement rather than replace unicast in a comprehensive architecture for disseminating Web data. One example is to dissipate hot spots caused by *flash crowds*, by marrying prefetching [87] with multicast-based data dissemination, as in the LSAM system [104].

## 10.5 Availability

The software we have developed and papers describing various parts of this work are available from `http://www.cs.berkeley.edu/~padmanab/phd-thesis.html`. The software includes our implementations of TCP session, TCP fast start, and the various techniques for alleviating the problems due to asymmetry. All of these implementation are part of the BSD/OS 3.0 TCP/IP stack. In addition, our user-level implementation of P-HTTP as part of the Mosaic and httpd programs from NCSA is also available.

2. As an aside, we note that connection scheduling in TCP session is well-suited to supporting a policy that, for instance, requires a larger fraction of the available bandwidth to be devoted to perceptually-important data.

# Appendix A

# Performance Analysis Tools

In this appendix, we describe briefly the software tools that we used for our performance analysis. These include existing tools as well as new ones that we developed.

***sock*:** The *sock* program [101] allows us to create sources and sinks of data that communicate with each other using the TCP and/or UDP protocols. The original *sock* program only allowed sending data in one continuous stream. We have enhanced this program to emulate Web traffic with its short bursts and relatively long pauses. We also added hooks to selectively enable the various performance optimization techniques that we develop in later chapters, which allow us to analyze the benefits of each technique in isolation.

***tcpdump:*** The *tcpdump* program [69,68] is packet capture tool which passively captures a subset of or all packets on a network link. This is very useful for analyzing the behavior of protocols such as TCP. We use *tcpdump* in our experiments to capture packet traces at the sender, receiver, and/or an intermediate point (such as a Web proxy). We wrote a set of scripts to process the *tcpdump* packet traces, extract data and ack sequence number traces for the individual connections, and identify interesting events such as packet loss, reordering, duplicate acknowledgement and retransmission.

***tcpstats:*** We developed a trace tool called *tcpstats* to monitor the evolution of the TCP protocol state. This state includes the congestion window size, the slow start threshold, the retransmission timer setting, etc. Together with the sequence number traces obtained using *tcpdump,* this information enables us to construct a comprehensive picture of the functioning of the protocol in action.

The variables of interest are traced by instrumenting the TCP/IP protocol stack in BSD/OS 3.0 [16] with trace points installed where the variables are modified. To minimize the impact of our instrumentation on the protocol execution, all tracing activity is confined to the kernel, with the traced values (and the corresponding timestamps) stored in an in-kernel buffer. Only after the experiment has concluded are the contents of the trace buffer collected and saved in a file by a user-level program.

***linkemu:*** The *linkemu* tool we developed emulates a wide range of network link characteristics in software (much as tools like *dummynet* [94]). With this tool, we can carefully control a number of link parameters including bandwidth, delay, buffer size, and a variety of scheduling and buffer management schemes (e.g., strict priority scheduling and priority dropping of packets). Using *linkemu* we can experiment with the real TCP/IP protocol code over a variety of emulated network links, ranging from dialup phone lines to geostationary satellite links, and yet have complete control over the traffic that traverses these links.

We implemented *linkemu* in the device-independent part of the BSD/OS driver for Ethernet-class devices (such as 10/100 Base-T Ethernet, WaveLAN, etc.). Our implementation adds two stages of queuing in addition to the existing interface queue. The first is used for bandwidth emulation, i.e., to hold packets for a length of time corresponding to their transmission time on the emulated link. This is also where the scheduling and buffer management policies are implemented. The second stage is a callout queue used to emulate propagation delay.

Note that the bandwidth of the underlying physical channel used for emulation (e.g., Ethernet) imposes a limit on the bandwidth of the emulated link.

# Appendix B

# Implementing TCP Session

## B.1 Introduction

Of the three components of TCP session — integrated congestion control, integrated loss recovery, and connection scheduling — integrated loss recovery is the most challenging to implement. So we focus on integrated loss recovery, with brief discussions of the other components where necessary.

### B.1.1 Implementing the Loss Recovery Rules

The main challenge in implementing the loss recovery rules is determining whether an ack is a later ack for one or more segments, and keeping an updated count of later acks for each unacknowledged segment. For this purpose, we define a structure, *segment*, containing the following information for each unacknowledged segment:

1. Start and end sequence numbers (*startseq* and *endseq*).

2. Number of duplicate acks and later acks, both initialized to 0 (*dupacks* and *lateracks*).

3. Whether or not this corresponds to a retransmission (*retransflag*).

4. Pointers to the corresponding TCB and SCB (*tcb* and *scb*).

These structures, corresponding to each data segment, are linked together in a list using:

5. Pointer to the next segment in the list (*nextseg*).

The SCB contains a pointer, *seglist*, to this list. In the discussion that follows, we use *segment* (in italics) to denote the list element corresponding to a TCP data segment.

Each time a data segment is transmitted (either a fresh transmission or a retransmission), a corresponding *segment* is appended to the tail of the list. This ensures that the list is automatically maintained sorted by transmission time.

Much of the processing happens at the time an ack is received. We consider several cases for the received ack:

***Case #1:*** *Ack for new data with no reordering across connections in the TCP session*

By "no reordering", we mean that the ack does not acknowledge any data segment that was sent more recently than the segments on other connections (if any) that have not yet been acknowledged.

Upon receipt of the ack, the sender identifies the corresponding TCB and SCB. Using the information contained in the TCB and in the ack just received, it computes *new_data_acked*, the amount of previously unacknowledged data that has now been acknowledged by this ack. It then starts scanning the *seglist* pointed to by the SCB, starting at the head. Each *segment* (belonging to the same connection as the ack) is examined to see if it has been acknowledged by this most recent ack. If yes, then the *segment* is removed from the *seglist*. It may be that the ack only acknowledges part of a segment. In such a case, the *startseq* field of the *segment* is updated to reflect the latest ack information, but the *segment* is retained in the *seglist*. In either case, a running count of the amount of acknowledged data accounted for (and removed from the *seglist*) is kept. The *session_ownd* is decremented by this amount to reflect the reduction in the amount of data that is still outstanding.

The sender continues to scan the *seglist* until the amount of acknowledged data accounted for equals *new_data_acked*. This would happen precisely when the sender has just scanned the *segment* corresponding the receiver's cumulative ack. There is no need to continue scanning because *segments* beyond this point correspond to data segments sent only after the one that triggered the ack. Since we assume that there is no reordering, the ack just received does not convey any information about these *segments*, so the scanning procedure is terminated at this point.

The amount of work to be done by the sender in this case (presumably the common case) is quite limited. At the time a segment is sent out, a small, constant number of operations are needed to append the corresponding *segment* to the tail of the *seglist*. At the time an ack arrives, the *seglist* needs to be scanned beginning at the head. However, since there is no reordering in this case, the scanning is confined to a small region close to the head of the list.

***Case #2:*** *Ack for new data, but with reordering across connections in the TCP session*

Reordering of acks across connections refers to the situation where the sender gets back an ack for a data segment on a connection sooner than one for another segment sent prior to it on a different connection. This can happen either because of reordering of packets in the network or because of the delayed ack algorithm.

In either case, the processing done by the sender is much as in case #1. The sender scans the *seglist,* removing/updating *segments* to reflect the newly acknowledged data, until *new_data_acked* amount of data has been accounted for. In addition, however, each time a *segment* of a different connection is encountered, its *lateracks* count is incremented by 1. The chronological ordering of the *seglist* coupled with the scanning procedure ensures that such (unacknowledged) *segments* must indeed have been sent prior to the one(s) that just got acknowledged.

At the time the *lateracks* count for a *segment* is incremented, the sender checks to see if the *segment* is now eligible for retransmission according to the rules listed in Section 6.3.4.1. If it is, the *segment* is added to a separate list, *retranslist*, corresponding to data segments that are candidates for retransmission. The retransmission procedure is discussed in Section B.1.2.

Rule #3 from Section 6.3.4.1 requires checking whether at least two *segments* of the same connection have each received the threshold number of duplicate/later acks. However, such a determination can only be made once the rest of the *seglist* has been scanned. To avoid scanning the *seglist* a second time, a *segment* that satisfies all the clauses of rule #3 with the possible exception of this one (i.e., a *segment* that is the first unacknowledged one in its connection and in itself has the threshold number of duplicate/later acks) is tentatively added to the *retranslist*. At the time the retransmission procedure is invoked, a check is done to see if the segment indeed satisfies *all* of the clauses in rule #3. Only if this is so is the corresponding data segment retransmitted.

The amount of work to be done at the time a new data segment is transmitted is the same as in case #1. However, when an ack is received, more of the *seglist* may need to be scanned. The amount to be scanned depends on the extent of reordering. Studies such as [88] have shown that the extent of packet reordering (if at all any) tends to be small, so there is reason to believe that not much of the *seglist* would need to be scanned. Furthermore, this case is presumably less common than case #1.

***Case #3:*** *Duplicate ack*

A duplicate ack[1] does not acknowledge any new data, so *new_data_acked* is set to 0. The *seglist* is scanned until the *segment* corresponding to the duplicate ack is reached. If this *segment* does not correspond to a retransmission (i.e., its *retransflag* is false), its *dupacks* count is incremented by 1. (We discuss the case where *retransflag* is true in Section B.1.2). The *lateracks* count of each *segment* (belonging to other connections) ahead of it in the *seglist* is also incremented by 1. As in case #2, if any segment becomes eligible for retransmission, it is added to the *retranslist*.

For each duplicate ack received, *session_ownd* is decremented by one MSS (maximum segment size) if the corresponding segment's *retransflag* is true. This emulates the window inflation that happens during fast recovery with standard TCP Reno.

Although the *segment* corresponding to the duplicate ack could in general be located at any place in the *seglist*, it would tend to be close to the head (especially if other connections continue to make progress, causing the corresponding *segments* to be removed from the list). So the portion of the *seglist* to be scanned would tend to be small. As we discuss in Section B.1.2, the only exception is when a duplicate ack is received for a data segment that has just been retransmitted, since in such a case, the corresponding *segment* may be close to the tail of the *seglist*.

1. Note that just as in standard TCP, a window update packet is not considered to be a duplicate ack.

### B.1.2 Retransmitting Segments

After the *seglist* has been processed as discussed above, the sender scans the *retranslist* to determine if there are any data segments that should be retransmitted. As discussed in Section B.1.1, certain *segments* may been added to the *retranslist* on a tentative basis pending full compliance with rule #3. Therefore, as it scans the *retranslist*, the sender checks again to make sure that a *segment* is indeed eligible for retransmission. Since we only assume cumulative ack information from the receiver (and no selective acks), there can be no more than one *segment* eligible for retransmission in each connection.

Each eligible data segment is then retransmitted. The corresponding *segment* is removed from the *seglist*, and a new one is inserted at the tail. This helps maintain the chronological ordering of the *seglist*. The *retransflag* of the new *segment* is set to true and its *dupacks* and *lateracks* counts are set to 0. If the *dupacks* count of the original *segment* was non-zero, *session_ownd* is decremented by the corresponding multiple of MSS.

The sender also initiates congestion control action. It halves *session_cwnd* and sets *session_ssthresh* to this new value of *session_cwnd*. However, as in TCP NewReno, it is desirable to avoid initiating congestion control action repeatedly for multiple losses in a loss window. For this purpose, the sender stores a pointer, *losswin*, to the tail of the *seglist* at the time congestion control action is initiated. This information is used and updated as follows:

- Congestion control action is initiated only if the *segment* being retransmitted lies beyond[1] the *losswin* recorded the last time.
- The *losswin* is updated only if the sequence number of the segment being retransmitted lies beyond the current value of *losswin*.

Figure B.1 illustrates the importance of these steps. It depicts a TCP session with three connections. The three connections suffer a packet loss each, in turn. Figure B.1(a) shows the correct manner in which *losswin* is updated and congestion control action initiated. When *losswin* is not used at all, Figure B.1(b) shows how congestion control action could incorrectly be initiated multiple times for losses within a single loss window. Finally, Figure B.1(c) shows the danger of updat-

1. By "beyond the *losswin*", we mean being closer to the tail of the *seglist* than *losswin*.

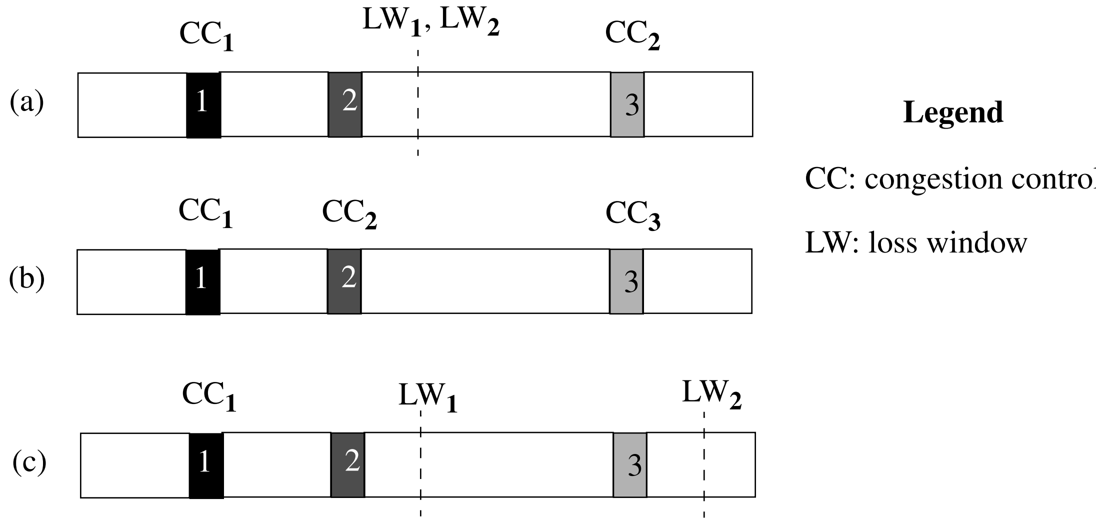



**Figure B.1 An illustration of congestion control and loss window dynamics in the aftermath of three packet losses, one on each of three connections in a TCP session. (a) shows the correct set of actions — congestion control is initiated in response to loss #1 and #3, and the loss window is not updated at the time of loss #2. (b) shows the incorrect invocation congestion control in response to loss #2 because the sender does not keep track of the loss window. (c) shows the loss window being incorrectly updated in response to loss #2, causing congestion control action not to be invoked in response to loss #3.**

ing *losswin* upon every retransmission, regardless of whether or not the segment being retransmitted lies beyond the old *losswin*. In such a case, congestion control action may not be initiated even when warranted.

Finally, as in TCP NewReno, a segment may become eligible for retransmission if a partial new ack is received for it. In this case, the loss recovery rules of Section 6.3.4.1 are bypassed.

### B.1.3 Retransmission Timeout

A retransmission timeout is the last resort for loss recovery. A single retransmission timer is set whenever there is outstanding data on any of the connections constituting the TCP session. As in TCP, the timer is set to *srtt + 4*rttvar*. However, since *srtt* and *rttvar* are estimated by pooling together RTT samples obtained on all of the connections in the session, these are likely to be more accurate than the corresponding estimates individual connections.

Note that with TCP session, a timeout will occur only if the ack stream has dried up on *all* the connections. This coupled with the sharing of loss recovery information across connections (by way of integrated loss recovery) implies that a timeout is much less likely than with standard TCP.

Nevertheless, when a timeout does occur, *session_cwnd* is reset to 1 segment and *session_ssthresh* is set to half of the old value of *session_cwnd*. In addition, the entire *seglist* is deleted and *session_ownd* is reset to 0. Thus, the sender starts all over again with a clean slate. The first unacknowledged segment in each connection is eligible for retransmission. However, since *session_cwnd* has been reset to 1 segment, only one of the connections will be able to retransmit at the beginning. Further retransmissions happen in accordance with the connection scheduling policy.

# Bibliography

[1] The Apache Web Server Project. http://www.apache.org/.

[2] R. Atkinson. *Security Architecture for the Internet Protocol*. Naval Research Laboratory, Aug 1995. RFC-1825.

[3] S. Bajaj, L. Breslau, and S. Shenker. Uniform versus Priority Dropping for Layered Video. In *Proc ACM SIGCOMM '98*, September 1998.

[4] H. Balakrishnan. *Challenges to Reliable Data Transport over Heterogeneous Wireless Networks*. PhD thesis, University of California at Berkeley, August 1998.

[5] H. Balakrishnan, V. N. Padmanabhan, and R.H. Katz. The Effects of Asymmetry on TCP Performance. In *Proc. ACM/IEEE MOBICOM Conf.*, September 1997.

[6] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R.H. Katz. TCP Behavior of a Busy Web Server: Analysis and Improvements. In *Proc. IEEE INFOCOM*, March 1998.

[7] H. Balakrishnan, S. Seshan, M. Stemm, and R.H. Katz. Analyzing Stability in Wide-Area Network Performance. In *Proc. ACM SIGMETRICS '97*, June 1997.

[8] Masinter L. Berners-Lee, T. and M. McCahill. *Uniform Resource Locators (URL)*. RFC, Dec 1994. RFC-1738.

[9] T. Berners-Lee. *Hypertext Markup Language - 2.0*. MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, Nov 1995. RFC-1866.

[10] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. RFC, May 1996. RFC-1945.

[11] N. Borenstein and N. Freed. *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, Sep 1993. RFC 1521.

[12] R. T. Braden. *Extending TCP for Transactions - Concepts*. RFC, Nov 1992. RFC-1379.

[13] R. T. Braden. *T/TCP – TCP Extensions for Transactions Functional Specification*. RFC, July 1994. RFC-1644.

[14] L. Brakmo. Personal communication, Apr 1998.

[15] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proc. ACM SIGCOMM '94*, August 1994.

[16] Berkeley Software Design, Inc. http://www.bsdi.com.

[17] Cellular Digital Packet Data System Specification: Release 1.0, CDPD, P.O. Box 97060, Kirkland, WA 98083, USA, 1993.

[18] Common Gateway Interface. http://www.w3.org/CGI/.

[19] Cisco Cache Engine. http://cco.cisco.com/warp/public/751/cache/cds_wp.htm.

[20] D. Clark, M. L. Lambert, and L. Zhang. NETBLT: A High Throughput Transport Protocol. In *Proc. ACM SIGCOMM*, August 1988.

[21] D. Clark and D. Tennenhouse. Architectural Consideration for a New Generation of Protocols. In *Proc. ACM SIGCOMM*, September 1990.

[22] D. D. Clark, M. L. Lambert, and L. Zhang. NETBLT: A High Throughput Transport Protocol. In *Proc. ACM SIGCOMM '87*, August 1987.

[23] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulations of a Fair-Queueing Algorithm. *Internetworking: Research and Experience*, V(17):3–26, 1990.

[24] Dynamic HTML Information Society. http://www.dhtml.org/.

[25] Differential Services for the Internet. http://diffserv.lcs.mit.edu.

[26] Digital Equipment Corporation. http://www.digital.com.

[27] DirecPC Web page. http://www.direcpc.com.

[28] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and Sack TCP. *Computer Communications Review*, July 1996.

[29] W. Feng, D. Kandlur, D. Saha, and K. Shin. Understanding TCP Dynamics in an Integrated Services Internet. In *NOSSDAV '97*, May 1997.

[30] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, Jan 1997. RFC-2068.

[31] S. Floyd. TCP and Explicit Congestion Notification. *Computer Communications Review*, 24(5), October 1994.

[32] S. Floyd and K. Fall. Router Mechanisms to Support End-to-End Congestion Control. Technical report, February 1997.

[33] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4), August 1993.

[34] S. Floyd and V. Jacobson. Link-sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, August 1995.

[35] J. Gettys and A. Freier. HTTP Connection Management. Internet Draft, IETF, March 1997. Expires Sep 1997.

[36] J. Gettys and H. F. Nielsen. WebMUX Protocol Specification. Internet Draft, IETF, August 1998. Expires Jan 1999.

[37] Graphics Interchange Format. http://members.aol.com/royalef/gif87a.txt, 1987.

[38] Graphics Interchange Format, Version 89a. http://members.aol.com/royalef/gif89a.txt, 1989.

[39] J. M. Gilbert and R. W Brodersen. Globally Progressive Interactive Web Delivery, 1998. Submitted for publication.

[40] M. Gray. Growth and Usage of the Web and the Internet, 1997. http://www.mit.edu/people/mkgray/net/.

[41] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proc. 1997 Usenix Symposium on Internet Technologies and Systems*, December 1997.

[42] Global System for Mobile communications. http://www.gsm.org.

[43] R. Gupta. WebTP: A User-Centric Web Transport Protocol , May 1998.

[44] J. Heidemann. Performance Interactions Between P-HTTP and TCP Implementations. *Computer Communications Review*, April 1997.

[45] J. C. Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, Massachusetts Institute of Technology, 1995.

[46] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *Proc. ACM SIGCOMM '96*, August 1996.

[47] Hybrid Networks, Inc. http://www.hybrid.com.

[48] IBM TCP Router. http://www.research.ibm.com/webvideo/tcproute.html.

[49] IBM Web Object Manager. http://www.womplex.ibm.com.

[50] IBM Olympics Web Site. http://www.olympic.ibm.com.

[51] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM 88*, August 1988.

[52] V. Jacobson. *Compressing TCP/IP Headers for Low-speed Serial Links*. RFC-1144, Feb 1990.

[53] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High Performance*, May 1992. RFC-1323.

[54] V. Jacobson and M. Karels. Congestion Avoidance and Control. Revised version of SIGCOMM '88 paper, 1992.

[55] R. Jain. A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. *ACM SIGCOMM Computer Communication Review*, October 1989.

[56] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, 1991.

[57] Java Technology Home Page. http://www.javasoft.com/.

[58] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan. Performance of Two-Way TCP Traffic over Asymmetric Access Links. In *Proc. Interop '97 Engineers' Conference*, May 1997.

[59] C. R. Kalmanek, H. Kanakia, and S. Keshav. Rate-Controlled Servers for Very High-Speed Networks. In *Proc. Globecom '90*, December 1990.

[60] P. Karn. Dropping TCP acks. Mail to the end-to-end mailing list, February 1996.

[61] S. Keshav. A Control-Theoretic Approach to Flow Control. In *Proc. ACM SIGCOMM '91*, 1991.

[62] D. Kristol and L. Montulli. *HTTP State Management Mechanism*. RFC, Feb 1997. RFC-2109.

[63] T. V. Lakshman, U. Madhow, and B. Suter. Window-based Error Recovery and Flow Control with a Slow Acknowledgement Channel: A Study of TCP/IP Performance, 1996. Preprint.

[64] T. V. Lakshman, U. Madhow, and B. Suter. Window-based Error Recovery and Flow Control with a Slow Acknowledgement Channel: A study of TCP/IP Performance. In *Proc. Infocom 97*, April 1997.

[65] B. A. Mah. An Empirical Model of HTTP Network Traffic. In *Proc. InfoComm '97*, April 1997.

[66] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*, 1996. RFC-2018.

[67] Mathis, M. and Mahdavi, J. and Floyd, S. and Romanow, A. *TCP Selective Acknowledgment Options*, 1996. RFC-2018.

[68] S. McCanne and V. Jacobson. Tcpdump Software. ftp://ftp.ee.lbl.gov/tcpdump.tar.Z.

[69] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proc. Winter '93 USENIX Conference*, San Diego, CA, January 1993.

[70] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven Layered Multicast. In *Proc ACM SIGCOMM '96*, August 1996.

[71] Metricom, Inc. http://www.metricom.com.

[72] Microsoft Corporation. http://www.microsoft.com/.

[73] J. C. Mogul. Observing TCP Dynamics in Real Networks. Technical Report 92/2, Digital Western Research Lab, April 1992.

[74] J. C. Mogul. The Case for Persistent-Connection HTTP. In *Proc. SIGCOMM '95,* August 1995.

[75] J. C. Mogul. Personal communication, Sep 1998.

[76] J. C. Mogul and S. E. Deering. *Path MTU Discovery*, April 1990. RFC 1191.

[77] The National Center for Supercomputing Applications. http://www.ncsa.uiuc.edu/.

[78] Netscape Communications Corporation. http://www.netscape.com/.

[79] Internet Domain Survey. Network Wizards, http://www.nw.com, 1998.

[80] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prudhommeaux, H. W. Lie, and C. Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *Proc. ACM SIGCOMM '97*, September 1997.

[81] UCB/LBNL/VINT Network Simulator - ns (version 2). http://www-mash.cs.berkeley.edu/ns/.

[82] V. N. Padmanabhan. Receiver-oriented Data Transport with Persistent Sessions. Qualifying Examination Proposal, http://www.cs.berkeley.edu/~padmanab/research/quals-proposal.ps, October 1996.

[83] V. N. Padmanabhan, H. Balakrishnan, K. Sklower, E. Amir, and R. Katz. Networking Using Direct Broadcast Satellite. In *Proc. Workshop on Satellite-Based Information Systems*, November 1996.

[84] V. N. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique For Speeding Up Web Transfers. In *Proc. Globecom Internet Mini-Conference*, November 1998.

[85] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proc. Second International WWW Conference*, October 1994.

[86] V. N. Padmanabhan and J. C. Mogul. P-HTTP Software. ftp://ftp.digital.com/pub/DEC/net/phttp.tar.Z, April 1995.

[87] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *ACM SIGCOMM Computer Communication Review*, 26(3), July 1996.

[88] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California at Berkeley, May 1997.

[89] J. B. Postel. *Transmission Control Protocol*. Information Sciences Institute, Marina del Rey, CA, September 1981. RFC-793.

[90] J. B. Postel and J. Reynolds. *TELNET Protocol Specification*. Information Sciences Institute, Marina del Rey, CA, May 1983. RFC-854.

[91] J. B. Postel and J. Reynolds. *File Transfer Protocol (FTP)*. Information Sciences Institute, Marina del Rey, CA, Oct 1985. RFC-821.

[92] K. K. Ramakrishnan and R. Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks. *ACM Transactions on Computer Systems*, 8(2):158–181, May 1990.

[93] S. Raman. Personal communication, March 1998. Shared whiteboard using unicast.

[94] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *Computer Communications Review*, January 1997.

[95] J. L. Romkey. *A Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP*. RFC, June 1988. RFC-1055.

[96] J. Saltzer, D. Reed, and D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2:277–288, Nov 1984.

[97] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared Passive Network Performance Discovery. *Proc. USENIX Symposium on Internet Technologies and Systems*, December 1997.

[98] K. Sollins and L. Masinter. *Functional Requirements for Uniform Resource Names*. RFC, Dec 1994. RFC-1737.

[99] SPECWeb96 Benchmark. http://www.specbench.org/osg/web96.

[100] S. Spero. Session control protocol (scp). http://www.w3.org/pub/WWW/Protocols/HTTP-NG/http-ng-scp.html, 1996.

[101] W. R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, Reading, MA, Nov 1994.

[102] K. Thompson, G. J. Miller, and R. Wilder. Wide-area Internet Traffic Patterns and Characteristics. *IEEE Network*, 11(6), November/December 1997.

[103] J. Touch. *TCP Control Block Interdependence*. RFC, April 1997. RFC-2140.

[104] J. Touch. The LSAM Proxy Cache - a Multicast Distributed Virtual Cache. In *Proc. 3rd International WWW Caching Workshop*, Jun 1998.

[105] Vikram Visweswaraiah and John Heidemann. Improving restart of idle TCP connections. Technical Report 97-661, University of Southern California, November 1997.

[106] D. Wetherall and C. Lindblad. Extending Tcl for Dynamic Object-Oriented Programming. In *Proc. Tcl/Tk Workshop*, July 1995.

[107] A. Woodruff et al. An Investigation of Documents from the World Wide Web. In *Proc. Fifth International World Wide Web Conference*, May 1996.

[108] L. Zhang. Why TCP Timers Don't Work Well. In *Proc. ACM SIGCOMM*, pages 397–405, August 1986.

[109] L. Zhang, S. Michel, K. Nguyen, A. Rosenstein, S. Floyd, and V. Jacobson. Adaptive Web Caching: Towards a New Global Caching Architecture . In *Proc. 3rd International WWW Caching Workshop*, Jun 1998.

[110] L. Zhang, S. Shenker, and D. D. Clark. Observations and Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proc. ACM SIGCOMM '91*, pages 133–147, 1991.

[111] H. Zimmerman. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.