BOSTON UNIVERSITY
GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

MODELING, MEASUREMENT AND PERFORMANCE OF WORLD WIDE WEB TRANSACTIONS

by

PAUL ROBERT BARFORD

B.S., University of Illinois, Urbana-Champaign, 1985

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
2001

Approved by

First Reader

Mark Crovella, Ph.D.
Associate Professor of Computer Science

Second Reader

John Byers, Ph.D.
Assistant Professor of Computer Science

Third Reader

Azer Bestavros, Ph.D.
Associate Professor of Computer Science

To Carol - the bright light in my world

Acknowledgments

I met Mark Crovella in the spring of 1995. That first meeting convinced me that the adventure of graduate school was something I wanted to pursue. Mark taught me many things during the five and a half years that I worked with him; the most important being how to write, how to do scientific research and how to be a professor. Mark is my mentor, collaborator and friend and I thank him from the bottom of my heart for his constant support throughout my program.

The success of my work is based on the ideas and direction of the OCEANS research group at Boston University. The founding members of that group included Mark Crovella, Azer Bestavros, Abdelsalam Heddaya, Robert Carter, Sulaiman Mirdad and Carlos Cunha. Their work in Web client analysis and Web caching was what got me started and I thank them for paving the way.

Special thanks go to Vern Paxson and Walter Willinger. Vern's NPD project and his various papers were some of the principal inspiration for the WAWM project and his thesis was and continues to be a guide for me. Vern has been a wonderful advisor in my projects and his insight and guidance have been invaluable in my research. Walter's self-similarity paper was a similar inspiration to me and has also been a significant guide in my work. It always amazed me that someone as smart as Walter would put up with someone with a million crazy ideas like me but for some reason he has. Walter is also a mentor and I deeply appreciate his support.

There have been a number of researchers outside of Boston University whose input and support have helped me in many ways. They include Keith Lowery, Anja Feldmann, Erich Nahum, Jim Pitkow, Kihong Park, Carey Williamson, Martin Arlitt, Virgilio Almeida, Michael Mitzenmacher and Sally Floyd. Each of these folks has either helped me with a project or advised me along the way and I am lucky to count them all as friends. To all of them go my sincere thanks.

There have been a number of folks at Boston University who have helped and/or collaborated with me on the technical aspects of my projects to whom I want to give special thanks. They include Lars Kellogg-Stegman who played a major role in getting the WAWM systems up and running, Adam Bradley who was always ready to answer a question a Linux question for me, David Martin who still deals with my questions, Azer Bestavros who was always a great sounding board for ideas and John Byers who has been wonderful to work with and who I know I will work with much more in the future. Special thanks also to Ellen Grady, our wonderful department administrator, who had to listen to me ask, "is my check in yet?" way more than she deserved.

As anyone who has been around the Boston University Computer Science Department knows, we like to have fun. The "coffee break posse" has been fantastic and I will miss them: Bob Carter, Sue Nagy, David Martin, Alia Atlas, Bob Frangioso, Drue Coles, Harry Marison, Azer Bestavros and John Byers. The experience would not have been nearly as enjoyable without you folks.

Many thanks go to officemate, Drue Coles. He has had to put up with five long years of me and I wouldn't wish that on anyone. Drue helped me on many many occasions and is one of the smartest and hardest working people I know. He is my brother.

Finally, I thank my family for their love and support throughout this entire process. This career choice of mine is certainly following in the footsteps of mother Judy and father Bob; two of the most incredible people in the world. My brothers and sister are my constant inspiration; I am so proud of all of you. My son Henry is such a bright joy for me and always bring me back to what is important. My wife Carol is the strongest, most talented most patient and, loving person in the world. It was the luckiest day of my life when I fell in love with you darling. Thank you all from the bottom of my heart.

MODELING, MEASUREMENT AND PERFORMANCE OF WORLD WIDE WEB TRANSACTIONS

(Order No.)

PAUL ROBERT BARFORD

Boston University Graduate School of Arts and Sciences, 2001

Major Professor: Mark Crovella, Associate Professor of Computer Science

Abstract

The size, diversity and continued growth of the World Wide Web combine to make its understanding difficult even at the most basic levels. The focus of our work is in developing novel methods for measuring and analyzing the Web which lead to a deeper understanding of its performance. We describe a *methodology* and a distributed *infrastructure* for taking measurements in both the network and end-hosts. The first unique characteristic of the infrastructure is our ability to generate requests at our Web server which closely imitate actual users. This ability is based on detailed analysis of Web client behavior and the creation of the Scalable URL Request Generator (SURGE) tool. SURGE provides us with the flexibility to test different aspects of Web performance. We demonstrate this flexibility in an evaluation of the 1.0 and 1.1 versions of the Hyper Text Transfer Protocol.

The second unique aspect of our approach is that we analyze the details of Web transactions by applying *critical path analysis* (CPA). CPA enables us to precisely decompose latency in Web transactions into propagation delay, network variation, server delay, client delay and packet loss delays.

We present analysis of performance data collected in our infrastructure. Our results show that our methods can expose surprising behavior in Web servers, and can yield considerable insight into the causes of delay variability in Web transactions.

Contents

List of Tables

List of Figures

List of Abbreviations

ACK:	....	Acknowledgement
BSD:	....	Berkeley Standard Distribution
BPF:	....	Berkeley Packet Filter
CDF:	....	Cumulative Distribution Function
CPA:	....	Critical Path Analysis
CPU:	....	Central Processing Unit
DNS:	....	Domain Name Server
ECN:	....	Explicit Congestion Notification
GPS:	....	Global Positioning System
GUI:	....	Graphical User Interface
HTTP:	....	Hyper Text Transport Protocol
IETF:	....	Internet Engineering Task Force
IIS:	....	Internet Information Server
IP:	....	Internet Protocol
IPPM:	....	Internet Protocol Performance Metrics
ISP:	....	Internet Service Provider
LAN:	....	Local Area Network
MAN:	....	Metropolitan Area Network
NIC:	....	Network Interface Card
NIMI:	....	National Internet Measurement Infrastructure
NIST:	....	National Insititute of Standards and Technology
NLANR:	....	National Laboratory for Applied Network Research
NPD:	....	Network Probe Daemon
NTP:	....	Network Time Protocol
PDG:	....	Packet Dependance Graph
PPS:	....	Pulse Per Second
RAM:	....	Random Access Memory
RED:	....	Random Early Detection
RFC:	....	Request For Comments
RPC:	....	Remote Procedure Call
RTT:	....	Round Trip Time

SACK:	....	Selective Acknowlegements
SURGE:	....	Scalable URL Reference Generator
TCP:	....	Transport Control Protocol
UE:	....	User Equivalent
UDP:	....	User Datagram Protocol
URL:	....	Universal Resource Locator
WAN:	....	Wide Area Network
WAWM:	....	Wide Area Web Measurement

Chapter 1

Introduction

1.1 Motivation and Approach

Over the past ten years, the Internet has become an immense, world-wide infrastructure rivaling highway systems, power grids and the telephone network in terms of size and complexity. Two important catalysts to the Internet's growth were the introduction of the World Wide Web in 1990, and the Mosaic Web browser in 1993 which brought together text, graphics and other media into a single application. Now, in late 2000, the Internet in general and the Web in particular are the foundation of thousands of companies and are used daily by millions of people worldwide. Despite its wide-spread acceptance and use, there are still many fundamental questions about the Web which remain unanswered: notable among these is why users often have to wait for long periods of time while downloading documents.

Investigation of this simple question quickly exposes gaps in what is currently known about Web performance. The following is a list of some of the simple factors affecting Web performance that are currently not well understood:

- Which component (server, client, network) of the Web is the major contributor to delays?
- How does the interaction between server and network through the transmission control protocol (TCP) affect Web performance?
- What effect does server load have on Web transactions?
- What effect does packet loss have on Web transactions?
- What effects do end-to-end path characteristics have on Web transactions?
- What effect does the HTTP/1.1 protocol have on the performance of Web transactions?

Until now, researchers have considered questions like these from the perspective of either Web servers or the network *in isolation*. While this strategy has led to important improvements in both server and network architecture, future improvements will require considering the server, client and network as an integrated system.

The focus of this dissertation is on developing a deeper understanding of the root causes of long response times in Web transactions while treating the server, client and network as an integrated system. Our goal is to answer this question as precisely as possible, while recognizing that there is not a single, simple answer. While our application of focus is the Web, our intention throughout this work is to develop general methods which can be adapted to other applications as they arise and become popular in the Internet in the future.

There several possible ways to study Web performance, including analytic modeling, simulation and direct measurement. Each approach has its own set of difficulties and all require a fundamental understanding of how each component in the Web works. Efforts to develop analytic models for Web behavior fall into three categories: models of Web clients, models of Web servers characteristics and models of Web transactions. Models developed in each of these areas provide important insights into Web operation and examples are discussed in Chapter 2. Unfortunately, all of the models which have been developed to date fall short of providing answers to any of the questions about Web performance posed above.

Simulation is an attractive alternative to analytic models because it enables the transport and application level protocols to be accurately modeled and analyzed in a controlled and systematic fashion. Simulation has been used effectively to develop a number of enhancements to networking protocols but it is limited in its ability to study the general problem of Web transaction performance for reasons which are described by Floyd and Paxson in [43]. The three principal difficulties they describe are the immense size of the Internet, its rapid changing nature, and its heterogeneity. The current generation of network simulators (*e.g.* the `ns` simulator [124]), cannot be configured with more than a few thousand nodes (due to large memory requirements) which falls well short of the millions of hosts which make up the Web today. While this may only be an issue of simulator technology, and as of now, no one knows how to create simulation models which capture the Web's rapid change and heterogeneity.

Direct measurement offers a means for gaining great insight into Web behavior but is not without significant challenges as well. The difficulties in using a measurement-based approach to analyzing Web transaction performance include:

1. Insuring to as great an extent as possible that measurements are "representative" of general Web use.

2. Overcoming logistical details in constructing, deploying and managing systems a wide area measurement infrastructure.

3. Analyzing the data collected in such a way as to answer the research questions of interest.

Given the difficulties with modeling and simulation, we took a direct measurement approach to studying Web performance. This approach required us to address the problems of representativeness, logistics and analysis noted above. Addressing the issue of representativeness required a detailed understanding of how each component (includes Web servers, client browsers and the hyper text transfer protocol (HTTP) which is the communications mechanism between servers and clients) of the Web operates. This included both the mechanics of the components of the Web *and* the way in which people use the Web. General descriptions of the mechanics of Web components and Web operation can be found in any number of reference books such as [120]. Below the application level are the networking protocols which allow transmission of HTTP messages and data, and at the lowest level are the host systems and physical network which are the Internet. Details of how the Web operates and how people use the Web have been addressed in studies such as Arlitt and Williamson [12] and Cuñha *et al.* [29] which address Web operation from the perspective of the servers and clients respectively.

In the ideal case, our measurement environment would have consisted of instrumentation deployed throughout the *entire* Web; that is, have measurement capability at all hosts and routers throughout the Internet. This would have given us a precise picture of everything that happens in the Web at all times. At the present time, this is clearly infeasible for some of the same reasons which make simulation of the Internet infeasible. Given this, then our task was to break down the Web into its principal components and to insure that each element in our measurement environment was representative of these components. We decomposed the Web into two general components: the end-to-end system (consisting of clients, network and server) and the workload generated in that system.

We addressed the issue of a representative end-to-end system by developing a measurement infrastructure which was deployed in the wide area Internet. We call this system the Wide Area Web Measurement (WAWM) infrastructure. The WAWM infrastructure consists of two components: client nodes located at remote sites which execute Web requests to a Web server system located at Boston University, and a means for generating a representative workload in the infrastructure. By using an actual Web server and clients deployed in the Internet we created a measurement environment that is exposed to the same network conditions as any other Web server or client in the Internet. However, deployment of end systems alone is still not sufficient for measuring all aspects of Web transaction performance since the behavior in the network is not yet addressed. Since we don't have access to the routers along the end-to-end paths between clients and server, WAWM systems are configured to take both active and passive measurements of network conditions during Web file transfers.

While the WAWM infrastructure as described is sufficient for measuring the client, server and network simultaneously, if the measurements taken in the individual components are not synchronized, we will not be able to assess how measurements of one component in the system relate to the other. Packet transfer latencies in WAWM are on the order of milliseconds, so WAWM measurements require time synchronization at least at this level. The standard method for time synchronization on distributed systems is through the Network Time Protocol (NTP) described by Mills in [80]. However, NTP even in the best case only enables synchronization on the order of milliseconds. Our solution to the problem was to insure synchronization between WAWM systems through the use of the Global Positioning System (GPS) clocks. GPS clocks use the pulses emitted from a system of satellites to calibrate ground-based clocks on the order of microseconds. GPS enables a low cost alternative to the only other means for synchronization which would have been the installation of cesium, atomic clocks (which offer the highest level of accuracy but are extremely expensive) in all WAWM systems.

The second major component of the WAWM infrastructure is the means by which workloads are generated. Workload generation required significant attention since the small number of client systems in the infrastructure significantly restricted the load which could be placed on the server. Server load could certainly be a limiting factor in Web transaction performance so we needed a method for testing the server under a wide range of representative loads. There are two general models for Web workload generation: trace based models and analytic models. Trace based workload generators such as Webjamma [127] enable log files collected at Web servers to be replayed. This "black-box" approach enables a workload which is identical to that experienced at an actual Web server but is neither scalable (in the sense of being able to vary the intensity of the workload) nor parameterizable (in the sense of being able to vary the characteristics of the workload). Analytic workload generators are typically based on the WebStone [123] model. This workload generator model makes requests at a constant rate (or as fast as requests can be served) for some small set of files on the server. While this model is certainly scalable and somewhat parameterizable, it bears very little resemblance to actual users making requests from a Web server. Our desire was to have a highly scalable and parameterizable Web workload generator which closely modeled how wide variety of actual users make requests from a Web server.

Our Web workload modeling work began with the simple notion of modeling a user as an ON/OFF process similar to the model proposed in [28]. In that model, an ON time is the period during which a Web page is transferred, and an OFF time the period which users spend looking at the page that has just been received. Despite its simplicity, developing a workload generation model based on ON/OFF behavior required the development of several new analytical models of Web client use. These included models for OFF times, file sizes (the body of the file size distribution), requested file sizes (the body of the requested file size distribution) and the number of embedded files in a Web page. We then developed algorithms which combine seven different statistical client use models (including file sizes, request sizes, document popularity, temporal locality, spatial locality, OFF times and session lengths) in order to generate requests in a scalable, parameterizable and representative fashion.

Our modeling effort resulted in the SURGE Web workload generation tool, which is currently used at over 140 universities, laboratories and corporations world wide. SURGE consists of a statistical distribution generator and an HTTP request generator. Our experience with SURGE shows that it places a load on a Web server which is much more aggressive and variable than other, standard workload generators such as SPECWeb96 [27].

The flexibility of SURGE enabled the study of Web server performance in a controlled, local area environment. Our experiments examined when bottlenecks arise in the CPU, memory or network systems of the server when using different versions of the Hyper Text Transfer Protocol (HTTP). We show that the 1.0 and 1.1 versions of HTTP place significantly different workloads on the server. In particular, HTTP/1.1 placed a lighter load on the network but a significantly greater load on the server when the disk subsystem is the bottleneck. This led to our development and analysis of a connection management method for HTTP/1.1. This method, which we call *early close*, significantly reduces load on the server while preserving the beneficial network characteristics of HTTP/1.1.

The combination of the WAWM infrastructure and the SURGE Web workload generator enabled collection of a wide variety of representative Web transaction performance data. We developed a test methodology which varies server load, network conditions and size of files requested by clients. We examined Web performance with both low and high server loads by connecting SURGE clients to the server in the local area while we measured performance in our WAWM client systems. We exploited the diurnal cycle of Internet traffic [104] to examine Web performance during both low and hight network load conditions by varying the time of day in which measurements are made.

To focus our efforts we restricted the files requested by WAWM clients to three sizes which are sufficient to exercise a range of operating conditions of the underlying transport mechanisms. This combination of server load, network load and client file size resulted in the collection of 12 distinct data sets for each day. The question of how to analyze the data so as to determine the causes of delay in Web transactions remained to be addressed.

Unfortunately, simple statistical analysis of data collected in WAWM is not sufficient to understand Web transaction performance at a fundamental level. Our goal was to break down delays into a level of detail that at least separates delays caused by servers, the network or clients. Attempts had been made to do just this in tools such as Lucent's net.Medic [55] however this tool (and other similar approaches) suffers from the fact that it has no means for precisely separating server delay from network delay. Our solution to this problem was to developed a method of applying *Critical Path Analysis* (CPA) to Web packet traffic which we measured through passive means in WAWM. CPA is a standard means for precisely identifying the set of events which determine the execution time of a program or process and has been applied in many disciplines such as parallel program analysis [131]. Our methodology applies CPA to HTTP transactions (which use TCP as their underlying transport protocol). CPA enables us to first extract the set of packets which determine the overall execution time of a file transfer. It then enables us to break down this packet sequence and determine precisely the fractions of overall response time due to server delays, client delays, and network delays (including propagation delay, network variation delay, and delays due to packet loss).

We implemented CPA for Web transactions in the tool `tcpeval`. Input to `tcpeval` are packet traces from `tcpdump` taken at end hosts along with implementation-specific constants for the version of TCP that is being used by the end hosts. The tool is general in nature, only requiring that end hosts involved in the Web transaction have a version of TCP which conforms to the Reno standard [121]. `tcpeval` works by tracking the evolution of TCP's state as it controls packet flow. It then determines the critical path of the TCP flow by tracing the longest path through the flow from beginning to end. The critical path is then *profiled*, which results in the assignment of delays to the various categories.

Although the primary focus of our work is in developing the modeling, measurement and analysis techniques, we also show preliminary results from measurements taken within WAWM. These results demonstrate the ability of our methods to provide insight to the problems at hand. We show that for our experimental design, high server load generates a significant gap between the transmission of connection setup packets and the first data packet flowing from the server. While this gap can be understood as a consequence of socket and file operations, it is notable that for typical (*i.e.,* short) transfers (1KB and 20KB files in our tests) the gap has a *significant* impact on transfer time. We investigate this effect on a single server (Apache which at present is one of the most widely used Web servers) so we expect that different server architectures would yield different results.

We also show a surprising effect of server load. When the network is heavily loaded (*i.e.,* packet loss rates are high), it is not uncommon for a heavily loaded server to show *better* mean response time for large file transfers (500KB in our tests) than a lightly loaded server. We observe this effect in 33% of our data. Our measurements suggest that this may be due lower packet loss rates in transfers from a heavily loaded server, although this result is not definitive and requires more extensive testing than is presented in this thesis. Finally, we also show that critical path analysis can pinpoint sources of *variability* in Web transfers: while packet losses are the primary contributor to variability of transfer time, we also find that the number of packets on the critical path can vary greatly.

1.2 Contributions of this Work

This thesis makes contributions in the following areas:

1. We develop distributional models of client Web use for OFF times, the body of the file size distribution, the body of the request size distribution and the number of embedded links in a Web page.

2. We develop statistical analysis methods for modeling and generating *hybrid* probability distributions. Hybrid distributions consist of one model for the body of the distribution and a heavy tail distribution (typically Pareto) for the tail of the distribution.

3. We present the SURGE Web workload request generator which is currently used at over 140 locations world wide. This tool enables scalable, configurable, representative Web page request sequences to be made on a Web server.

4. We describe a characterization of Web server performance (including CPU, memory and network performance) over a range of loads when using HTTP/1.0 and HTTP/1.1. This resulted in a proposal for a connection management methodology which has the beneficial server load characteristics of HTTP/1.0 *and* the beneficial network effects of HTTP/1.1.

5. We present an evaluation of the performance impact of wide area network emulation in a local area environment using the NISTnet emulator.

6. We describe the WAWM measurement infrastructure and measurement methodology.

7. We develop a method for applying critical path analysis to TCP transactions and its realization in the tool `tcpeval`which performs a variety of statistical analyses in addition to CPA.

8. We present an end-to-end performance evaluation of HTTP transactions measured in the WAWM infrastructure. This evaluation uses CPA to decompose contributions of delay into six different areas. We show that transfers of small files are dominated by server effects and that the transfers of large files are dominated by network effects.

1.3 Thesis Outline

In Chapter 2 we begin by reviewing related work. Due to its broad scope, our work is based on the contributions of a large number of prior studies in a variety of areas.

The thesis is then divided into two major sections. In the first, we present the details of the architectural components used in measuring Internet performance dynamics.

In Chapter 3, we present the details of the SURGE workload generator. We discuss the models that were developed for the project as well as the hybrid distribution modeling techniques.

In Chapter 4, we begin our investigation of delays in the Web by presenting comparisons of the performance of the two most widely used versions of HTTP in the Web today: v1.0 and v1.1. We analyze both of these in a controlled, local area environment and show when and where bottlenecks arise over a range of loads using SURGE.

In Chapter 5, we present the details of the both the WAWM v1.0 and v2.0 infrastructures. This includes our design of both client and server systems used in the study, analysis of the NISTnet [128] wide area network emulator, use of GPS timers and our test methodology.

The second major section of the thesis focuses on the analysis techniques and results of our measurements. In Chapter 6, we present and discuss the results of packet transfer delays, packet loss characteristics and overall file transfer delays for Web transactions executed in the WAWM infrastructure over a two week period.

In Chapter 7, we present the details of our method of applying CPA to TCP transactions. We present the results of CPA of data analyzed in Chapter 6.

In Chapter 8, we summarize our results from each of the chapters and present conclusions of our study. We also discuss a number of future research projects which are based on the results presented in this thesis.

Chapter 2

Related Work

In this chapter we discuss the studies that are basis for our investigation of Web transaction performance. Our initial focus is on the four components that make up the Web: clients, HTTP, servers and the Internet. We begin by discussing studies of both Web client workload characteristics and Web workload generation. We then address developments and performance results in the HTTP protocol, Web servers, and the Internet. Background in these areas form the basis for most of the work presented in the first major section of this thesis including our work in Web workload generation and Web performance evaluation. We follow this with a discussion of studies in wide area network measurement and timing synchronization. These studies provided important insights during the development of our WAWM infrastructure. We conclude with a discussion of studies of CPA which provided the background for our application of CPA methods in the analysis of Web transactions.

2.1 Web Workload Modeling and Workload Generation

The goal of our workload generation development effort was to create a means for generating a scalable request sequence at the server that was a close as possible to a sequence generated by a set of actual Web clients. Realizing such a method in a tool would enable us to test a variety of hypotheses concerning where, when and why Web performance degrades in both a controlled laboratory environment and in a widely distributed infrastructure.

The task of generating scalable workloads to test Web servers in even the most simplistic way presents a number of difficulties. This subject has been studied intensely since at least 1996 [78]. Many tools have been developed which generate synthetic workloads [14, 26, 88, 27, 95, 123, 126, 1]. All of these tools are based on what we call the "WebStone" model. The WebStone HTTP request generator [123] makes repeated requests either at a uniform rate, or as fast as the client system(s) can, for a small set of files on a Web server. While this is adequate for simply generating a load on a Web server, it is clearly not how real users access a Web server.

Another model for workload generators is to obtain an access log file from an actual Web server and then to "replay" that log file from a set of clients. Examples of these trace-based Web workload generators include [127, 47]. This model has the benefit of being representative of an actual set of clients. However, this *black box* approach offers extremely limited flexibility from the perspective of scalability and parameterization options since as soon as the workload is modified in any way, it may no longer be representative of actual use.

A hybrid of the WebStone and trace-based models is the tool developed in [73]. This tool generates synthetic requests based on a statistical sampling of data from a specific Web server trace. This approach is similar to the approach we use in SURGE however; since it uses only data from the server means that one important feature of Web workloads – OFF times – cannot be modeled accurately. While all of the aforementioned tools are useful for certain kinds of server testing, our approach is to measure server performance under loads which are scalable, parameterizable and as realistic as possible.

Prior to our study, some attempts were made to tie different aspects of Web use together into a single model [61, 71] but none has presented a scalable model which accurately mimics actual users accessing a Web server. In addition, previous work has not created tools based on models that could be used to generate workloads at a Web server (although the model presented in [71] has recently been incorporated in the `ns` network simulator). The work that is most similar to ours in the area of workload generation is by Liu *et al.* [68]. Their tool, called WAGON [69] displays some (but not all) of the same characteristics as SURGE and was developed after our initial publication on Web workload generation [17].

Our approach in developing SURGE was first to gather available information about Web client use characteristics. Studies of Web client characteristics include: analysis of file sizes [28, 29], OFF periods [28, 31], document popularity [29], reference locality [28], and session lengths [52]. The details of all of these characteristics are explained in in Chapter 3 and all are incorporated in the workloads generated by SURGE. However, this set of characteristics alone was not sufficient to generate a scalable, representative Web workload.

In order to generate the remaining models required in SURGE it was necessary to characterize specific aspects of Web client behavior from Web client logs. To develop these models we use the client traces collected at Boston University in 1994 and 1995 by Cuñha *et al.* [29]. Our modeling methodology is similar to the techniques used by Paxson in [102]. These techniques use *goodness of fit tests* described in detail in [22, 30, 109]. In this thesis, we present and use models extracted from the '94 and '95 client traces[1].

Workload generators are often used in a local area network when researchers do not have access to client systems deployed in the wide area. Use of workload generators in the local area means that effects of packet propagation delay and packet loss will not be experienced during tests. This difficulty can be overcome through the use of wide area network (WAN) emulation. WAN emulators apply random delay, loss and bandwidth limits to packet flows between hosts in the local area. Two tools which enable WAN emulation are available at present: `dummynet` [33] and NISTnet [128]. While these tools have been available for some time, only recently have studies been made about their use in Web workload characterization. In [89], Nahum *et al.* use `dummynet` to investigate the effects of WAN emulation in local area web server performance tests. That work focuses on comparing the performance of the Reno and SACK versions of TCP. We utilize the NISTnet WAN emulator in our work and present the results of server performance analysis using NISTnet

[1]A more recent client trace study at Boston University is described in [16]. That paper compares distributional characteristics of the client traces taken in 1998 to the '94 - '95 study. It also describes the implications of the distributional changes for document caching.

in Chapter 5.

2.2 The Hyper Text Transfer Protocol

Clients use the HTTP protocol to communicate with servers in the Web. The design and implementation of HTTP can have dramatic effects on Web transaction performance so a complete and detailed understanding of HTTP is required in our work. Studies of the dynamics of HTTP transactions, including their interaction with the underlying transport protocol, TCP, can be found in [14] and [99]. HTTP/1.0 [19] is currently the most widely used version of the protocol, although HTTP/1.1 is gaining ground due to its adoption in most of browsers being deployed in new systems today. HTTP/1.0 requires a separate TCP connection for transfer of each file associated with a Web object. Since busy servers can serve millions of requests per day, efficiency in setting up and tearing down connections on Web servers is important to good server performance using HTTP/1.0 [84]. However there are many inherent drawbacks to this approach. In [67], Liu and Edwards show that over one quarter of the transaction time in HTTP/1.0 is spent in TCP handshaking before any data flows between Web server and client.

Studies by Mogul [84] and Padmanabhan [99] address the issue of how network conditions affect user-perceived latency during Web transactions. Congestion in the Internet is a well known cause of delay in TCP transfers [104], and when network delays are assumed to be the primary source of user-perceived latency, a number of features of HTTP/1.0 become logical candidates for improvement. These studies were the basis for the development of the 1.1 version of HTTP. HTTP/1.1 provides for the following enhancements, all aimed at improved performance from the perspective of the user: a) persistent connections, b) pipelining, c) link level document compression and d) caching. We evaluate the effects of persistent connections and pipelining in our comparison of HTTP/1.0 and HTTP/1.1 in Chapter 4. Persistent connections reduce packet traffic since ideally only a single TCP connection is set up and used during a client browsing session on a particular server. This means that TCP connections are maintained between consecutive Web object requests (*i.e.* connections are open during OFF times), which means that many more connections can be open on a server at any moment. Efficiently maintaining many open connections places a different kind of burden on the server. Mogul points out in [84] that managing many open connections on the server is the biggest risk of HTTP/1.1. We characterize this risk in detail in Chapter 4.

HTTP/1.0 file requests are made sequentially: a request is sent to the server, the response is read by the client and then the next request is sent. Pipelining in HTTP/1.1 enables both the client and the server to avoid sending only partially full packets by placing as much data (either requests or responses) as possible into each packet and placing separator strings between distinct files within a packet. Pipelining has the effect of reducing the number of request and response packets, thus reducing the number of network round trips required to complete a Web request/response transaction. Nielsen *et al.* [48] show results indicating that pipelining is important to reduce latency when using HTTP/1.1.

A number of other studies have analyzed the performance aspects of HTTP. In [25], Cheng *et al.* compare the performance of HTTP/1.0 versus HTTP/1.1 on a high-latency wireless network and finds that there can be a significant benefits of HTTP/1.1. More recently Liu *et al.* compare the performance of Web servers under a variety of loads and specifically examine the performance of HTTP/1.0 versus HTTP/1.1. Their study closely resembles the work that we published in [18] and essentially confirms our findings presented in Chapter 3. They also confirm that the *Early Close* policy (which we describe and evaluate in Chapter 4) for connection management preserves network benefits while increasing server performance in HTTP/1.1.

2.3 Web Server Performance

Investigations of Web server performance can be divided into two categories: analyses of servers under actual use and under synthetic load generators. Measurements of servers under actual use are important not only for understanding server performance and problems, but also for comparison with synthetic loads. Measurements using synthetic workloads suffer from the fact that they are an abstraction of actual use. However, these studies demonstrate server performance over a range of loads in a controlled fashion, which is not possible in real-use studies.

Several studies have characterized server activity under actual use conditions [85, 64, 12, 11, 74, 8] (in [2, 23, 72, 37] there are analyses of Web proxy logs, which are also relevant). These studies of actual use have focused primarily on server log analysis and have documented a wide variety of use characteristics. However, the amount of performance data from the perspective of CPU, network and memory load on the server and how client access patterns affect Web performance is limited. Because of the variability of real world studies and lack of reproducibility, controlled laboratory studies using synthetic workloads remain important.

A number of studies have characterized Web server performance using synthetically generated workloads [17, 7, 14, 15, 78, 100]. These studies have pointed out that server bottlenecks can occur in a subsystems such as CPU or disk, and that inefficiencies in specific operating system calls can limit performance. However, unlike our work, none of these studies have considered how interactions of networks, clients and servers in the wide area affect end-to-end performance. In [7], Almeida *et al.* evaluate the burden of lingering TCP connections in HTTP/1.0. They also show that for the workloads they consider, the server's CPU is typically the bottleneck in the system. The work in [15] contains a detailed analysis of a server under heavily loaded conditions. Through kernel profiling, Banga and Mogul find inefficiencies in the *select* and *ufalloc* system calls in Digital's UNIX. Improvements in the implementations of these calls lead to significant improvements in overall performance. These experiments were specifically designed to explore the issue of open TCP connections, and their solution is at the kernel level. While these studies make important contributions, they are examples of studying a single component of the Web in isolation and thus do not address how improvements in server performance will affect client perceived latency.

2.4 Network Performance

Considerable work has focused on understanding the behavior and improving the performance of the network infrastructure for Internet applications such as the Web. These studies can be grouped into the following categories: studies of general network packet dynamics, studies of TCP behavior within the context of Web transactions, studies which propose analytic models of steady-state TCP behavior, and studies of TCP packet trace analysis techniques. Each area has implications in our study of Web performance.

One of the earliest and most important studies of **general network packet traffic** was by Leland *et al.* in [66]. This study combined high time resolution measurements of Ethernet packet traffic collected at Bellcore Labs over a four year period with extremely thorough statistical analysis of the data. The study highlighted the *self-similar* or fractal nature of Ethernet traffic. The study by Paxson and Floyd in [108] also used measurements to show that the commonly held notion that network arrivals could be well modeled by Poisson processes was, in fact, incorrect. Further studies of Internet network dynamics such as those in [20, 104] and important improvements to the transport layer in [21, 42, 40, 50, 58, 98] have focused on application-independent issues and typically use a combination of both simulation and measurement. These studies provided us with background for understanding how to analyze Web transactions at the TCP packet level.

There have been some studies of **TCP behavior within the context of Web transactions** [13, 48, 84, 86]; such studies have resulted in a variety of enhancements to HTTP including link level data compression, persistent connections and pipelining, which are all part of the HTTP/1.1 specification [39]. While none of these studies examines how end system behavior affects network performance, two recent studies begin to consider this perspective [5, 97]. In particular, Allman in [5] considers Internet traffic from the perspective of a live Web server collecting both server logs and packet traces over a 17 month period. He characterizes a number of TCP characteristics including the use of round trip time measurements, packet loss characteristics, and the prevalence of SACK deployment. These interesting results could be used in future WAWM studies as seed parameters.

Analytic models for steady-state TCP behavior have been developed over the past three years in [24, 76, 96]. The difficulty with these models is that they assume that no delays are introduced in the end systems which could have a significant impact on their predictions of throughput and latency. Our approach in CPA is designed to assess the effects of both network and end-host delays on overall response time. The packet loss measurement and modeling paper by Yajnik *et al.* is also relevant to our work [130]. In that paper, three different Markov models are developed to explain loss. We present results of Web performance for transfers with and without loss in Chapter 5

Our work also depends on **TCP packet trace analysis**, and extends the set of tools available to perform automated packet trace analysis. The most basic problem with TCP packet trace analysis is that traces are usually large and complex, and key information in a trace may be difficult to identify. The most simple analysis techniques are graphical, such as time line plots or sequence plots [58, 117] which enable one to understand the data and acknowledgment sequences of a transaction. In [21], Brakmo *et al.* developed a sophisticated graphical tool which enables multiple details of a TCP transaction to be tracked simultaneously. In [103], Paxson describes the tool `tcpanaly` which he used to document a wide variety of unusual behaviors from a number of different TCP implementations. Paxson's work is also significant for our study because it points out the pitfalls of packet traces relevant to the development of `tcpeval`.

One of the implications of our analysis of Web performance is that bursts of packets from a Web server may actually *cause* loss along an end-to-end path. This could happen when a router buffer in the path is nearly full during the duration of a test and that even small packet bursts from a single source could lead to loss from that burst. While we do not evaluate the details of this hypothesis in this thesis, if true it could mean that a *rate-controlled* TCP would perform better in the face of congestion. Examples of rate-controlled TCP appear in [4, 60] and we discuss the analysis of bursts of data as future work in Section 8.6.

2.5 Wide Area Network Measurement

Development of our WAWM infrastructure is based on techniques and methods developed in a number of other wide area measurements studies. The notion of making measurements in the Internet to understand its behavior is not new: Kleinrock did exhaustive studies of the ARPANET's behavior in the mid 70's [62]. Two of the best examples of successful measurement-based studies are Leland *et al.* [66] and Paxson [105]. Both studies utilized measurement infrastructures to gather general network traffic data and then analyzed the data using a variety of techniques. We are influenced by Paxson's *Network Probe Daemon* (NPD) study [105] which utilized 37 measurement sites distributed across the Internet. This enabled measurement of routing and packet transfer behavior over 1,000 distinct Internet paths. In addition to the performance results, that work also discussed timing synchronization, distributional modeling methodology and measurement methodology which all provided important guidance in our work.

At present, several wide area network measurement projects are either proposed or underway. None do or propose to do the type of work that we present in this thesis, although there are some similarities to our work. The projects which are the most closely related to ours are:

The National Internet Measurement Infrastructure (NIMI) project [3] is the next generation of Paxson's NPD study [104]. The project's goal is to develop and deploy a secure management platform for facilitating general types of coordinated measurement in the Internet. The software has been distributed on 51 systems across the Internet at present. NIMI is not focused on a single measurement objective and is currently used in a number of different studies. NIMI differs from WAWM in that we were specifically interested in Web performance measurements.

Cooperative Association for Internet Data Analysis (CAIDA) [45] manages a variety of measurement and data collection efforts but the project most closely related to ours is called Coral Reef [92]. This project is focused on building a passive network measurement and analysis tool suite to be used with high speed packet filters such as the OCxMONs [10]. The emphasis of this project is the generation of a suite of measurement and analysis tools and not on a specific research objective.

AT&T Research has been making detailed measurements in AT&T Worldnet for a number of years. Beginning with the development of their PacketScope passive measurement system [9], they have been able to extract large volumes of detailed packet traces and have developed methods for reconstructing Web browsing sessions from those traces [36]. Recently, they have enhanced their network measurement infrastructure to study the problems of traffic demand modeling [38] and flow reconstruction [32]. The measurement work being carried out at AT&T has the advantage of having complete visibility of the interior of Worldnet but is constrained by the fact that it *only* instruments a single ISP.

Internet Protocol Performance Metrics (IPPM) working group within the IETF [57, 107]. The IPPM working group's mission is to "develop a set of standard metrics that can be applied to the quality, performance, and reliability of Internet data delivery services." The project which has grown out of the IPPM work is called Surveyor [112]. This project, facilitated by Advanced Network Systems, is focused on taking one-way measurements of packet delay and loss between 61 systems deployed world wide. The most important aspect of this system from our perspective is that it has been constructed with Global Positioning Satellite (GPS) clocks which enable very precise synchronization between their systems. We have learned much about deploying systems with GPS from this project.

Network Analysis Infrastructure (NAI) projects at the National Laboratory for Applied Network Research (NLANR) [44] are focused in three areas. The first is their Active Measurement Program (AMP). This is an infrastructure consisting of a confederation of approximately 100 universities principally in North America that participate in on-going round trip time, loss and topology measurements [93]. The second is their Passive Measurement and Analysis (PMA) program [94]. This is an infrastructure consisting of a confederation of 17 sites that participate in on-going passive measurements of throughput and packet flows. The third project is NLANR's Squid cache hierarchy [46]. This is a series of 8 caches distributed in the Internet which collect logs and publish them on a daily basis.

Keynote Systems, Inc. [54]. Keynote has an infrastructure of over 1,000 measurement systems deployed world wide which they use to measure the performance of Web sites and, to a lesser extent, Internet Service Providers. Their principal service is to provide measurements of Web server download time to their customers, who are either content providers or content delivery companies. While Keynote provides a useful service for their customers, they provide only DNS response time and document download time. They are not able to differentiate between network and server delay in their system. Keynote also publishes a "performance index" for both consumer and business sites. This index attempts to assign a single "health" index value to the 40 most heavily used Web sites in each category.

A more accurate approach to characterizing client response time is used in **net.Medic** [55]. This tool, which resides on a client system, infers server delays from packet-level measurements. More recently, Huitema [53] presented measurements that differentiate between network and server delay by equating server delay with the time between HTTP GET and receipt of the first data packet. Results presented in Chapter 7 support this as a good first-order approximation. We show that in many cases (*e.g.,* for small transfers) the majority of server-induced delay occurs at precisely this point in the transaction. However, for large transfers when the server is under heavy load, we also show that there can be significant server delays long after connection start up. The work in [53] also points out that DNS lookups can also be a significant cause of delay; since we are mainly interested in network/server interaction, we do not consider these sources of delays in this paper.

NPD Group [49]. NPD Group (no relation to Paxson's NPD study) is a market research organization which has tracked the Web use of thousands of households continuously since 1996. They pay participants in their study to allow their Web browsing sessions to be monitored. Results include estimated number of different people who visit Web sites, frequency of visits, length of visits, comparative data across sites and some additional data. There are a number of other market research companies conducting similar studies.

The Collaborative Advanced Interagency Research Network (CAIRN) is a national passive measurement test bed consisting of 31 installations in government, academic and industrial sites nationwide [91]. Their primary focus has been to provide a set of programmable routers in the network for general research. However, in the future, they intend to broaden their scope to enable colocation of host measurement systems at their sites. This would enable application-level research similar to our WAWM infrastructure.

There are a number of measurement studies of routing behavior. The **Route Views** project is one example [125]. This project passively collects BGP routing information via a non-forwarding router located at the University of Oregon. This router is peered with 36 other networks. Data from this project is used in a variety of studies. We envision that routing level data will be an important component in the future expansion of our own research.

At the structural level of the Internet, there are also several measurement studies of Internet topology. The largest of these is the Skitter project being conducted at CAIDA [118]. This project consists of 17 source sites which send out `traceroute` [59] probes to approximately 54,000 destination sites world wide. Topology also plays a significant role in overall performance and, like routing, it will be considered in greater detail in our future work.

Each of the projects described here attempts to measure or analyze some aspect of Internet performance. Our measurement efforts are similar in that we propose to use a wide area measurement infrastructure and that there are significant data management and analysis tasks. However, our project is different for a number of reasons. First, our project focuses on a *user level* application; the Web. Most of the projects listed above focus on general network measurement, monitoring and analysis. Although we take network level measurements during tests, measurements of user level characteristics are made at the same time. Second, since we control the entire end-to-end systems in WAWM, we have the ability to install performance monitoring tools on our end hosts which enable detailed system measurements during tests. This gives us the unique capability to do *integrated* network/host measurement.

2.6 Network Time Synchronization

The nature of our investigation of the root causes of delay in Web transactions necessitates tight synchronization of system clocks in WAWM. One-way packet propagation delays in the Internet are on the order of milliseconds, with typical times in the 50ms to 100ms range. In order to accurately measure one-way delays at this granularity, clocks on the two end host systems must: a) have a timestamp resolution on packets that is on the order of tens of microseconds, and b) be synchronized such that the clock drift between the two systems is on the order of single milliseconds. We rely on the *fast timer* kernel patch in Linux (which is part of the PPS patch for NTP [113]) to give us microsecond timestamp resolution. The problem of synchronizing clocks between systems is more problematic.

The principal work on network clock synchronization was done by Mills in [80, 82]. These studies have resulted in the Network Time Protocol (NTP) and the ntpd daemon [116] which enables clock synchronization on host systems in networks. NTP is used in WAWM to synchronize clocks but instead of synchronizing them from standard, tier 1 time servers (which have either GPS or cesium, atomic clocks), we use it to synchronize from GPS clocks installed on WAWM systems themselves.

Another means to correct clock skew between measurement systems in a network is to adjust the timing measurements off-line. Algorithms for doing off-line clock calibration are presented in [106, 87]. While these methods are certainly useful, our approach is to simplify the entire procedure by synchronizing all clocks in WAWM via GPS.

2.7 Critical Path Analysis

Our approach to understanding the details of Web transaction performance uses the method of *critical path analysis* [70]. Critical path analysis is the natural tool for understanding distributed applications because it identifies the precise set of activities that determine an application's performance [51, 79, 131].

Prior to our work, CPA has not been applied to TCP transactions. In [105] Paxson discusses the notion of "cause-effect" issues in TCP transactions, which is an idea that also underlies our approach. However, Paxson had difficulty fully developing this notion because he was interested in establishing this relationship by measuring at a single point along the end-to-end path. While they do not explicitly use the term "critical path analysis", Schroeder and Burrows essentially do critical path analysis of the Firefly RPC mechanism in [114]. In this work the authors define a static set of events which make up the *fast path* in RPC. They either estimate or measure the performance of each event in a local area environment. They then suggest a series of improvements to the fast path which they estimate can speed up the RPC mechanism by a factor of 3.

Part I

An Architecture for Internet Measurements

Chapter 3

Generating Representative World Wide Web Workloads

3.1 Overview

In order to conduct our measurement study without having to recruit large number of actual users, we needed to develop a method for placing loads on servers which were as representative as possible. In particular, the ability to generate a stream of HTTP requests that mimics a population of real users is important not only for our study but more generally for performance evaluation and capacity planning of servers, proxies, and networks.

Generating representative Web reference traces is difficult because Web workloads have a number of unusual features. First, empirical studies of operating Web servers have shown that they experience highly variable demands, which is exhibited as variability in CPU loads and number of open connections (*e.g.,* see [85]). This indicates that it is important to pay attention to those properties of Web reference streams that have been shown to exhibit high variability, such as request interarrivals [31, 35] and file sizes [28, 12].

The second unusual feature of Web workloads is that network traffic due to Web requests can be *self-similar—i.e,* traffic can show significant variability over a wide range of scales. Self-similarity in traffic has been shown to have a significant negative impact on network performance [34, 101], so it is an important feature to capture in a synthethic workload.

To capture these properties in a workload generator, one of two approaches could be used: a trace-based approach or an analytic approach. Trace-based workload generation uses prerecorded records of past workloads and either samples or replays traces to generate workloads. In contrast, analytic workload generation starts with mathematical models for various workload characteristics and then generates outputs that adhere to the models.

These two approaches each have strengths and weaknesses. The trace-based approach has the advantage that it is easy to implement, and mimics activity of a known system. However, it treats the workload as a "black box." As a result, insight into the causes of system behavior is hard to obtain. Furthermore, it can be hard to adjust the workload to imitate future conditions or varying demands. Analytic workloads do not have these drawbacks, but they can be challenging to construct for at least three reasons. First, it is necessary to identify those characteristics of the workloads which are important to model. Second, the chosen characteristics must be empirically measured. Third, it can be difficult to create a single output workload that accurately exhibits a large number of different characteristics.

In this chapter we describe methods we have developed for generating analytically-based Web reference streams. Our goal is to imitate closely a stream of HTTP requests originating from a fixed population of Web users. These methods are embodied in our tool SURGE. The methods used in SURGE have the advantages that come with the analytic approach. In particular, the models used in SURGE are explicit and so can be examined directly by the user. In addition, SURGE's models can be varied to explore expected future demands or other alternative conditions.

To construct an analytic Web workload, it is necessary to address the three challenges described above. In this chapter we show how each challenge was addressed. First, we describe the set of characteristics of Web reference streams we chose to model, and explain why we believe that this set is important. Second, we describe the results of new measurements of the Web that were needed to populate the SURGE models. Third, we describe the issues involved in incorporating these models into a single output reference stream, and how we resolved those issues.

The final result, SURGE, shows a number of properties that are quite different from a common and widely used Web workload generator: SPECweb96 [27]. We configured SURGE and SPECweb96 to exercise a server at equal rates measured in terms of bytes transferred per second. Under these conditions, SURGE places much higher demands on the server's CPU, and the typical number of open connections on the server is much higher.

In addition we show that in an another important respect, the workload generated by SURGE is quite different from that of typical Web workload generators. In particular, the traffic generated by SURGE shows self-similar properties; we show that this is not generally true for a typical alternative Web workload generator (SPECweb96) when run at high loads. Because SURGE's analytic approach makes its models explicit, we can examine the causes of self-similarity in SURGE traffic. Based on this insight we conclude that most Web workload generators proposed to date probably do not generate self-similar traffic at high loads.

3.2 Web Workload Characteristics

Our goal in developing SURGE is to be able to exercise servers and networks in a manner representative of the Web. In particular, for servers we are interested in the behavior of the network stack and filesystem, and their associated buffering subsystems. In the case of both networks and servers, we are also interested in the effects of high variability in the workload.

These motivations drove the choice of particular Web workload characteristics used in SURGE. These characteristics can be divided into two main categories. The first category is concerned with the idea we call *user equivalents.* The second category is the set of *distributional models.*

3.2.1 User Equivalents.

The idea behind user equivalents is that the workload generated by SURGE should roughly correspond to that generated by a population of some known number of users. Thus, the intensity of service demand generated by SURGE can be measured in user equivalents (UEs).

A user equivalent is defined as a single process in an endless loop that alternates between making requests for Web files, and lying idle. Both the Web file requests and the idle times must exhibit the distributional and correlational properties that are characteristic of real Web users. Each UE is therefore an ON/OFF process; we will refer to periods during which files are being transferred as ON times, and idle times as OFF times. UEs have a straightforward implementation as independent threads or processes.

There is not generally a simple relation between UEs and metrics like HTTP requests per second or bytes requested per second. This is because the rate at which an individual UE requests service is dependent on the state of the system; delays in the server or network in transferring data will increase the time between requests. However, for any particular experiment it is easy to observe the *effective* request rate and data transfer rate. We use this approach in Section 3.5 to make comparisons between workloads on the basis of equivalent system loading.

Basing the workload model on user equivalents has important effects on system performance. Since each UE has significant idle periods, a UE is a bursty process. Each UE exhibits long periods of activity followed by long periods of inactivity.

This is quite different from approaches typically taken in other Web workload generators. Most other workload generators send requests without preserving consistent properties of OFF times [27, 95, 123]. The general approach used in these workload generators is simply to make requests for files from a server as quickly as possible. We will show in Section 3.5 that ignoring OFF times destroys self-similarity in the resulting traffic at high loads.

Specifying UE behavior during the ON period requires understanding some of the details of Web organization. In particular, Web files may include by reference other files; these files are required to display the result properly. (Typically these included files might supply images or graphics.) Thus the user's request for a single Web file often results in the transfer of multiple files from the Web server. We call a Web file along with all the files that also must be transferred to display it a *Web object.*

The particular details of how components of a Web object are transferred depends on the browser used and the version of the HTTP protocol employed. HTTP 0.9/1.0 uses a separate TCP connection for each file, while HTTP 1.1 allows a single TCP connection to be used for multiple files. In addition, some HTTP 0.9/1.0 browsers open multiple simultaneous TCP connections when transferring components of a Web object. For the experiments reported here, SURGE was configured to use HTTP 0.9 and multiple TCP connections were not used; however modifying SURGE to use other styles is straightforward.

This style of transferring Web objects means that there are two kinds of OFF time, as shown in Figure 3.1. *Inactive OFF* time corresponds to the time between transfers of Web objects; this is user "think time." *Active OFF* time corresponds to the time between transfer of components of a single Web object; it corresponds to the processing time spent by the browser parsing Web files and preparing to start new TCP connections.

3.2.2 Distributional Models.

Having described the UE basis for the design of SURGE, we describe the set of probability distributions needed by each UE.

An important feature of some of the distributions we will consider is that they exhibit *heavy*

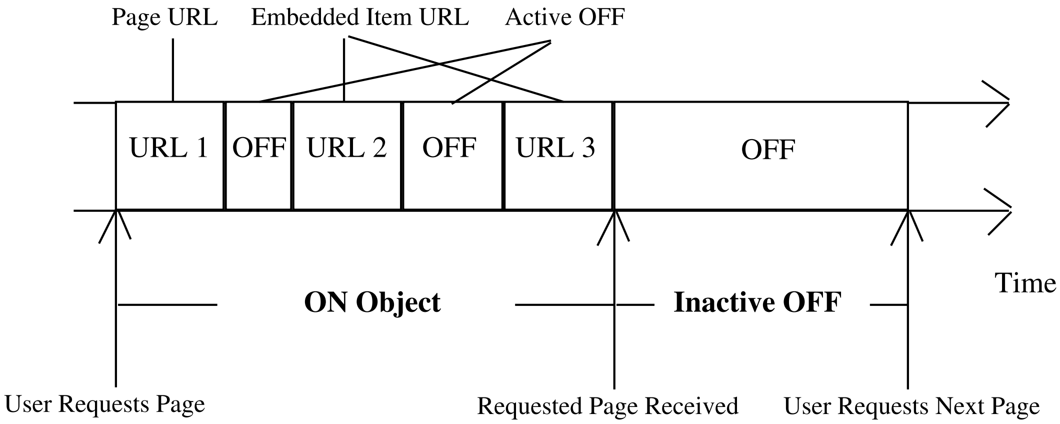


Figure 3.1: ON/OFF Model Used in SURGE (HTTP 1.0; one TCP connection at a time).

tails. We say here that a random variable X has a heavy-tailed probability distribution if its upper tail declines like a power law, *i.e.,*

$$P[X > x] \sim x^{-\alpha} \qquad 0 < \alpha \leq 2$$

where $a(x) \sim b(x)$ means $\lim_{x\to\infty} a(x)/b(x) = c$ for some constant c. Random variables whose distributions are heavy tailed exhibit very high variability; in fact, their variance is infinite, and if $\alpha \leq 1$, their mean is also infinite.

To fulfill SURGE's goals, its output must adhere to the following six statistical properties of Web reference streams.

1) File Sizes. In order to exercise the server's filesystem properly, it is important that the collection of files stored on the server agree in distribution with empirical measurements. As noted in [28, 12], these distributions can be heavy-tailed, meaning that the server's filesystem must deal with highly variable file sizes.

2) Request Sizes. In order to exercise the network properly, the set of files transferred should match empirical size measurements. We call this set the *requests.* The distribution of requests can be quite different from the distribution of files because requests can result in any individual file being transferred multiple times, or not at all. Again, empirical measurements suggest that the set of request sizes can show heavy tails [28].

Note that the difference between the sets we call file sizes and request sizes corresponds to differences in distribution between what is stored in the filesystem of the server and what is transferred over the network from the server.

3) Popularity. A third property of the workload, related to the previous two, is the relative number of requests made to individual files on the server. Popularity measures the distribution of requests on a per-file basis. Note that even when the previous two properties are fixed (distributions of file sizes and request sizes) there still can be considerable flexibility in how requests are distributed among individual files. The distribution of popularity has a strong effect on the behavior of caches (*e.g.,* buffer caches in the file system), since popular files will typically tend to remain in caches.

Popularity distribution for files on Web servers has been shown to commonly follow Zipf's Law [132, 8, 29]. Zipf's Law states that if files are ordered from most popular to least popular, then the number of references to a file (P) tends to be inversely proportional to its rank (r). That is:

$$P = kr^{-1}$$

for some positive constant k. This property is surprisingly ubiquitous and empirical measurements of the exponent are often quite close to -1, although the reasons behind this effect in Web workloads are unclear. This distribution of references among files means that some files are extremely popular, while most files receive relatively few references.

4) Embedded References. In order to capture the structure of Web objects, it is important to characterize the number of embedded references in a Web object. This is done by characterizing the distribution of the number of embedded references in Web files. In the configuration of SURGE used in this paper, the number of embedded references is important because OFF times between embedded references (Active OFF times) are typically short, while OFF times between Web objects themselves (Inactive OFF times) are typically much longer. Previous work has not looked closely at the question of how many references are typically contained in a Web object.

5) Temporal Locality. Temporal locality in Web workloads refers to the likelihood that, once a file has been requested, it will be requested again in the near future. Accurate characterization of temporal locality is important because caching effectiveness can be significantly increased when temporal locality is present.

One way to measure temporal locality is using the notion of *stack distance*. Given a sequence of requests, its corresponding stack distance sequence can be generated as follows. Assume that the files are stored in a push-down stack. Considering each request in order, move the requested file to the top of the stack, pushing other files down. The depth in the stack at which each requested file is found is the request's stack distance [8, 77].

Stack distance serves to capture temporal locality because small stack distances correspond to requests for the same file that occur close to each other in the reference stream. Note that, assuming the initial contents of the stack are known, the stack distance sequence contains information equivalent to the request sequence; each can be obtained from the other.

Thus the distribution of stack distances for a Web request sequence serves as a measure of the temporal locality present in the sequence. Typical distributions for request sequences arriving at Web servers have been studied in [8]; results show that these are well modeled using lognormal distributions. Since the lognormal distribution has most of its mass concentrated at small values, this is indicative that significant temporal locality is often present in Web request sequences.

6) OFF Times. As described in the previous section, accurate modeling of inactive OFF times is necessary to capture the bursty nature of an individual Web user's requests. Proper characterization of active OFF times is necessary to replicate the transfer of Web objects. Previous work has measured OFF times [28] and the related question of interarrival times of HTTP requests at a server [31].

In summary, the notion of user equivalents along with the six distributions discussed in this section constitute the definition of the SURGE model for Web workload generation. These characteristics represent a large set of contraints on the properties of the workload generated by SURGE. In the next two sections we describe the difficulties presented in trying to satisfy all these constraints simultaneously, and how those difficulties are addressed in SURGE.

3.3 Obstacles to Creating Representative Web Workloads

Two basic problems arise in building a workload generator to meet the requirements described in the previous section. First, models for each of the six distributions are required, and second, methods for combining those distributions into a single output stream are required.

3.3.1 Distributional Model Obstacles

To populate our set of distributional models fully, three new distributions were needed.

File Sizes on the server were identified as a key characteristic of Web workloads in Section 3.2. File sizes on servers were studied in [28] but that work concentrated only on the tail of the distribution. For SURGE we need a model for the distribution that is accurate for the body of the distribution as well as the tail.

Active OFF Times have not been specifically addressed in other studies. Client OFF times in general are described in [28]. However, no attempt to describe Active OFF time as a separate distribution was made; thus a model for Active OFF time was required.

Finally, a model for the number of **Embedded References** in a document was required. This number is difficult to extract from client trace data since there is typically no record in the data that indicates which documents are embedded. This characteristic can, however, be inferred using Active OFF times (as is described in Section 3.4).

The task of developing distributional models for each of these Web characteristics required the analysis of empirically measured Web workload traces. The most common way of specifying a statistical model for a set of data the is through the use of *visual* methods such as quantile-quantile or cumulative distribution function (CDF) plots [30]. These methods, however, do not distinguish between two closely fitting distributions nor do they provide any level of confidence in the fit of the model. To address this drawback one can use goodness-of-fit tests [30, 102]. However, these tests also present a number of problems. Methods which place data into bins (such as Chi-Squared tests) suffer from inaccuracies due to the choice of bin size. Methods based on empirical distribution functions (such as the Anderson-Darling test) are more likely to fail when applied to large datasets. Since most empirical Web measurements constitute large datasets, the methods we used to select distributional models needed to address these shortcomings of goodness-of-fit tests. We explain these methods in detail in Section 3.4.

3.3.2 Obstacles in Combining Distributions

Generating a single output workload which exhibits each of the six characteristics which make up the SURGE model is difficult. We address this problem by developing methods for matching requests to files and for generating a representative output sequence.

3.3.2.1 The Matching Problem

The matching problem starts from three Web workload characteristics: file size distribution, request size distribution, and popularity. Given these three distributions, determining the total number of requests for each file on the server is the matching problem. The matching problem arises because even when these three distributions are fixed, there still can be considerable flexibility in how requests are actually distributed among individual files.

First, any server will contain a set of uniquely named files each with a specific size. Let X be the set of file sizes on the server, where x_i is the size of file i. Second, using Zipf's law we can calculate the set Y which consists of the number of references to each of the files on the server (the popularity distribution). However, Zipf's Law does not determine *which* files correspond to each element of Y. Finally, request size distribution is also described by an empirical distribution, $F(x)$.

Given these as inputs, the matching problem is to find a permutation of the set Y such that the following conditions are satisfied:

1. There is a one-to-one mapping between elements in the sets X and Y. The permutation of Y will be described by the set of indices Z which is a permutation of integers $1, ..., n$. Thus, each y_{z_i} is the number of requests for file i.

2. The mapping between X and Y results in a set of file requests. This set can be described by a CDF $G(x)$ that is defined as follows:

$$G(x_j) = 1 - (\sum_{i=1}^{j} y_{z_i} / \sum_{i=1}^{n} y_{z_i})$$

   Thus, we can determine a desired value for each y_j which we will call $\widehat{y_j}$ by equating $G(x)$ and $F(x)$ and solving for $\widehat{y_i}$ so:

$$\widehat{y_j} = F(x_j) \sum_{i=1}^{n} y_{z_i} - \sum_{i=1}^{j-1} y_{z_i}$$

3. We create a permutation Z such that $\widehat{y_i}$ is as close as possible to y_{z_i} for all i.

3.3.2.2 Temporal Locality

The result of the matching process is that each unique file will be matched with a total request value. However, this is still not sufficient for the generation of a request stream which exhibits temporal locality. Temporal locality can be analyzed using the distribution of stack distances. This distribution is an indication of temporal locality because stack distance measures the number of intervening references between references to the same document [8]. To obey temporal locality, the sequence of document requests must be arranged such that when cast in a push-down stack the resulting distribution of stack distances will match empirically measured distributions.

In addition to the distribution of stack distances, a method for request sequence generation must distribute references to each file as evenly as possible throughout the entire sequence.

3.4 Solutions Adopted in SURGE

3.4.1 New Distributional Models

The BU client trace data sets discussed in [29] were used to develop the three models required to complete SURGE. These traces record the behavior of users accessing the Web over a period of two months. In general, it is desirable that a large number of datasets from different environments be examined to determine representative distributions. However, our focus in the work to date was in defining the necessary set of workload characteristics, and so we did not include a wide range of datasets in our analysis. We expect to examine a larger number of datasets in future work, and for that reason SURGE's algorithms and general structure were developed to allow easy adaptation to other distributions. Thus, SURGE has been developed as a highly parameterizable tool.

To develop these models, we used standard statistical methods, similar to those used in [102]. We used the Anderson-Darling (A^2) empirical distribution function (EDF) test for goodness of fit (recommended in [30, 102]) and the λ^2 test (described in [109]) to compare how well analytic models describe an empirical data set. The basic technique is as follows:

1. Use log-log complementary distribution (LLCD) plots to determine whether or not the data set has a heavy tail. If so, then hybrid modeling may be necessary (hybrid models may also be necessary even if the data is not heavy tailed).

2. Use standard visual techniques such as simple histograms or complementary distribution function (CDF) plots to narrow the set of candidate models for the data. Logarithmic transformation may be necessary to distinguish important characteristics.

3. If the data appears to best be modeled by a hybrid model then use censoring methods to determine how to divide that data.

4. Use maximum likelihood estimators (MLE) to estimate parameters for candidate models for the data.

5. Use goodness of fit *tests* such as the Anderson-Darling (A^2) [30] test to see if there is a close fit between model and data. If this test shows no significance, then use a random subsampling technique to test fit for small sample sizes.

6. Use goodness of fit *metrics* like the λ^2 [109] test to determine a discrepancy measure between data and model. This test is especially important if all goodness of fit tests fail.

7. Outlier analysis should be done to determine if there are any data points which are skewing analyses.

File Sizes. Completion of the file size model began with the assumption that the heavy tailed characterization of the distribution as described in [28, 12] was accurate. The model was then developed as a hybrid consisting of a new distribution for the body, combined with a Pareto distribution for the upper tail. Details of the model are shown in Table 3.1. The λ^2 test on the body of data (11,188 points) versus a number of distributional models (lognormal, Weibull, Pareto, exponential, and log-extreme) showed that the best (smallest) λ^2 value was for the lognormal distribution. The CDF of the log-transformed data versus the normal distribution can be seen in Figure 3.2.

However, the A^2 showed no significance in terms of goodness of fit. The failure of the A^2 test was due primarily to the fact that a fairly large data set was used for the test; this is a common problem with EDF tests [30, 102]. Therefore a method of testing goodness of fit using random sub-samples (as described in [102, 22]) was used, which indicated a good fit between the EDF of file sizes and the lognormal distribution.

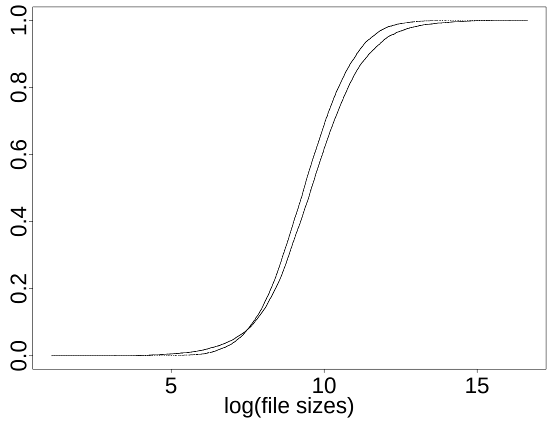


Figure 3.2: CDF of Log-transformed File Sizes vs. Fitted Normal Distribution

Censoring techniques were employed to determine where to split between the lognormal distribution for the body and the heavy tailed distribution in the tail. A sample is said to be *right censored* if all observations greater than some value are missing. The body of our sample can be assumed to be right censored since we assume that it is contaminated with a heavy tail. We use the A^2 statistic to determine the cutoff point between the body and the tail. By successively increasing the amount of right censoring in the empirical data and then testing goodness of fit, we determine the cutoff between the two distributions to be at approximately 133KB (93% of the files lie below the cutoff). The cutoff value along with the hybrid distributional model allows us to generate the appropriate distribution of file sizes on the server.

Active OFF Times. We consider an OFF time to be "Active" if it is less than a threshold time, which we chose to be 1 second based on inspecting the data. λ^2 tests showed Weibull to be the best fit of those considered. The CDF plot of the set of Active OFF times versus the fitted Weibull distribution is shown in Figure 3.3. We found no significance at any level for the A^2 test, which we again attribute to the relatively large sample size (40,037 elements). However, for random sub-samples the A^2 test indicated a good fit to the Weibull model.

Embedded References. The number of embedded references in each file was extracted from the traces of file transfers by identifying sequences of files fetched by a given user for which the OFF time between transfers was always less than the one second threshold (resulting in 26,142 data points). Initial inspection of this data set showed that its distribution had a long right tail. Generation of distributional plots suggested the Pareto distribution as the best visual fit for the data. A least squares estimate of the tail slope in a log-log complementary distribution plot resulted in an estimate of α which gave a good visual fit as can be seen in Figure 3.4.

A^2 tests once again failed to find significant goodness of fit. Even the random sub-sample method did not indicate a good fit which we attribute to the fact that there were only a few values in the tail of our empirical data and thus very few in the tails of the sub-samples.

A summary of the model distributions and parameters used in SURGE are given in Table 3.1.

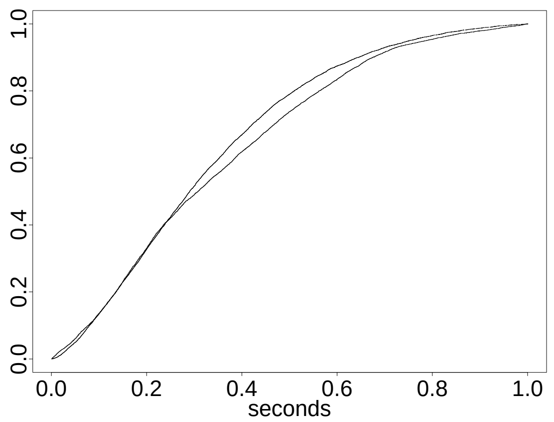


Figure 3.3: CDF of Active OFF Times vs. Fitted Weibull Distribution

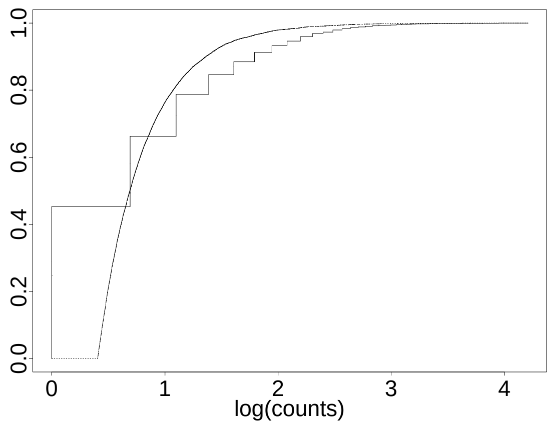


Figure 3.4: CDF of Number of Embedded References vs. Fitted Pareto Distribution

Component	Model	Probability Density Function	Parameters
File Sizes – Body	Lognormal	$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-(lnx-\mu)^2/2\sigma^2}$	$\mu = 9.357$; $\sigma = 1.318$
File Sizes – Tail	Pareto	$p(x) = \alpha k^\alpha x^{-(\alpha+1)}$	$k = 133K$; $\alpha = 1.1$
Popularity	Zipf		
Temporal Locality	Lognormal	$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-(lnx-\mu)^2/2\sigma^2}$	$\mu = 1.5$; $\sigma = 0.80$
Request Sizes	Pareto	$p(x) = \alpha k^\alpha x^{-(\alpha+1)}$	$k = 1000$; $\alpha = 1.0$
Active OFF Times	Weibull	$p(x) = \frac{bx^{b-1}}{a^b} e^{-(x/a)^b}$	$a = 1.46$; $b = 0.382$
Inactive OFF Times	Pareto	$p(x) = \alpha k^\alpha x^{-(\alpha+1)}$	$k = 1$; $\alpha = 1.5$
Embedded References	Pareto	$p(x) = \alpha k^\alpha x^{-(\alpha+1)}$	$k = 1$; $\alpha = 2.43$

Table 3.1: Summary Statistics for Models used in SURGE

3.4.2 Solutions to the Matching Problem

The solution to the matching problem as described in Section 3.3.2.1 results in a mapping of requests to files such that the desired file size distribution, request size distribution, and popularity characteristics are all present. The success of a matching is expressed as the difference between $\widehat{y}_i$, the desired number of references to file of size x_i, and y_{z_i}, the actual number of references to file of size x_i. There are a number of ways in which these differences can be combined to form an optimization problem. First, we could bound the maximum difference between the two distributions: $\max |\widehat{y}_i - y_{z_i}|$. Second, we could bound the total error between the two distributions: $\sum_{i=1}^{n} |\widehat{y}_i - y_{z_i}|$. Finally, we could bound the error in some important portion of the distribution: $|\widehat{y}_i - y_{z_i}|$ given $z_1 ... z_{i-1}$ are already determined.

A single algorithm is optimal for the first two cases. This algorithm creates the permutation index *Z* such that the largest value in $\widehat{y}$ is matched to the largest value in *y*, the next largest in $\widehat{y}$ in is matched with the next largest in *y* and so on. The proof of optimality of this method for the first two cases above is straightforward.

However, the optimal method can allocate error between the two distributions in an undesirable way (*e.g.,* it can concentrate the error in the tail of the distribution). Since large files cause the greatest impact on network and server performance, the ability to match most closely in the tail is important. Therefore another method of matching was developed which allows matching in either the tail or head of the distribution to be optimal. This method matches values in the set Y to those generated from $F(x)$ beginning with either the smallest value in X or the largest. This method results in very close fits in the tail of the distribution, and does not introduce large errors in the body.

3.4.3 Solution to the Sequence Generation Problem

To generate reference sequences with the proper temporal locality, SURGE begins by placing each of the individual file names in a stack (the initial ordering is not important). A sequence of values drawn from the proper lognormal distribution (Table 3.1) is then generated. This sequence is inverted to obtain a sequence of names. This is done by repeatedly selected files from the stack at a distance equal to the next value in the sequence, reordering the stack after each selection.

Unfortunately this simple method, if followed exactly, results in a nonuniform distribution of file names throughout the request sequence. Therefore the method was modified by defining a small window on either side of the location specified in the lognormal sequence. Each of the files in this window are then assigned a weight whose value is proportional to the remaining number of requests required for that document. The choice of which file to move to the top of the stack is then based probabilistically on the weights of the files within the window.

We found that this method of file sequence generation gives a very good distribution of file names throughout the sequence, and the resulting stack distance values still follow the desired lognormal distribution.

Finally, the use of multiple client hosts presents a problem for temporal locality. When UE threads are running on the same host they can share a common file name sequence, each thread using the next file name in the sequence as needed. However, when UE threads are running on multiple hosts, they cannot share a common list without significant synchronization overhead. To handle this case we generate independent file name sequences with related stack distance properties. This is done by scaling the values output from the lognormal distribution by the number of clients which will be used in the simulation. In the case in which requests from separate hosts interleave at the server in a regular fashion this will result in the proper temporal locality properties. Our results show that this simple scaling approach is sufficient to maintain approximately the same stack distance distribution as in the single client case.

3.5 Performance Characteristics of SURGE Workloads

In this section we describe the original implementation of SURGE and the results of initial experiments performed using SURGE. For comparison purposes we also present results using SPECweb96. In Section 4.3 we describe a number of updates that we make to the original model which make SURGE more robust and more efficient.

3.5.1 Implementation and Validation of SURGE

SURGE is implemented in two parts: a set of programs written in C that precompute four datasets, and a multithreaded program written in Java that makes Web requests using the four datasets. The precomputed datasets consist of the sequence of requests to be made, the number of embedded files in each Web object to be requested, and the sequences of Active and Inactive OFF times to be inserted between requests. The implementation used Java for portability to a wide range of client machines.

Validation consisted of verifying that SURGE's output agreed with the six distributional models (file sizes, request sizes, popularity, embedded references, temporal locality, and OFF times). Results showed that the efficiency of the Java implementation affected the accuracy of OFF time generation. In particular, an interpreted implementation of Java introduced too much overhead to allow a close match to ideal OFF times; switching to a compiled implementation remedied the problem. In addition, we found that short runs (less than 15 minutes) did not always allow enough samples to be made of some distributions to reach acceptable match with the ideal cases; however runs of 30 minutes were generally adequate to achieve close fits between the ideal distributions and the measured results.

Of all the distributional models, only temporal locality was affected when SURGE was scaled up (run on larger numbers of host machines). As the number of hosts used by SURGE was increased, the resulting stack distance distributions always appeared to be lognormal, but their characteristics changed somewhat. This was expected since an exact match to temporal locality properties could not be guaranteed unless there was strict synchronization between all clients, which would have resulted in an unacceptably low maximum request rate.

3.5.2 Experimental Setup

The environment for these experiments consisted of six PCs running on a 100 Mbps network that could be isolated from other networks. The PCs were configured with 200MHz Pentium Pro CPUs and 32MB of RAM. SURGE clients (from 1 to 5 hosts) ran under Windows NT 4.0, while the server system (a single host) ran Apache v1.2.4 under Linux 2.0.

Since we are primarily interested in the behavior of the Web server and the network, it was important to prevent bottlenecks on the client from limiting overall system performance. We found that when more than approximately 50 UE threads are run on a client host, its resource limitations start to affect performance. As a result, we never run more than 50 UE threads on any host.

On the Web server, we examined CPU utilization and the number of active TCP connections; for the network, we measured the total number of both files and packets transferred over the run. The number of open connections on the server was sampled every 100ms and the CPU utilization was sampled every second. Both values were taken from the Linux `/proc` filesystem. Measurements of traffic on the network were taken using `tcpdump`.

In each of the experiments discussed in this section, the number of UEs is kept constant throughout the experiment. Thus we only explore the stationary behavior of the workload at this time. This allows a direct comparison with other workload generators, which also assume stationary conditions. However, there is no barrier to constructing a workload with varying numbers of UEs over time, such as would correspond to the case in which users visit the Web site for some period of time and then move on.

This assumption of stationarity in the workload for real servers probably only applies over short timespans during which the number of users at the Web server remains approximately constant. For this reason we restrict ourselves to runs no greater than 30 minutes. Note that even over intervals as short as these, Web servers can show high variability in many system metrics [85].

The workload generated by SPECweb96 depends on two user-defined parameters: a target number of HTTP operations per second, and the number of threads used to make requests. The general approach used by SPECweb96 is for each thread to generate HTTP requests at a constant rate; the request rate for each thread is determined by dividing the target rate by the number of threads. In all our experiments we used 16 threads, which was sufficient since the achieved operations per second were always close to the requested values.

Thus the intensity of the SPECweb96 workload is specified in terms of expected number of HTTP operations per second, while (as discussed in Section 3.2) the intensity of the SURGE workload is expressed in terms of user equivalents. To compare the effects of SURGE and SPECweb96, we empirically determined configurations of each resulting in approximately equal amounts of data

	SPECweb96			SURGE		
Nominal pkts/sec	HTTP ops/sec	Requests	TCP Packets	UEs	Requests	TCP Packets
70	3	5901	118560	50	5293	131642
300	14	26028	560238	150	26055	507727
500	25	46520	1000289	250	48238	874570

Table 3.2: Summary of Comparison Experiments.

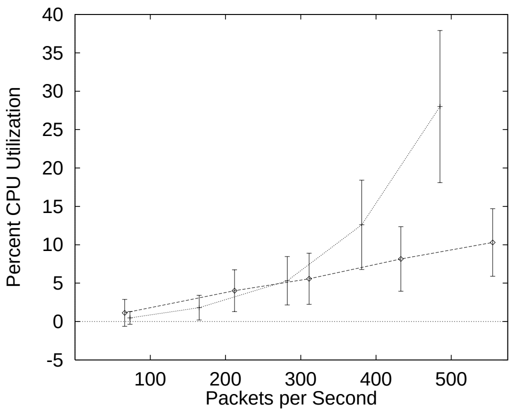


Figure 3.5: CPU Utilization for SURGE (upper at right) and SPECweb96 (lower at right) Workloads

transferred during a 30 minute run. We identified three such configuration levels of each workload generator; for each level, the difference between the two in terms of data transferred is less than about 20%. These configuration levels are used only for general comparisons between the two workloads. Table 3.2 summarizes the SURGE and SPECweb96 workloads used, showing the number of requests satisfied and the number of TCP packets transferred in each 30 minute run.

3.5.3 Results

We first consider the differences between SURGE and SPECweb96 in terms of their impact on the server; then we examine their different impacts on the network.

3.5.3.0.1 Server Impacts. The most immediately noticeable difference between the effects of the SURGE and SPECweb96 workloads is in server CPU utilization. Figure 3.5 shows plots of mean CPU utilization for the two workloads as a function of average packets transferred per second. Errorbars show one sample standard deviation above and below the mean. Mean and standard deviation were measured over the steady state of each experiment, with the startup transient removed.

Figure 3.5 shows that when the average transfer rate approaches 500 packets per second, the

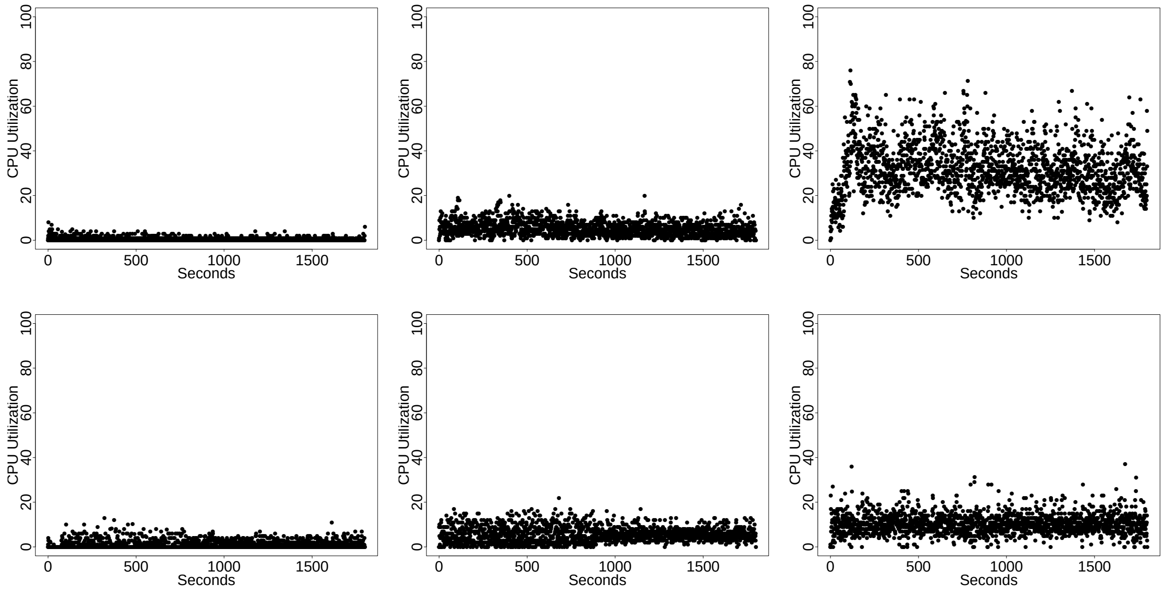


Figure 3.6: CPU Utilization for 70 pps (left), 300 pps (middle), and 500 pps (right) (upper: SURGE; lower: SPECweb96).

SURGE workload causes a sharp increase in the server's CPU load, while the server's CPU is not very heavily loaded under the SPECweb96 workload.

The difference in CPU load over time is shown in Figure 3.6. This figure plots instantaneous CPU load over the entire course of each of three experiments for both workloads. This figure shows how the two workloads are quite different in practice. The figure shows that SURGE results in a highly varying CPU load with utilization as high as 76% for transfer rates of about 500 pps. In contrast, the CPU load is relatively stable and never goes above 37% for the SPECweb96 workload.

One reason for the difference in CPU demands of the two workloads can be seen by examining the number of active TCP connections in each case. Table 3.3 shows the mean and standard deviation of the number of connections open on the server measured at 100ms intervals. This table shows that a large number of connections are typically open when running the SURGE workload; in contrast, the number of open connections for the SPECweb96 workload is typically very small. In addition, the variability in number of connections is much greater for the SURGE workload than for the SPECweb96 workload.

Figure 3.7 shows the number of open connections at 100ms intervals for the three experiments. Note that figures in the two rows use different scales on the y axis. This figure shows the large fluctuations in the number of open connections for the SURGE workload.

Maintaining and servicing a large number of open connections is computationally expensive, as pointed out in [85]. Clearly the difference in the number of active connections is likely to be a contributing factor in the CPU utilization differences seen between the two workloads (Figure 3.5).

The reason that SURGE maintains a much greater number of open server connections on average can be understood by comparing the request generation process in SURGE with that used

	SPECweb96		SURGE	
Nominal pps	Mean	Standard Deviation	Mean	Standard Deviation
70	0.028	0.18	13.9	3.92
300	0.37	0.69	33.2	12.1
500	0.71	1.41	67.1	35.3

Table 3.3: Active TCP Connections

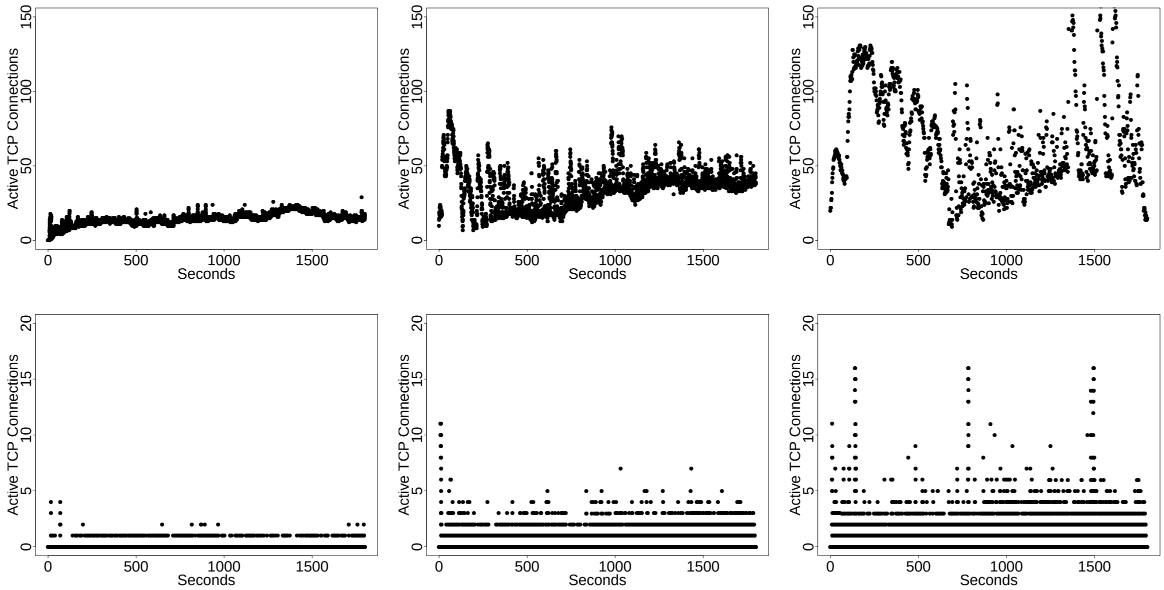


Figure 3.7: Active TCP Connections for 70 pps (left), 300 pps (middle), and 500 pps (right) (upper: SURGE; lower: SPECweb96).

in SPECweb96. The inter-request time in SPECweb96 is determined by the goal of generating a fixed number of requests per time interval. Thus for SPECweb96, as the number of requests per second increases, each thread's idle time between requests decreases. In contrast, for SURGE, as the number of requests per second increases, the idle times of individual threads do not decrease; instead more threads are used.

The result is that in SPECweb96, connections are multiplexed onto a relatively small number of threads, and so the number of simultaneous connections is low. On the other hand, under SURGE the number of threads grows in proportion to the intensity of the workload, which results in a corresponding increase in the number of simultaneous connections that are possible. When connections happen to overlap in large numbers the per-connection rate is slowed, and so in SURGE connections stay open for much longer periods on average than they do under SPECweb96.

In the case of SPECweb96, this difference could be addressed by scaling the number of connections in proportion to the requested operation rate, although it is apparently not customary to do so in Web server benchmarking. However, this effect highlights the drawbacks of the approach commonly used in Web workload generators of employing a small or fixed number of threads to make requests.

3.5.3.0.2 Network Impacts. Finally we consider differences in network traffic generated by the two workloads. As described in Section 3.1 we are particularly interested in whether the two workloads generate traffic that is self-similar [66] since this property has been shown to be present in Web traffic [28] and has significant implications for network performance [34, 101].

Self-similarity in the network traffic context refers to the scaling of variability (burstiness). A timeseries $X_t, t = 1, 2, ...$ is said to be *exactly second-order self-similar* if

$$X_t \stackrel{d}{=} m^{-H} \sum_{i=m(t-1)+1}^{mt} X_i \qquad \text{for } 1/2 < H < 1 \text{ and all } m > 0$$

where $\stackrel{d}{=}$ means equality in distribution. This definition suggests a simple test for self-similarity in network traffic, called the variance-time plot. This test plots the variance of $\sum_{i=m(t-1)+1}^{mt} X_i$ against m on log-log axes, where the X_is are measurements of traffic in bytes or packets per unit time. Linear behavior with slope greater than $-1/2$ is suggestive of nontrivial self-similarity.

Variance-time plots for SURGE and SPECweb96 traffic are shown in Figure 3.8. As the amount of data transferred increases from left to right in the figures, the traffic generated by SURGE shows roughly linear behavior with slope different from $-1/2$. That is, it continues to demonstrate burstiness across all time scales. In contrast, the traffic generated by SPECweb96 shows evidence of self similarity when the traffic intensity is low but as traffic increases, its self-similarity disappears: the variance-time plot approaches a slope of $-1/2$.

This effect can be understood by analyzing each of the two cases as a collection of individual sources. Consider a thread as an ON/OFF source; then its marginal distribution is that of a Bernoulli random variable. Its variance is maximized when the two states are equiprobable; when one state starts to dominate the other, the thread's variability in demand declines.

As the workload intensity increases, OFF times of individual threads under SPECweb96 grow shorter, so individual threads exhibit lower variability in demand. Threads increasingly approximate an always-ON source. Thus the traffic generated by SPECweb96 comes from a constant

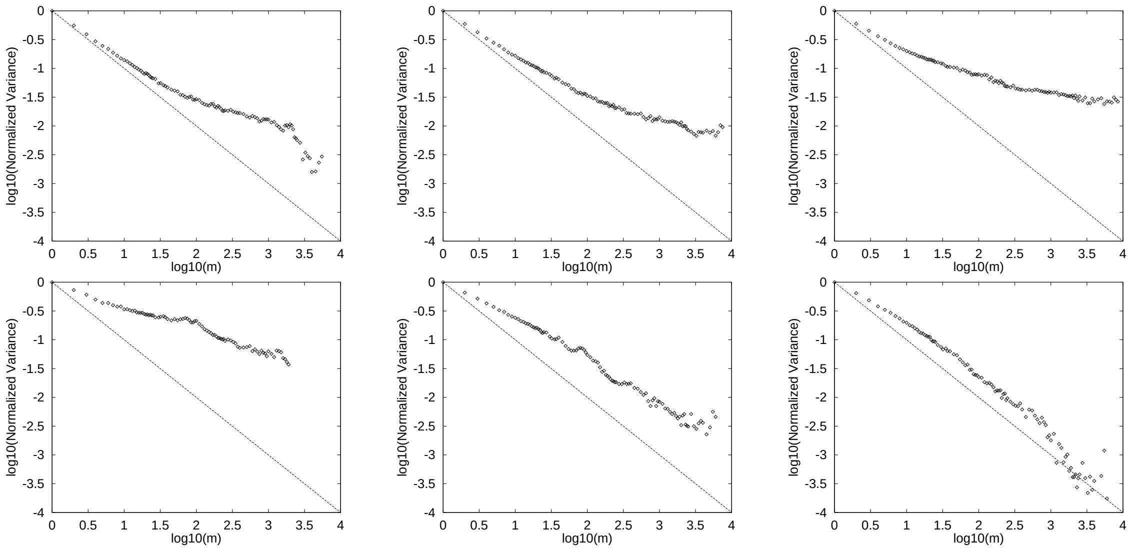


Figure 3.8: Variance-Time Plots for SURGE traffic (upper) and SPECweb96 traffic (lower). Left: 70 pps, Middle: 300 pps, Right: 500 pps.

number of sources which each decrease their variability as the workload intensity increases. Hence variability in the resulting aggregate traffic is reduced as well.

In contrast, the traffic generated by SURGE comes from constant-variability sources; it is the number of such sources that increases as workload intensity increases. Theoretical work [129] has shown that when transfer durations are heavy-tailed (as they are in SURGE) this condition is sufficient for the generation of self-similar traffic. As a result we expect that SURGE will generally produce self-similar network traffic under conditions of both high and low workload intensity.

Note that in this respect, the source behavior of SPECweb96 is similar to other Web workload generators whose general goals are to make requests from servers as quickly as possible [95, 123]. Since these workload generators do not preserve significant burstiness in individual thread demands, it is unlikely that under heavy loads they will produce self-similar traffic in practice.

Chapter 4

A Performance Evaluation of HTTP/1.0 and HTTP/1.1

4.1 Overview

Beyond a Web client's immediate connection to the Internet there are essentially two possible sources of latency in a Web transaction: the network and the server. Bottlenecks in the network exist due to congestion and/or limited bandwidth links along the path from client to server. Bottlenecks at the server occur when the server is heavily utilized either in the CPU or the disk system. When developing methods for improving performance in the Web, it is important to consider any potential solution from the perspectives of the client, network and server.

Initial work on improving Web performance focused on delays in the network as the principal source of latency for Web clients [84, 99]. Based on this premise, these studies proposed a number of enhancements to the HTTP 1.0 standard [19] that decrease latency by reducing the number of network round trips required to transfer Web objects during a browsing session (A *Web object* consists of an HTML file along with all the files that are embedded in it by reference, and a *browsing session* is a period during which Web objects are transferred with intervening idle periods (OFF times).). We consider the effects of two of these enhancements: the use of *persistent connections* and *pipelining*. A persistent connection is a single TCP connection that is used to transfer all files during a browsing session. Pipelining allows a collection of requests to be sent to a server without waiting for a response between requests, and allows the server to respond with a minimum number of data packets. These enhancements, along with others, paved the way for the HTTP/1.1 specification which was proposed as an IETF standard in 1997 [39].

Initial evaluation of HTTP/1.1 indicated that under certain conditions it can reduce both client latency and network traffic [48]. However, the enhancements in HTTP/1.1 come at a cost. Because connections are persistent, servers must typically manage a much larger number of open connections. The work in [84] points out that requiring the server to manage many open connections could increase its processing and memory load. In general, however, a comprehensive understanding of the implications of the new features in HTTP/1.1 for servers has not been available. In particular, the effects of bottlenecks other than the network on HTTP performance have only recently begun to be studied [15].

In this Chapter, we evaluate the impacts of using HTTP/1.0 and HTTP/1.1 when the server CPU or server disk is the bottleneck resource in the system. We also analyze in detail the network demands of each protocol. We run tests using two popular Web servers: Apache v1.3.0 running on Linux and Internet Information Server (IIS) v4.0 running on Windows NT. We use SURGE to create realistic Web workloads in a local area network environment. We analyze Web system performance by measuring throughput at the server and file latency at the client. We take detailed measurements of server subsystem (CPU, memory, disk, network) performance in order to determine where bottlenecks occur and to understand their effects. We use packet traces to analyze the network impact of our tests.

In order to use SURGE in this project, a number of enhancements to support HTTP/1.1 were added. Persistent connections and pipelining were added to the request generator per RFC 2068 [39]. The distributional model for file sizes was extended and a distributional model for the number of files requested during a browsing session was added. Details of the distributional models used in SURGE are given in Section 4.3.

We exercise each server using a range of workloads that enable us to drive the server to overload and thereby expose system bottlenecks. Our results indicate that:

1. When the server is not the bottleneck, the network demands of HTTP/1.1 are somewhat lower than those of HTTP/1.0 in our LAN test configuration. Our results suggest that in a WAN environment, these differences could be significant.

2. When the server CPU is the bottleneck resource, there is no significant difference in performance between HTTP/1.0 and HTTP/1.1.

3. When memory on the server is limited and the disk system becomes the bottleneck resource, HTTP/1.1 performs significantly worse than HTTP/1.0.

The insight gained in our experiments led us to conclude that there is a gradient along which Web performance must be considered. If a server is not heavily loaded or if the CPU or the network is the bottleneck, the features in HTTP/1.1 are desirable due to reduction in packet traffic. However, if a server's disk system is the bottleneck, the persistent connections that HTTP/1.1 maintains between Web object transfers cause severe degradation in server throughput. Based on this observation, we propose and evaluate a connection management policy for HTTP/1.1 in which clients proactively close connections after a Web object has been transferred. This simple policy greatly improves server throughput when the disk system is the bottleneck and still reduces overall packet traffic versus HTTP/1.0.

4.2 Experimental Design

Our experiments were conducted in an environment consisting of three PC's connected via a 100Mbps network. This was a stand alone local area network with no intervening external traffic. The PC's were configured with 200Mhz Pentium Pro CPU's with 128MB of RAM on the two SURGE client systems and between 32MB and 256MB of RAM on the server system. The SURGE client systems ran Linux 2.0.30. The server ran either Apache v1.3.0 on Linux 2.0.30, or Microsoft IIS v4.0 Windows NT Server 4.0. These servers were selected because of their popularity [90] and

Abbr	Webmonitor Value	Perfmon Value
USR	% CPU server processes	% CPU server processes
KER	% CPU kernel processes	% CPU kernel processes
IDL	% CPU idle process	% CPU idle process
PGR	Pages read from disk	Pages read from disk
PGW	Pages written to disk	Pages written to disk
TPS	TCP packets sent	N/A
TPR	TCP packets received	N/A
BYS	N/A	Bytes sent by server
BYR	N/A	Bytes received by server

Table 4.1: Detailed Performance Measurement Categories

because they are based on a different implementation models. Apache creates a separate process for each HTTP connection, while IIS uses a lighter weight threaded model which requires less system resources for each HTTP connection. Details of the configurations for each of the servers are given in Section 4.3.3 and 4.3.2.

We measured the average throughput on the server system in terms of HTTP GET operations initiated per second. On the SURGE client systems we measured the average latency for each HTTP GET. Detailed performance measurements of the servers were taken using Webmonitor (under Linux) [7], and PerfMon (under NT) with samples taken once per second during tests. When packet traces were taken, `tcpdump` was run on the client systems.

Each experiment exercised the server over a range of loads between 20 and 640 UE's. The heavy load test levels were able to push the server into an overload state. Each experiment was run for ten minutes which was a sufficient amount of time to stabilize the subsequent measurements. SURGE was configured with 2000 unique files, resulting in about 50MB of data on the server.

Experiments were divided into three groups. The first set measures and compares the details of the network effects of each protocol using TCP packet traces. It is important to note that the development of HTTP/1.1 was based on the assumption of delays in a *wide area* network. In that environment, round trip times between clients and server can be long (hundreds of milliseconds). Our tests were run in a local area environment which means that file transfer times and connection durations tend to be much shorter on average.

The second set of experiments compares the performance of HTTP/1.0 and HTTP/1.1 when the server's CPU is the bottleneck. We increased the amount of memory on the server by powers of two until the measured performance did not change. Once this threshold size was determined, we took detailed system performance measurements using Webmonitor and Perfmon to verify that the CPU was the system bottleneck. The list of system characteristics which were tracked are given in Table 4.1. In these experiments we measure performance at three different load levels (low, medium and high).

The third set of experiments was designed to explore the performance differences between HTTP/1.0 and HTTP/1.1 when the disk system was the bottleneck resource on the server. To create a bottleneck at the disk, it suffices to reduce the amount of RAM available. This forces paging under heavy load since not all data files can be cached in main memory. In these experiments, we measure the latency and throughput for both Apache and IIS when memory size ranges between 32MB and 256MB. In each case the server is driven to overload.

4.3 SURGE Models and Configuration

4.3.1 SURGE Model Enhancements

SURGE was updated with support for HTTP/1.1 for this work. Specifically, persistent connections and pipelining were added in order to evaluate their effects of HTTP/1.1. The distributional model for file sizes was extended in order to more accurately represent Web objects. A distributional model for the number of files requested during a browsing session was added in order to enable a *session* model. Finally, Active OFF times were removed from SURGE since this characteristic is client system dependant. Another important update was that SURGE was rewritten in C due to the performance limitations of the original Java implementation. As was shown in Chapter 3, five client systems were necessary to drive a server to approximately 30% CPU utilization. The C implementation enables two client systems to drive a server to overload. This revision of SURGE was used in tests throughout the remainder of the work presented in this thesis.

Details of the updates to the SURGE model include:

- **File sizes.** File sizes refer to the sizes of files which make up the test set on the Web server. For this study, file sizes are divided into three categories in order to more accurately represent Web objects. Base files refer to HTML files which contain embedded files. Embedded files refer to files which are referenced by base files. Single files are files which are neither base nor embedded. Both base files and single files are modeled with hybrid distributions.

- **Session lengths.** Session lengths refer to the number of Web objects a user will request from a particular Web server before moving onto another Web server (*i.e.* during a browsing session). Accurate modeling of this distribution is important in HTTP/1.1 tests since it determines the number of Web objects which will be requested for the duration of a persistent connection. This distribution is a new addition to SURGE for the purpose of testing HTTP/1.1. Data for this characteristic is based on the work in [52].

A summary of the model distributions and parameters used in SURGE is given in Table 4.2.

4.3.2 Apache Configuration Parameters

Apache v1.3.0 was configured to be as efficient as possible for the tests in this study. The configuration used was based on the highperformance.conf file which is supplied with the distribution. The specific parameters used are given in Table 4.3.

4.3.3 IIS/NT Configuration Parameters

IIS was configured to be as efficient as possible for the tests in this study. The configuration used was published in [83] and is given in Table 4.4.

Component	Model	Probability Density Function	Parameters
File Sizes, Base File Body	Lognormal	$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-(\ln x - \mu)^2/2\sigma^2}$	$\mu = 7.630$; $\sigma = 1.001$
File Sizes, Base File Tail	Pareto	$p(x) = \alpha k^{\alpha} x^{-(\alpha+1)}$	$k = 10K$; $\alpha = 1.0$
File Sizes, Embedded File	Lognormal	$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-(\ln x - \mu)^2/2\sigma^2}$	$\mu = 8.215$; $\sigma = 1.460$
File Sizes, Single File1	Lognormal	$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-(\ln x - \mu)^2/2\sigma^2}$	$\mu = 7.101$; $\sigma = 1.200$
File Sizes, Single File2	Lognormal	$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-(\ln x - \mu)^2/2\sigma^2}$	$\mu = 11.151$; $\sigma = 1.143$
Request Sizes, Body	Lognormal	$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-(\ln x - \mu)^2/2\sigma^2}$	$\mu = 7.881$; $\sigma = 1.339$
Request Sizes, Tail	Pareto	$p(x) = \alpha k^{\alpha} x^{-(\alpha+1)}$	$k = 34102$; $\alpha = 1.177$
Popularity	Zipf		
Temporal Locality	Lognormal	$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-(\ln x - \mu)^2/2\sigma^2}$	$\mu = 1.5$; $\sigma = 0.80$
OFF Times	Pareto	$p(x) = \alpha k^{\alpha} x^{-(\alpha+1)}$	$k = 1$; $\alpha = 1.4$
Embedded References	Pareto	$p(x) = \alpha k^{\alpha} x^{-(\alpha+1)}$	$k = 2$; $\alpha = 1.245$
Session Length	Inverse Gaussian	$p(x) = \sqrt{\lambda/2x^3\pi} e^{-\lambda(x-\mu)^2/2x\mu^2}$	$\mu = 3.86$; $\lambda = 6.08$

Table 4.2: Summary of SURGE Configuration Parameters used in tests

Parameter	**Value**
KeepAlive	On
MaxKeepAliveRequests	150
KeepAliveTimeout	15
MinSpareServers	5
MaxSpareServers	10
StartServers	5
MaxClients	900
MaxRequestsPerChild	10,000,000

Table 4.3: Apache v1.3.0 Configuration Parameters

Feature	Configuration
Operating System	Windows NT Server v4.0; Service Pack 3 NT; Windows NT Option Pack updates: w3svc (vesion 4.02.0636) 12/97 TCP/IP and AFD Transport hotfix IIS 4.0 logfix 4/98 NDIS hotfix 12/97
Tuning	Performance set to maximize file sharing; foreground application boost set to NONE; ran chkdsk /l:65536 on disk to increase the log size. The following registry settings in HKEY_LOCAL_MACHINE,SYSTEM,CurrentControlSet,Services : Tcpip,Parameters,MaxFreeTcbs 72000 Tcpip,Parameters,MaxHashTableSize 65536 Tcpip,Parameters,TcpTimedWaitDelay 60 Tcpip,Parameters,TcpWindowSize 17520
Web Server	Internet Information Server 4.0
Tuning	Performance set to handle over 100,000 hits per day; removed all mappings except asp., connection close set to 15 seconds The following registry settings in HKEY_LOCAL_MACHINE,SYSTEM,CurrentControlSet,Services : InetInfo,Parameters,ListenBackLog 250 InetInfo,Parameters,MaxPoolThreads 2 InetInfo,Parameters,MemoryCacheSize 10,000,000 InetInfo,Parameters,ObjectCacheTTL 0xffffffff InetInfo,Parameters,OpenFileInCache 7000 InetInfo,Parameters,PoolThreadLimit 2

Table 4.4: Windows NT 4.0/IIS v4.0 Configuration Parameters

4.4 Experimental Results

4.4.0.1 Network Performance

Our first set of experiments compares the network demands of HTTP/1.0 and HTTP/1.1. In order to understand the effects of both persistent connections and pipelining we capture traces of all packets transferred during tests. The network effects of persistent connections and pipelining can be expressed in terms of their impact on the quantity of each type of packet (SYN, FIN, ACK, DATA) sent during the test. Figure 4.1 shows the relative differences under three different client loads. The figure shows that there is a significant reduction in the number of SYN, FIN and ACK packets between HTTP/1.0 and HTTP/1.1 reflecting the benefit of persistent connections. However, the figure also shows that there is only a small change in the number of DATA packets transferred during tests. This is important because SYN, FIN and ACK packets are typically small in size when compared to DATA packets thus the total byte traffic is not greatly affected by the change in protocol. This is shown quantitatively in the first four rows of Table 4.5.

Evaluation of the effects of pipelining can also be seen in Table 4.5. The last row in the table shows that the average number of data bytes per packet remains relatively constant across loads and protocols, indicating that there is no significant impact due to pipelining. This result differs from results reported by Nielsen *et al.* [48]. This difference is due to the distributional model for Web object sizes used in our experiments. Object sizes in our study are based on an empirically derived model with a mean of 3.3 files per object and a median of 1.0 file per object. The object transferred in the Nielsen *et al.* study was composed of 43 files. We consider this size to be much larger than what is seen in typical Web objects. A similar observation is made in [67].

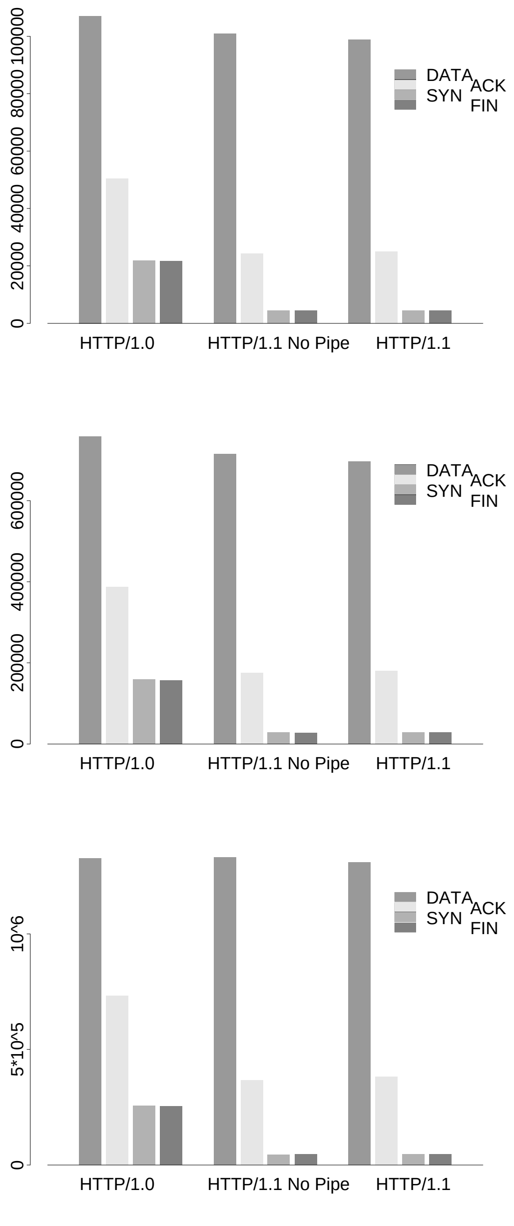


Figure 4.1: Counts of the total number of packets transferred of each type for 40 UE load (top) 280 UE load (middle) and 520 UE load (bottom) in Apache v1.3.0 and 128MB

Type	HTTP/1.0			HTTP/1.1 No Pipelining			HTTP/1.1 Pipelining		
	40 UE	280 UE	520 UE	40 UE	280 UE	520 UE	40 UE	280 UE	520 UE
Data Packets	0.107	0.758	1.327	0.101	0.716	1.330	0.099	0.697	1.308
Total Packets	0.201	1.462	2.571	0.134	0.947	1.784	0.133	0.935	1.784
Total Data Bytes	128.2	897.7	1,602.9	119.0	842.8	1,607.4	118.6	836.9	1,608.7
Total Bytes	136.2	956.2	1,705.7	124.4	880.7	1,678.8	123.9	874.3	1,680.0
Data Byts/Data Pkts	1,197	1,184	1,207	1,180	1,178	1,208	1,198	1,200	1,229

Table 4.5: Packet and Byte Counts for Apache v1.3.0 with 128MB RAM (Counts in millions of packets or bytes)

	HTTP/1.0			HTTP/1.1 No Pipelining			HTTP/1.1 Pipelining		
	40 UE	280 UE	520 UE	40 UE	280 UE	520 UE	40 UE	280 UE	520 UE
CWND Average	5.79	4.04	3.81	10.48	10.79	9.28	11.04	11.30	9.72
CWND Max	29	83	99	68	186	150	106	202	141

Table 4.6: Congestion window comparison for Apache v1.3.0 with 128MB RAM

As we show in the next section, there is very little performance difference between HTTP/1.0 and HTTP/1.1 in the absence of a bottleneck in the disk. However, these results may be due to the use of a LAN for testing. Performance in a WAN would be strongly affected by the value of the server's congestion window[1] [58]. For this reason, we examined how the TCP congestion window (CWND) size differed between the two protocols. In the absence of congestion in the network, persistent connections should enable the server to avoid TCP slow start for successive file transfers by keeping CWND large. This means that HTTP/1.1 should be more efficient in using available network bandwidth. In the presence of congestion in the network, persistent connections should more consistently keep CWND in its congestion avoidance mode. This means HTTP/1.1 should be more efficient when network bandwidth is limited as well.

We tracked the CWND size with an instrumented version of Linux 2.0.30. The modifications of Linux simply made the CWND size for each active TCP connection available for sampling. The mean (per second) and maximum CWND sizes under low, medium and high client loads are given in Table 4.6. The table shows that the CWND size is much larger on average in HTTP/1.1 which should have a beneficial effect on latency reduction in the wide area. In the switched 100Mbps local area environment in which the tests are run, there is no packet loss due to congestion. Thus, CWND will typically grow exponentially through the slow start threshold and then grow linearly until the connection is closed.

The distribution of CWND sizes under each protocol and load level can be seen in the cumulative distribution function (CDF) plots in Figure 4.2. The plots illustrate that there is relatively little difference between pipelined and non-pipelined HTTP/1.1 but a significant difference between HTTP/1.0 and HTTP/1.1. The plots show that under heavy load, nearly 80% of all HTTP/1.0 connections have a CWND size of 1 or 2 which means that most of the connections have just opened and have not yet gone through slow start. The plots also show that over 50% of the HTTP/1.1 have a CWND size above 7 indicating that in a WAN, much higher average transfer rates should be achievable under HTTP/1.1.

4.4.0.2 CPU Constrained Performance

In this section we examine the case in which the CPU is the bottleneck in the system's performance. We ran performance tests with the server configured with sufficient RAM to contain the Web server, operating system and all data files in main memory. Our test file set consists of approximately 50MB of data; Apache on Linux 2.0.30 uses approximately 14MB while IIS on NT 4.0 uses approximately 30MB. The server for this set of experiments was configured with 128MB of memory.

Performance monitoring data gives us better insight into the details of system utilization in this configuration. The performance monitors enable us to track the low level system components on the servers that are listed in Table 4.1. The results of the detailed system performance measure-

[1]The TCP congestion window (CWND) is a means for throttling the flow of TCP packets in the presence of network congestion. The size of CWND begins small but grows exponentially as ACK's are received until a threshold size is reached. This mode of operation is called *slow start*. Slow start keeps the initial packet sending rate low in order to avoid saturating the network. Beyond the slow start threshold, CWND enters its *congestion avoidance* mode where it grows linearly. CWND will grow in either mode until a packet loss is detected. When a packet is lost, CWND size is dramatically reduced in order to ease network congestion.

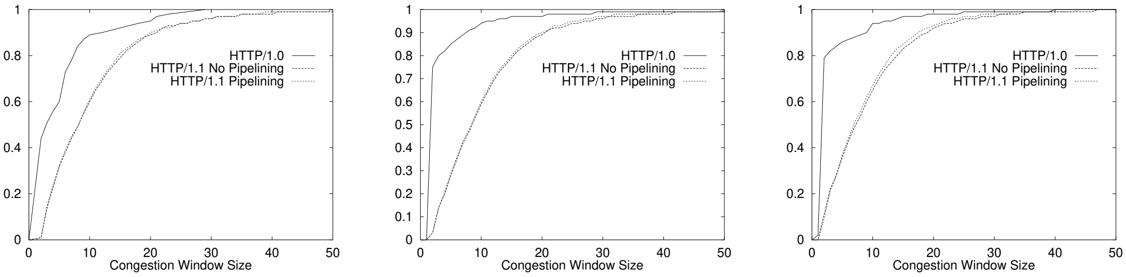


Figure 4.2: Cumulative Distribution Function plots of congestion window sizes for 40 UE load (left) 280 UE load (center) and 520 UE load (right) in Apache v1.3.0 and 128MB

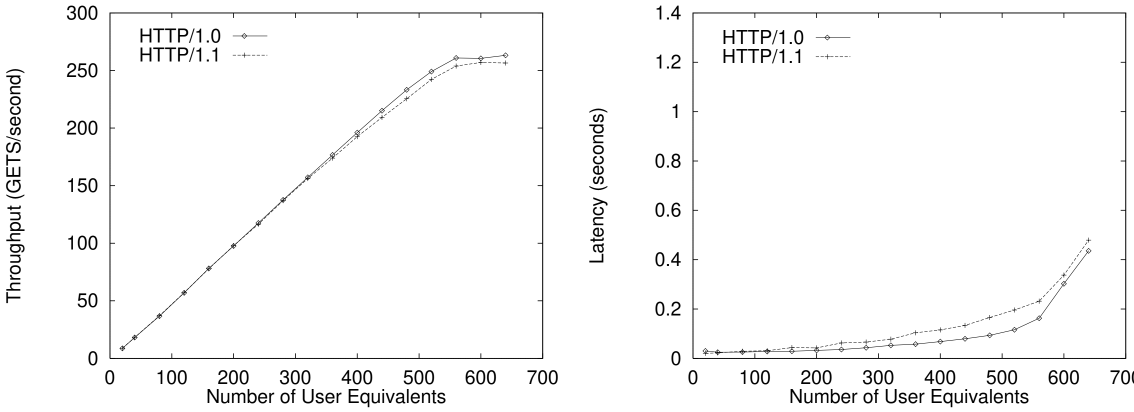


Figure 4.3: Throughput (left) and latency (right) for Apache v1.3.0 with 128MB RAM

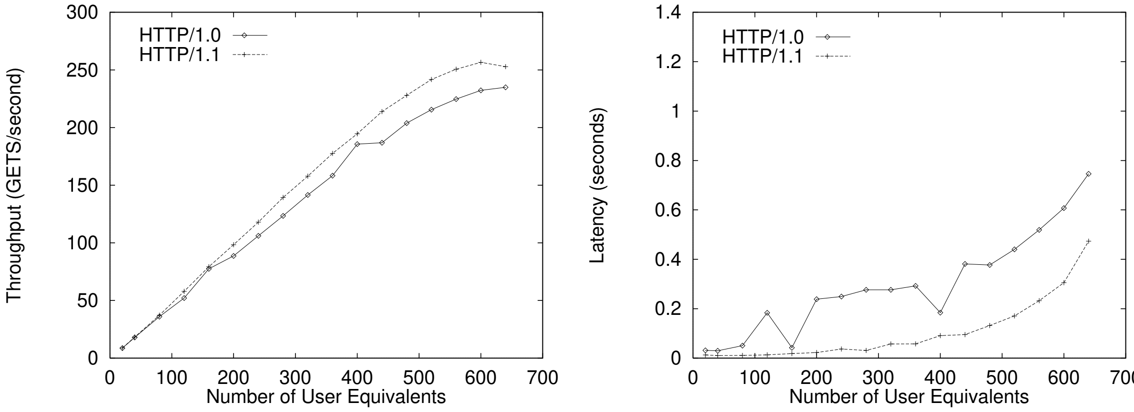


Figure 4.4: Throughput (left) and latency (right) for IIS v4.0 with 128MB RAM

Value	HTTP/1.0			HTTP/1.1		
	40 UE	280 UE	520 UE	40 UE	280 UE	520 UE
USR	1.66	15.18	27.15	1.33	13.10	20.84
KER	7.89	41.95	69.96	6.90	43.70	72.02
IDL	90.45	42.87	2.89	91.77	43.20	7.14
PGR	0.00	0.00	0.01	0.02	0.02	0.02
PGW	11.16	33.74	48.93	12.44	36.73	46.38
TPS	233	1,859	3,515	188	1,382	2,135
TPR	125	944	1,745	66	441	710

Table 4.7: Detailed Performance Measurements for Apache v1.3.0 with 128MB RAM

Value	HTTP/1.0			HTTP/1.1		
	40 UE	280 UE	520 UE	40 UE	280 UE	520 UE
USR	1.07	4.70	7.32	1.10	4.61	7.59
KER	6.77	46.45	78.26	6.46	46.85	80.92
IDL	92.16	48.85	14.42	92.44	48.54	11.49
PGR	0.00	0.00	0.00	0.00	0.00	0.00
PGW	0.00	0.00	0.00	0.00	0.00	0.00
BYS	232,100	1,829,474	2,967,467	230,295	1,840,909	3,177,293
BYR	1,045	7,832	12,332	1,362	10,149	16,769

Table 4.8: Detailed Performance Measurements for IIS/NT with 128MB RAM

ments for Apache and IIS are given in Tables 4.7 and 4.8 respectively. Each test was run with a warm file cache, in this case achieved by running each test twice and taking measurements during the second test. This gives a clearer picture of the steady state operation of the server under each of the loads. The disk activity data shows that there is virtually no paging taking place during tests (except for writing, which is due to server log file generation) indicating that the disk system is not the bottleneck during tests. The network activity data shows a mean transfer rate of approximately 22Mbps for Apache and 25Mbps for IIS. These are well below the 100Mbps capacity of the network indicating that the network is not the bottleneck during tests. Furthermore, the very low CPU percent idle times indicate that the CPU is the limiting resource under the 520 UE load.

Performance results are shown over a range of client loads in Figure 4.3 for Apache and Figure 4.4 for IIS. The graphs indicate that for the Apache server, performance does not vary greatly between HTTP/1.0 and HTTP/1.1, and that for the IIS server, performance under HTTP/1.1 is only slightly better than under HTTP/1.0. These results show that the new features added to HTTP/1.1 do not significantly reduce the server's CPU processing load per byte transferred.

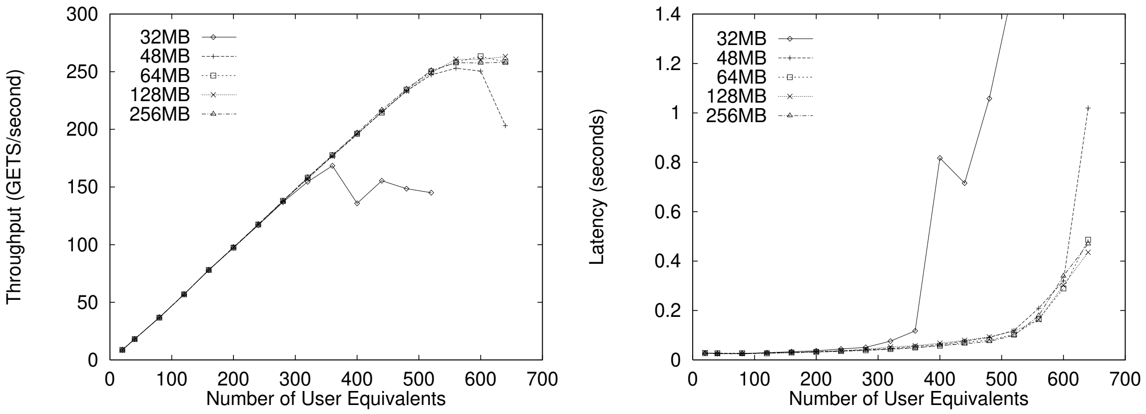


Figure 4.5: Throughput (left) and latency (right) for Apache v1.3.0 and HTTP/1.0

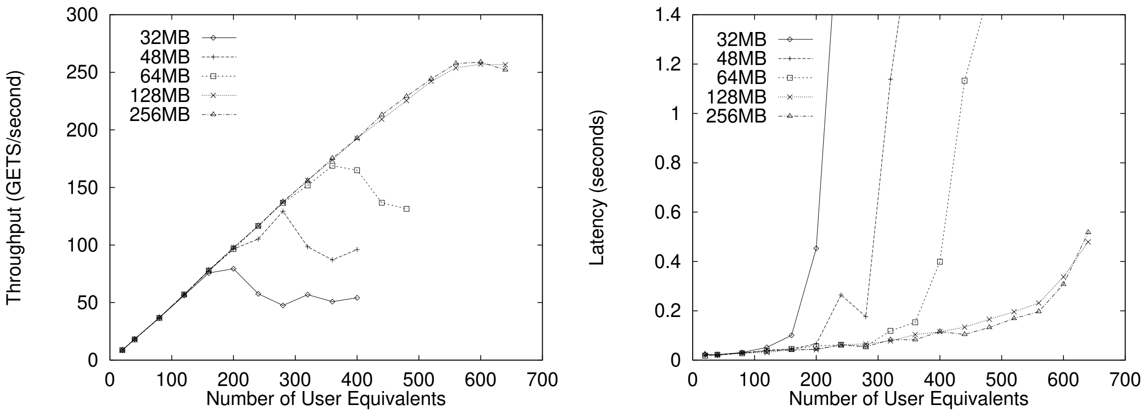


Figure 4.6: Throughput (left) and latency (right) for Apache v1.3.0 and HTTP/1.1

4.4.0.3 Disk Constrained Performance

As the amount of RAM on the server is reduced to a level where paging of data files becomes necessary, the disk system becomes the bottleneck in the system. Performance results are shown over a range of memory configurations and client loads, in Figure 4.5 for Apache and HTTP/1.0, in Figure 4.6 for Apache and HTTP/1.1, in Figure 4.7 for IIS and HTTP/1.0 and in Figure 4.8 for IIS and HTTP/1.1. The graphs show that performance degrades dramatically when memory on the server is limited. While this by itself may not be surprising, what is particularly interesting is that the degradation of performance is much more pronounced under HTTP/1.1 than under HTTP/1.0.

4.4.0.4 Memory Use and TCP Connections

The differences in performance between HTTP/1.0 and HTTP/1.1 when the disk system is the constraining resource can be understood through more detailed measurements of memory use. We measured the amount of memory used and the number of TCP connections in the ESTABLISHED

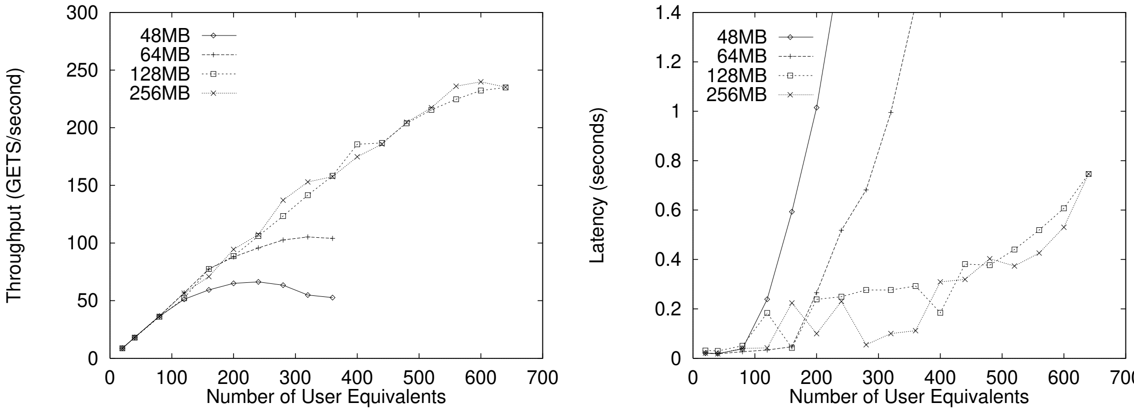


Figure 4.7: Throughput (left) and latency (right) for IIS v4.0 and HTTP/1.0

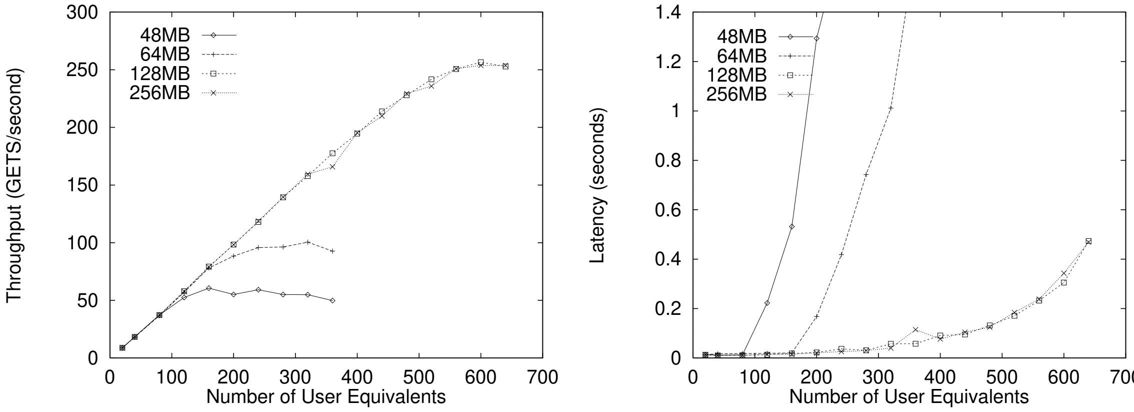


Figure 4.8: Throughput (left) and latency (right) for IIS v4.0 and HTTP/1.1

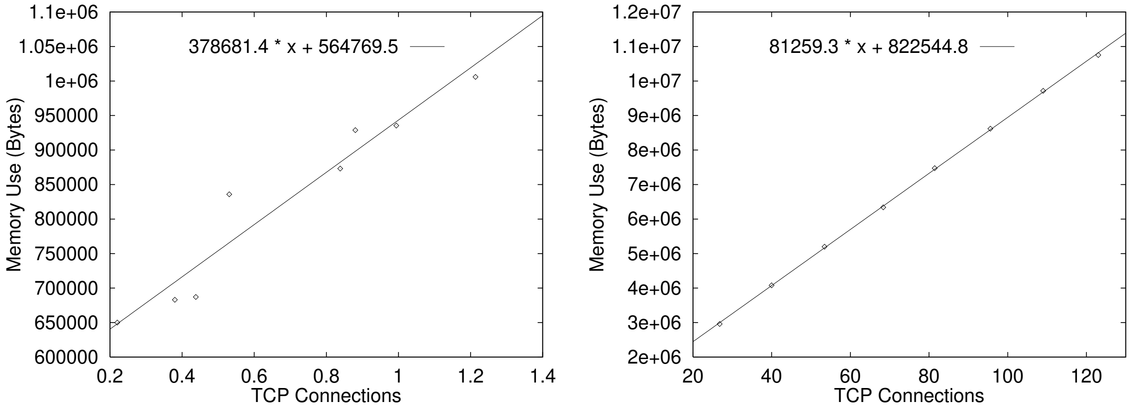


Figure 4.9: Memory use per TCP connection for HTTP/1.0 (left) and HTTP/1.1 (right) for Apache v1.3.0 and 128MB

state[1] for HTTP/1.0 and HTTP/1.1 on Apache under a range of low loads. (We attempted to take the same measurement for IIS; however, due to NT's methods of memory allocation, this was not possible.) We fit a least squares line to the points to determine the per connection memory use. The results on the left side of Figure 4.9 show the per connection memory cost under HTTP/1.0 to be approximately 378KB, while the plot on the right side of of Figure 4.9 shows the per connection memory cost under HTTP/1.1 to be about 81KB. These figures indicate that the memory needed by a connection in Apache is large; for an active connection (*i.e.* an HTTP/1.0 connection) the memory demand is four times higher than for a mostly-idle connection (*i.e.* an HTTP/1.1 connection).

These results are significant because when main memory is full, creating a new HTTP connection will require that data files be paged from memory; subsequent requests for these files will no longer hit in the cache and will require disk I/O. The mean file size for our data set was 26KB, and the median was 3KB. In the case of HTTP/1.1, this means that when memory is full, about 3 files on average must be paged out of memory in order to accommodate a new connection — and in the common case (files of median size or less), 25 files or more may be evicted in order to create a new connection.

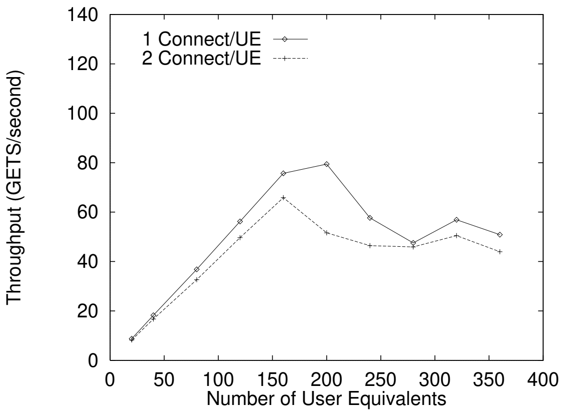
While these results show that Apache uses a large amount of memory per connection, obtaining corresponding results for the IIS server is more difficult (because precise memory allocation measurements in Windows NT are hard to obtain from the Perfmon tool). However, our initial investigations indicate that IIS uses less memory per connection, perhaps because of its thread-based implementation. This helps to explain why the performance degradation under HTTP/1.1 as compared to HTTP/1.0 is less pronounced in the IIS server (Figures 4.7 and 4.8).

Since connection effects are so important, we measured the number of TCP connections made under each protocol. Table 4.9 shows these measurements under three different load levels. The second line in the table shows the mean number of TCP connections in the ESTABLISHED state.

[1] A TCP connection is in the ESTABLISHED state after the three way packet exchange which sets up a connection has been completed and before a connection has been closed. Data can be transferred in either direction when a connection is in the ESTABLISHED state.

	HTTP/1.0			HTTP/1.1		
	40 UE	280 UE	520 UE	40 UE	280 UE	520 UE
Total Connects	10,902	83,826	153,231	2,278	15,194	26,456
Avg. ESTABLISHED Cncts,	0.16	3.47	23.82	35.41	234.11	398.05
Avg. Cnct, Duration (sec)	0.02	0.04	0.12	13.02	13.59	15.59
Connect Seconds	218	3,353	18,388	29,659	206,486	412,449

Table 4.9: TCP Connection Measurements for Apache v1.3.0 with 128MB RAM



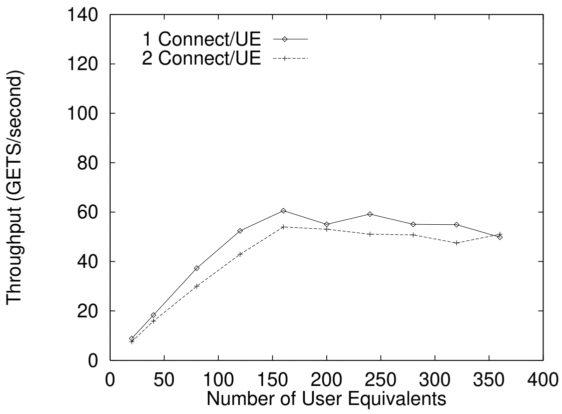


Figure 4.10: Effects of multiple connections for Apache v1.3.0 with 32MB (left) and IIS v4.0 with 48MB (right)

This shows that there are many more TCP connections made in HTTP/1.0 but that they are very short lived on average. The third line in the table, average connection duration, highlights the effect of persistent connections. Connection duration for HTTP/1.0 is simply the average file transfer latency, but connection duration for HTTP/1.1 includes both file transfer latency and idle ("OFF") periods. The table shows that connection duration in HTTP/1.1 is dominated by the OFF periods. The last line in the table is the product of lines one and three. It is the total number of connection seconds during the test. The connection seconds for HTTP/1.0 represent demands to transfer data, while this value for HTTP/1.1 includes open but idle connections. The ratio of these two values shows that for high loads, HTTP/1.1 can place more than *20 times* the connection load on the server that HTTP/1.0 does.

The per connection cost can also explored by configuring SURGE clients to open multiple connections per UE. In this experiment, two connections per UE are used by SURGE clients to transfer files when a Web object has more than one embedded file. Figure 4.10 shows the impact on throughput for both Apache and IIS when multiple connections are used under HTTP/1.1. In both cases throughput is decreased. Not only does this highlight the cost per connection issue, but it also suggests that multiple connections per client used as a default setting in some popular browsers may have a serious negative impact on server throughput.

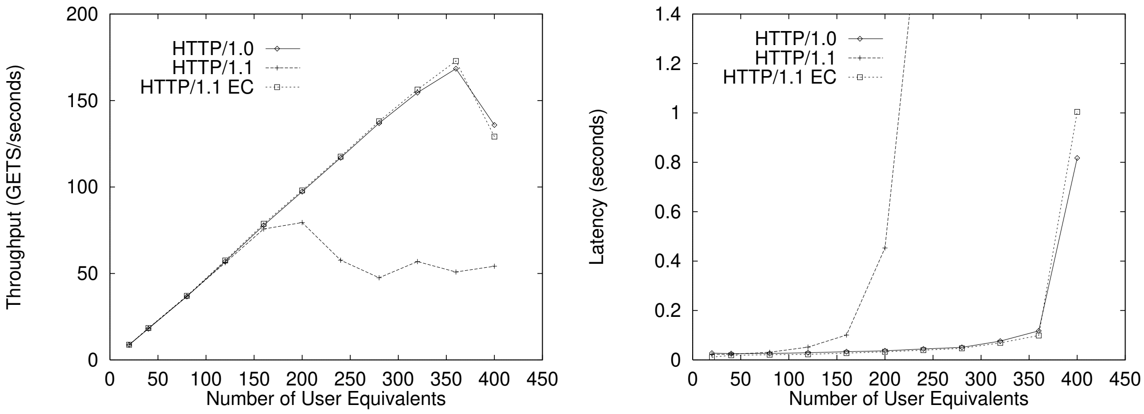


Figure 4.11: Throughput (left) and latency (right) for Apache v1.3.0 with 32MB memory and HTTP/1.1 with early close (EC)

4.4.0.5 HTTP/1.1 Connection Management

Analysis of the low utilization of most connections under HTTP/1.1 leads us to conclude that when a server is constrained by its disk system, the relative benefit of keeping connections open across successive Web object transfers is much smaller than the benefit of keeping the connection open during the transfer of a single object. Thus, an effective policy for managing connections may be to keep the persistent, pipelined connection open for the transfer of a Web object but to close the connection after each web object transfer. This enhancement is similar to the GETALL construct proposed in [99]. GETALL is a request sent by the client in which the server responds by sending an entire Web object. Our policy simply closes the connection after each GETALL. We call this policy *early close*.

If we consider that there are F files per object and N objects requested during a browsing session to a single server, then the effects of each protocol are as follows:

- HTTP/1.0 will make $F * N$ connections per client which are only open during the transfer of each file,
- HTTP/1.1 with early close will make N connections per client which are only open during the transfer of each Web object,
- Typical HTTP/1.1 will maintain 1 connection per client (note: servers are typically configured with a timeout on persistent connections, thus 1 connection per client is the best possible case), which is open during Web object transfers and OFF times.

We expect that HTTP/1.1 with early close should perform somewhere between HTTP/1.0 and standard HTTP/1.1 in terms of server throughput and network efficiency. The performance results of HTTP/1.1 with early close can be seen in Figures 4.11 and 4.12 for Apache under 32MB and 128MB memory configurations. The figures show that HTTP/1.1 with early close actually performs as well as HTTP/1.0 in terms of throughput in either configuration and much better than standard HTTP/1.1 in the disk constrained test.

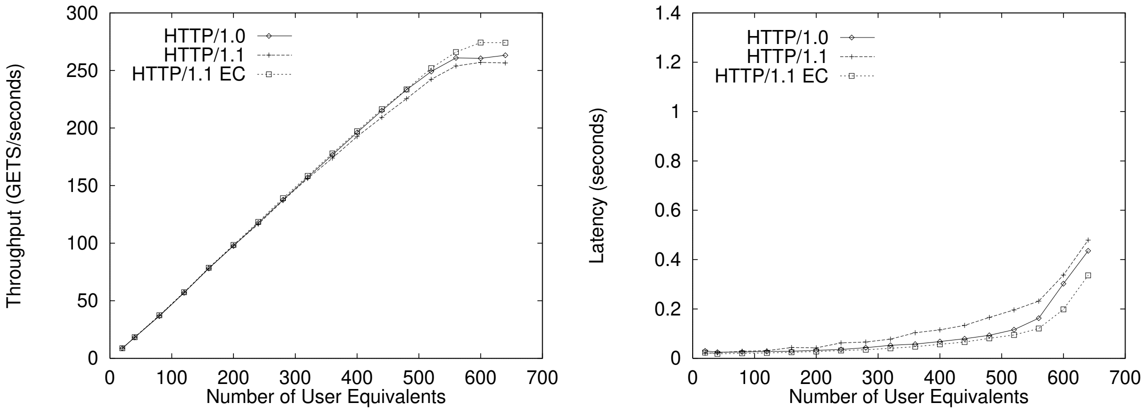


Figure 4.12: Throughput (left) and latency (right) for Apache v1.3.0 with 128MB memory and HTTP/1.1 with early close (EC)

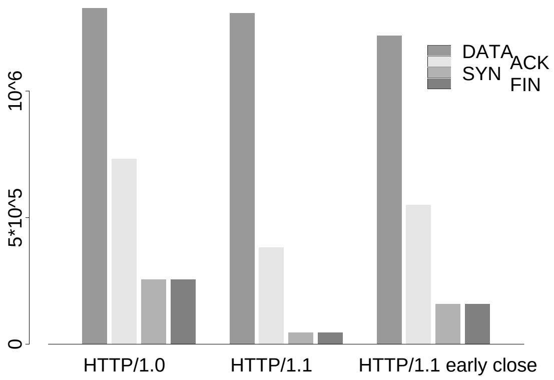


Figure 4.13: Counts of the total number of packets transferred of each type for 520 UE load in Apache v1.3.0 and 128MB using HTTP/1.1 with early close

The drawback of HTTP/1.1 with early close is that closing connections between Web object transfers will increase network traffic due to the extra connection startups. Details of the impact of HTTP/1.1 with early close on the network are given in Table 4.10. From this we can see that HTTP/1.1 with early close roughly falls in between HTTP/1.0 and standard HTTP/1.1 in terms of total connections opened during tests and that only 17% more packets are transferred versus standard HTTP/1.1. Details of the counts for each type of packet are compared in Figure 4.13. This figure again shows that HTTP/1.1 with early close falls roughly between HTTP/1.0 and standard HTTP/1.1 as expected.

Type	HTTP/1.0			HTTP/1.1 Pipelining			HTTP/1.1 early close		
	40 UE	280 UE	520 UE	40 UE	280 UE	520 UE	40 UE	280 UE	520 UE
Total Connects	10,902	83,826	153,231	2,278	15,194	26,456	7,876	54,667	96,776
CWND Average	5.79	4.04	3.81	11.04	11.30	9.72	4.87	5.40	4.39
CWND Max	29	83	99	106	202	141	35	233	182
Data Packets	0.107	0.758	1.327	0.099	0.697	1.309	0.105	0.703	1.220
Total Packets	0.201	1.462	2.571	0.133	0.935	1.784	0.176	1.180	2.089
Total Data Bytes	128.2	897.7	1,602.9	118.6	836.9	1,608.7	127.3	844.4	1,493.6
Total Bytes	136.2	956.2	1,705.7	123.9	874.3	1,680.0	134.3	891.6	1,577.1
Data Byts/Data Pkt	1,197	1,184	1,207	1,198	1,200	1,229	1,210	1,200	1,224

Table 4.10: Network impact and Packet/Byte Counts (in millions) for HTTP/1.1 with early close on Apache v1.3.0 with 128MB RAM

Chapter 5

Wide Area Web Measurement Infrastructure

5.1 Overview

Given our ability to generate representative workloads as demonstrated in the last two chapters, we now trun our attention to creating an infrastructure in which we can take measurements of Web transactions in the wide area. This chapter describes the details of both a prototype implementation (WAWM v1.0) and a larger, second generation of the WAWM measurement infrastructure (WAWM v2.0). We describe the physical infrastructure, our measurement methods, our experiences with the NISTnet [128] wide area emulator and our experiences with GPS timers in this section.

Our measurement architecture is based on measuring end host state *and* network state simultaneously. We do this by taking concurrent measurements at both the client and the server ends of sample connections using both active (by sending probes into the network) and passive (by logging packets that arrive at hosts) techniques. We do not have the ability to take measurements within the network at this time, but we discuss this possibility in Chapter 8. One of the goals of our infrastructure is to enable evaluation of Web performance over a wide variety of conditions. One means for doing is is by varying server load for different tests so as to allow controlled exploration of a range of server/network interactions. By varying server load using SURGE and running experiments at different times of day, we measure Web performance when either or both are heavily or lightly loaded (in qualitative terms).

The test apparatus of WAWM v1.0 consisted of a Web server running at at Boston University, along with two SURGE client systems connected to the server on a LAN. The SURGE clients enabled the generation of varying load on the server. The infrastructure also consisted of a single off-site client, located across the Internet (15 hops) from the server which generates test connections to the server. Passive packet trace measurements were taken both at the off-site client and at the server. Active measurements of the connection path are also collected during each transfer using the `Zing` tool [3] which sends UDP packets at Poisson intervals along the same path as the connection and measures the time-averaged network state (packet delay and loss rate). Although this apparatus is a reduced version of the larger system we deployed in WAWM v2.0 (as described in Section 5.2), it is sufficient to yield surprising and informative results.

The system deployed for v2.0 is an expansion of the v1.0 system based on lessons learned during our initial, wide-area study. The first element of expansion was to make the local load generation more representative by adding the NISTnet wide area emulator to the server cluster system. The second element of expansion was to make to increase the timing synchronization between the server and client systems by adding GPS timers to all hosts. The third element of expansion was to add more client host systems to enable measurement along a wider variety of network paths.

The results we present in Section 6.2 are from WAWM v1.0. While they do not represent a thorough assessment of wide-area Web performance, they yield some immediate insights about the benefits of integrated server/network measurement, and about the relationship between active and passive network measurement. For example, we find that server load has distinctive effects on the pattern of packet flow through the network. We show that for our experimental setup, one of the most noticeable effects of high server load is to generate a significant gap between the transmission of connection setup packets and the first data packet flowing from the server. While this gap can be understood as a consequence of socket and file operations, it is notable that for typical (*i.e.,* short) transfers (20 KB) the gap generally *dominates* transfer time.

In addition, we show an even more surprising effect of server load. When the network is heavily loaded (*i.e.,* packet loss rates are high), it is not uncommon for a heavily loaded server to show *better* mean response time than a lightly loaded server. Our measurements suggest that this may be because heavily loaded servers often show lower packet loss rates, and since packet losses have dramatic effects on transfer latency, this can reduce mean response time.

5.2 WAWM Design and Methods

In this section we describe the systems, tools and methods used to gather data in WAWM v1.0 and v2.0. We break them down into four categories: measurement architecture and tools, management tools, testing protocol, and analysis methods. In each category we first describe the general project goals, then we describe the specific implementations used v1.0 and v2.0.

5.2.1 Measurement Architecture

5.2.1.1 General Architecture

The general architecture of the WAWM hardware components is shown in Figure 5.1. On the left are components that are located at Boston University. We refer to this collection of systems as the *server cluster*. The first system is configured as the Web server and is only used to serve documents to the clients. Servers are also configured with performance monitoring tools so that detailed measurements of CPU, memory and network characteristics can be made during tests. The second and third systems are configured as local load generators. They run the SURGE workload generator which is used to adjust the amount of load on the server throughout the duration of a test. This is important since we cannot expect remote systems to generate enough load to saturate the server when we want to run tests under heavy server load. The fourth system is used to take both passive measurements (*e.g.,* packet traces) and active measurements (*e.g.,* using network probe packets). Using these tools we expect to measure packet delays, packet losses, path routes, and

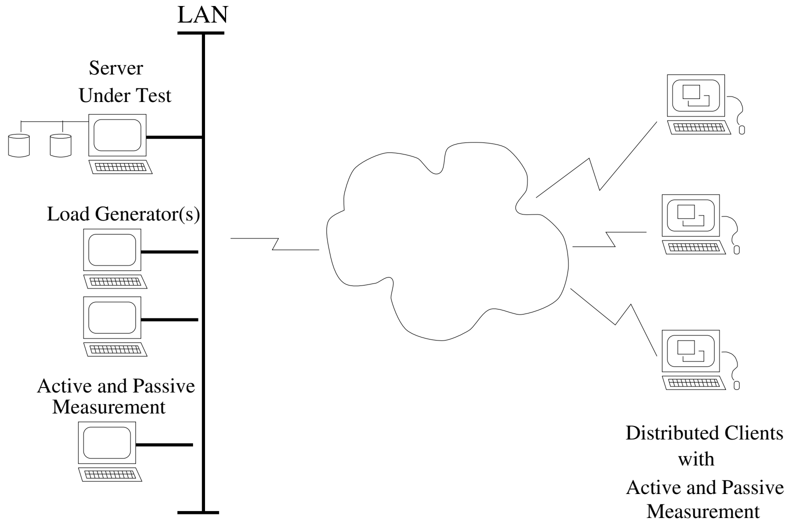


Figure 5.1: General Architecture of WAWM Hardware Components

path characteristics such as bottleneck link speed. Another system is used as the control system which initiates tests and gathers data from the infrastructure. All of the systems in the server cluster are connected to the local Internet service provider (ISP) such that the server cluster network traffic is not a bottleneck.

On the right side of the figure are systems that are located remotely from our site. Our goal is to distribute these systems widely across the Internet in an attempt to explore a wide range of path types. The remote systems make requests for files from the server during tests. They are also configured with both passive and active network measurement tools. Requests are generated by clients using a modified version of SURGE. This modified version can be configured to run in its standard mode or to make repeated requests for a single file in much the same way as WebStone [123].

We are influenced by Paxson's *Network Probe Daemon* study [104] in terms of the number of remote sites we would like to have in the complete WAWM (that study utilized 37 sites which enabled measurement of over 1,000 distinct Internet paths). Selection of the client sites is based on geographic location and Internet connection type with heterogeneity as the goal.

5.2.1.2 WAWM v1.0 Infrastructure

We implemented the 1.0 version of this architecture in order to expose measurement and design issues and act as a platform for developing test infrastructure expertise. The results presented in Section 6.2 are all taken from this prototype system. In our prototype, a single server cluster was located at Boston University (BU) and consisted of six PCs connected via switched 100Mbps Ethernet which is connected to the BU ISP via a 10Mbps bridge. The PCs were configured with 200Mhz Pentium Pro CPUs and 128MB of RAM each. Each system ran Linux v2.0.30 and the server could dual boot to FreeBSD v3.3. The server was also configured with Apache v1.3.0 [111]. `Traceroute` v1.4a5 and `tcpdump` v3.4 were used to measure route characteristics and to gather packet traces respectively. `xntpd` v3-5.93 was used to synchronize time stamps between the server cluster and the remote system. One of the PCs in the server cluster was set up as the time server for all tests. The `xntpd` update interval was set to 64 seconds. `Zing` was configured to run both to and from the server cluster with the mean measurement interval set to one second and with a packet size of 256 bytes. The system set up as the Web server was also configured with Webmonitor [7] to measure detailed server performance characteristics in the CPU, memory and network components of the server.

The local load generating systems in the server cluster were configured with SURGE which was set up to make requests for a file set which consisted of 2000 distinct files. The total size of this data set was about 50MB; thus the entire data set could be cached in the server's RAM. Experiments were run under either *low* or *high* local load levels. Characterizations of load under a range of SURGE UEs for the same PC systems are presented in Chapter 4. Using HTTP/1.0, we used 40 UEs for *low* load and 520 UEs for *high* load.

The v1.0 installation used only a single remote site which was located at the University of Denver (DU). This system was configured with a 333Mhz Pentium Pro CPU and 64MB of RAM and ran Linux v2.0.35. It was connected in the local area via switched 100Mbps Ethernet and then to the DU ISP via a 10Mbps bridge. The system was set up with the same network measurement and time synchronization tools as the systems in the server cluster. The modified version of SURGE was set up to make repeated requests for one of three files on the server. The three file sizes selected were 1KB, 20KB and 500KB. These file sizes were chosen to demonstrate a range of behavior in the network: 1KB files measure essentially only the connection setup time; 20KB files are typical-sized Web transfers and typically involve about 6 or 7 round trips, so are dominated by TCP's slow-start behavior; 500KB files allow us to observe the mode in which most packets are sent in the Web, namely in TCP's congestion avoidance mode.

5.2.1.3 WAWM v2.0 Infrastructure

Based on lessons learned in our v1.0 implementation of WAWM, we enhanced the v2.0 Infrastructure in a number of ways. The first enhancement was an upgrade of the operating systems used throughout the infrastructure. In v1.0 we used a version of Linux with a documented TCP bug. This bug caused random delay of the transmission of ACK packets from clients in response to the receipt of packets retransmitted after a timeout drop. This delays is evident in the results presented in Section 7.3. The systems were upgraded to Linux v2.2.14. Important issues with the v2.2.14 of Linux are that its TCP stack implements the NewReno [41] version of TCP and is SACK [75]. Both of these issues have effect the operation of our analysis tool `tcpeval` which is described in Chapter 7.

Our second enhancement was to use an improved kernel monitor. System monitoring in v1.0 was done using ad-hoc tools and Webmonitor which is very limited in what elements of system performance it can trace. We installed the `Linuxtrace Toolkit` (LTT) on our Web server [122]. LTT is a robust tracing and analysis package which enables lightweight but detailed tracing of kernel performance as well as instruction execution which was not possible with the tools used in WAWM v1.0.

The next enhancement was a change in client request generation. In v1.0 our SURGE client simply made requests back to back as quickly as possible. In v2.0, in an effort to reduce load on the client systems' networks, we ran SURGE clients which injected random OFF times between file transfers. OFF times were distributed via a Pareto distribution using the same parameters given in Section 4.3. While this reduced the total number of files transferred during a test, we were still able to gather ample data during each test.

We used 6 clients in v2.0 of WAWM. These client systems were hosted in the following locations:

1. Harvard University, Cambridge, MA.
2. Purdue University, West Lafayette, IN.
3. University of Denver, Denver, CO.
4. University of Saskatchewan, Saskatoon, Canada.
5. The ACIRI Research Lab, Berkeley, CA.
6. Hewlett-Packard Labs, Palo Alto, CA.

This expanded number of clients provided us with much more varied paths between our clients and the server cluster at BU. The details of the path characteristics are given in Section 6.2.1. All of the client systems were identical: 333Mhz Pentium II's with 64MB RAM, 10/100Mbps Ethernet cards and dual bootable to Linux v2.2.14 or FreeBSD v3.3.

The final two enhancements were the addition of the NISTnet wide area emulator in our server cluster and GPS timers in all of our systems. These enhancements are presented in detail in Sections 5.2.2 and 5.2.3.

5.2.2 Wide Area Emulation in Server Cluster

A possible shortcoming in terms of the realistic nature of the WAWM v1.0 environment was that the local load generators were connected to the server via a switched, local area, 100Mbps Ethernet connection. We investigate this issue in this section and present our approach to local load generation in WAWM v2.0.

5.2.2.1 Issues in Local Load Generation

Local load generation in WAWM v1.0 is representative of of actual users transferring files from the perspective of the seven distributional models that are part of SURGE. However, there are some possible short comings given the fact that we are trying to place as realistic a load as possible on the server and the fact that in our heavy load conditions, most of the requests come from SURGE clients connected on a LAN. The effects of connecting the SURGE local load generators via switched, high speed LAN in WAWM v1.0 are:

1. Local load transfers are subject only to signaling delays, *i.e.* there is no appreciable propagation delay, no router queuing delay and no delay due to route variation.
2. Local load transfers experience virtually no packet loss and certainly none due to intervening background traffic.

The impact of both of these effects is that connections for individual transfers will close more quickly than if the transfers were made from clients distributed in the Internet. This is self evident for the first effect. Packets losses lengthen the overall transfer either due to the server not being able to send packets during a portion of the transfer (which can happen when a loss is detected via time out) or to the deflation of the congestion window after a loss, which means that fewer packets are transferred for each round trip time.

Our results in Chapter 4 show that holding connections open for a greater period of time places a greater load on the server for a fixed set of clients. It could also be the case that if client connections were made from a variety of locations in the wide area, that load on the server (for the same number of local clients) would also be more *variable* due to the fact that some clients would be closer (in terms of number of hops) to the server than others. Our goal in WAWM v2.0 is to enhance our test environment so as to make it more representative, thus we incorporate wide area delay *emulation* for our local load generators.

5.2.2.2 Wide Area Emulation via NISTnet

Wide area emulation is the ability to artificially create the wide area conditions of delay and loss in local area packet flows. At the present time, there are only two choices for off-the-shelf WAN emulators; `dummynet` (which comes standard with a number of BSD distributions) and NISTnet [128] from the National Institute of Standards and Technology (NIST). `dummynet` is a shim layer between IP and the link layer in BSD which enables random delays and loss to be introduced at hosts generating packets. While it is certainly a useful tool in laboratory tests, `dummynet` can only be used on systems running BSD [1] which limits is general use. Since it runs on the host that are generating data (either clients or the server), under heavy load, it has the potential to perturb the timing values in tests.

We selected NISTnet as our WAN emulator for WAWM v2.0. NISTnet is a tool which replaces the normal IP forwarding mechanism in a system which is configured as a router. It enables random delays, loss, and bandwidth restrictions to be introduced to packet flows going through that router. NISTnet was developed in 1997 by Mark Carson and has recently had a number of new features included which make it useful for our purposes. The v1.3bis release which we used in WAWM v2.0 includes the ability to configure multiple IP addresses, ports, and protocols (TCP or UDP) on a single system and the implementation of a high efficiency queue management mechanism which enable NISTnet to continue to operate effectively even under heavy load.

NISTnet under Linux consists of a kernel patch and loadable modules which allow the emulator to be configured and managed. It is typically installed on a system set up as a router between a client and server: we describe the details of our installation in Section 5.2.2.3. Once installed, the user is required to specify the source and destination of packets to which is will apply WAN emulation. NISTnet only applies WAN emulation *one way*, so both directions of a flow must be specified. This enables asymmetric characteristics to be applied to a flow. For each direction of a flow, the user must specify:

- Mean value of one-way delay,

[1] Support for `dummynet` in Linux is currently under development

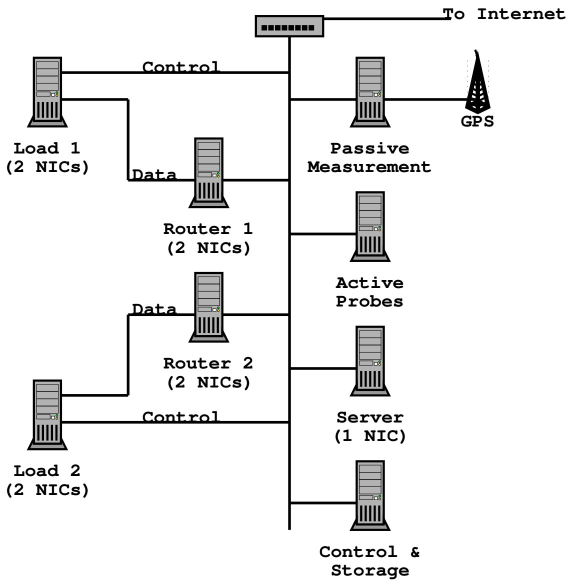


Figure 5.2: WAWM v2.0 Server Cluster Architecture.

- Standard deviation of one-way delay,
- Maximum allowed bandwidth in bytes per second,
- Percentage packet loss,
- Percentage duplicate packet,
- "Derivative" random drop parameters (similar to Random Early Detection).

NISTnet provides a GUI interface for parameterization. This interface also provides real time values of traffic flowing through it which is useful for insuring that it is running correctly. Unfortunately, we found that if the GUI is run during tests with heavy load, it tends to make the system unstable. We used the command line interface for our tests.

5.2.2.3 NISTnet in WAWM v2.0

Our use of NISTnet in WAWM v2.0 consisted of setting up a separate router system running NISTnet for each of our local load generators as shown in Figure 5.2. Each of the router systems was a 333 MHz Pentium II with 128MB RAM (the load generators were 200 MHz Pentium II's). All systems were running Linux v2.2.14. One of our primary concerns in configuring the systems was to insure that routers would not become overloaded during tests and thereby change the desired delay and loss characteristics. We monitored the CPU and memory on the routers during tests and found that under the heaviest loads (360 UE's per SURGE client system), CPU utilization never exceeded 5% and memory utilization never exceeded 70%.

Parameter	Value
Delay Mean	30ms
Delay Standard Deviation	3ms
Bandwidth	Maximum
Percent Loss	1
Percent Duplicates	0

Table 5.1: NISTnet v1.3bis Configuration Parameters used in WAWM v2.0. Incoming and outgoing flows were configured identically.

The routers were configured with two, 100Mbps Ethernet network interface cards (NICs). Routing functions were enabled via the standard IP forwarding and routing functionality in Linux. One NIC exposed a standard IP address to the rest of the server cluster LAN while the other was configured as a private interface with IP masquerading. All packets received on the private interface were forwarded to the public interface and then to the LAN with a masqueraded IP of the forwarding router.

The load generators were connected to the routers via a private IP (192.168.xxx.xxx). By binding to this IP during socket creation in SURGE clients, all local load traffic passed through the routers. While this architecture worked fine for load generation, it did pose a problem for control. With a single, private interface, the only way to remotely control the load generators was to first log into the router and then into the load generators. Since all of our tests are automated via remote, secure shell commands, this was not an acceptable solution.

To address the issue of remote management and control of the local load generators, we added a second NIC to each system. These NICs exposed standard IP addresses to the rest of the server cluster and gave us a direct channel to the local load generators for control and management. We still had to be very careful in constructing the routing tables in the local load generators in order to route control traffic through the standard IP and all data traffic through the private interface.

While the enhancements in NISTnet v1.3bis were great improvements over prior releases of the tool, there was still one major shortcoming. The random number generator used in NIST is extremely simple and is noted in the source that it does not produce a very good random number sequence. We address this by adding the `ran1` random number generator from [110]. While this is certainly not the best random number generator available, it is a great improvement over what comes in the standard NISTnet distribution.

Once operational, we still needed to set the configuration parameters on the NISTnet routers to enable more realistic local load generation. We investigate the effects of a number of different NISTnet configurations in Section 5.2.2.4 and we selected the parameters listed in Table 5.1 for tests run in WAWM v2.0. Note that incoming and outgoing flows were configured identically, thus a mean delay of 30ms in each direction gives a round trip mean delay of 60ms.

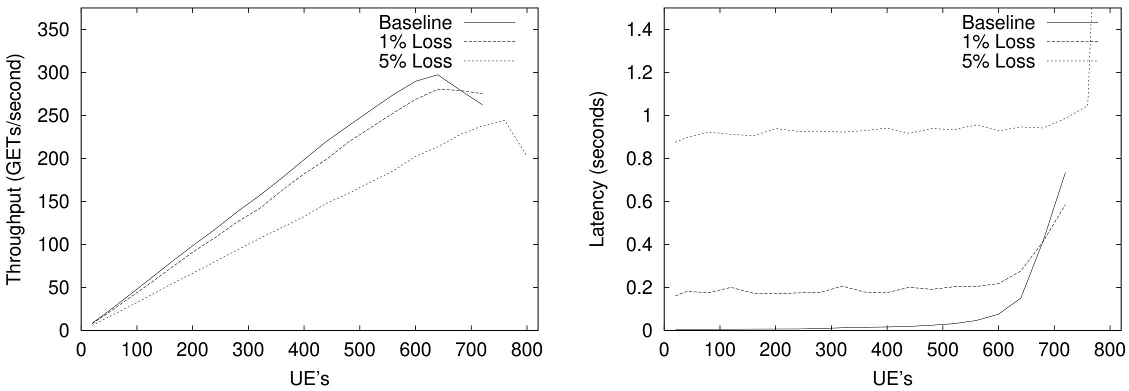


Figure 5.3: Effects of Packet Loss on Performance in WAN Emulation

5.2.2.4 Performance Impact of NISTnet in WAWM v2.0

Before we could begin tests in WAWM v2.0 we needed to know how wide area emulation would affect local load generation. In Chapter 4, we present performance results of tests using SURGE over a variety of loads and server configurations. In these tests, we have a single server configuration that is used in all tests: a 200MHz CPU, 128MB RAM, and a single 100Mbps Ethernet NIC. The server ran Linux v2.2.5 and Apache v1.3 and we use HTTP/1.0 for all our requests.

Our first test was to simply run a baseline of SURGE clients through the routers with NISTnet turned off. The baseline throughput and latency values are slightly higher than those shown in Chapter 4 due to enhancements in Linux and Apache, but the overall maximum performance is comparable, which shows that the routers without NISTnet running do not introduce observable overhead to our transactions. The throughput and latency results from the baseline experiment are used for comparison in the subsequent tests.

Our first experiment is to explore the effects of emulating loss conditions without delay. NISTnet implements a uniform, uncorrelated loss model. We investigate the effects on performance as compared to the baseline with 1% and 5% loss. The results are shown in Figure 5.3. From the figure we see that 1% loss causes approximately a 5% reduction in maximum throughput and an increase in steady state latency (latency before maximum throughput) from approximately 10ms to 170ms. With 5% loss, there is approximately a 20% decrease in maximum throughput from the baseline and an increase in steady state latency from approximately 10ms to 900ms.

Next we test the effects of emulating propagation delay without loss. NISTnet implements a random delay model which requires users to specify average and standard deviation in each direction of a flow. We used the default distributional model for delay which is Pareto-normal. We investigate the effects of propagation delay in two test: the first is with a 60ms mean delay, 6ms standard deviation and the second is with a 30ms mean delay, 3ms standard deviation. The throughput results are shown in Figure 5.4. From this figure, we see that maximum throughput is reduced in both tests by approximately 15% with the 30ms delay reaching its maximum before the 60ms delay. Steady state latency increases from approximately 10ms to 300ms for the 30ms delay, and from approximately 10ms to 600ms for the 60ms delay.

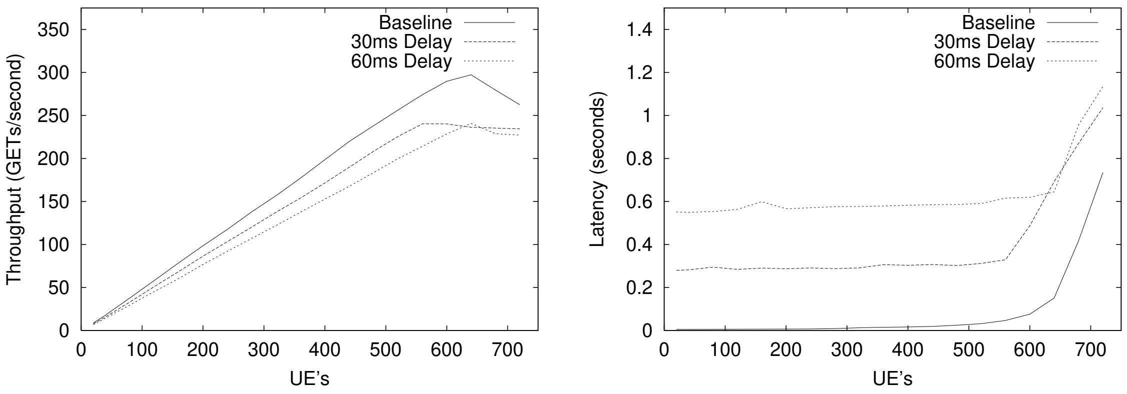


Figure 5.4: Effects of Propagation Delay on Performance in WAN Emulation

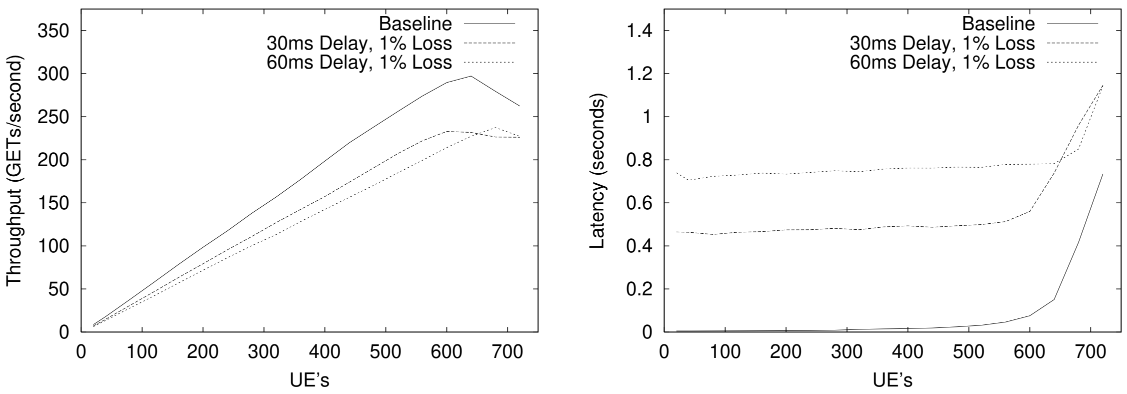


Figure 5.5: Combined Effects of Propagation Delay and 1% Loss on Performance in WAN Emulation

Up to this point, we have considered the effects of propagation delay or loss in isolation, however, real world conditions are a combination of the two. We show the effects of combined latency and loss emulation in Figures 5.5 and 5.6. Figure 5.5 shows the effects on performance of adding 30ms average delay with 1% loss and 60ms delay with 1% loss. The figure shows that maximum throughput is reduced by 22% and steady state latency increases from approximately 10ms to 500ms for the 30ms/1% loss combination. The figure also shows that maximum throughput decreases by 22% and steady state latency increases from approximately 10ms to 750ms. These results indicate that the combination of delay and loss emulation has a somewhat additive effect, namely that the effects of delay and loss in isolation can be roughly added when combined delay and loss emulation is used.

Figure 5.6 demonstrates the combined effects of 30ms and 60ms propagation delay with 5% loss. The figure shows that the effects of 30ms delay and 5% loss reduces maximum throughput by 25% and increase steady state latency from approximately 10ms to 1,200ms. The combination of 60ms propagation delay and 5% loss decreases maximum throughput by 25% and increases steady

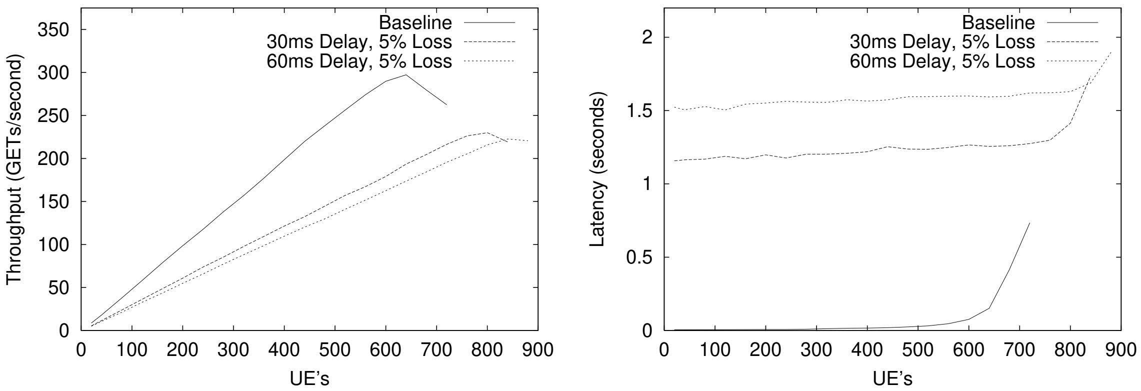


Figure 5.6: Combined Effects of Propagation Delay and 5% Loss on Performance in WAN Emulation

Test	Load (UE's)	CPU Utilization	
		Mean	Std. Dev.
Baseline	40	1.3	1.1
30ms, 1% loss	40	3.9	4.9
Baseline	560	87.1	8.6
30ms, 1% loss	560	80.5	9.8

Table 5.2: Web Server CPU utilization comparing baseline and WAN emulation for low and high loads.

state latency from approximately 10ms to 1,600ms. These results support the additive effects on performance of the combination of delay and loss emulation.

Our tests show that WAN emulation clearly has an impact on performance from the perspective of maximum throughput and steady state latency. However, the question that is more important from the context of WAWM experiments is whether or not WAN emulation in WAWMv2 will place a *different kind* of load on the server which will also be serving files to our remote clients. If not, then there may be no need for WAN emulation in conjunction with the local load generators. We evaluate this question from the perspective of the CPU load placed on the server during low and high server loads used during WAWM tests. We compare the 30ms/1% loss conditions for WAN emulation against the baseline. Table 5.2 shows that the mean and variance of the CPU load placed on the server under low and high SURGE client load is different if WAN emulation is used. CPU utilization is higher and more variable using WAN emulation under low client load. CPU utilization is lower but more variable under high client load conditions. These differences indicate that server load characteristics are indeed different using WAN emulation. Thus, we adopt its use in WAWMv2.

5.2.3 Time Synchronization

In this section we discuss the details of our clock synchronization methodology in WAWM v2.0. This entails the use of GPS timers in our client systems and in our server cluster.

5.2.3.1 Timing Issues

We show in Chapter 7 that minimum delays for individual packet transfers during a file transfers in the Internet take place on the order of as little as a few milliseconds. The synchronization method used in WAWM v1.0 enabled precision of measurements between client and server on the same order of magnitude. Thus, our measurements over very short paths could be quite unreliable. One of our primary goals in WAWM v2.0 was to increase the *accuracy* (defined as the agreement on time value between systems) of clocks in the infrastructure.

Clocks on computers are typically implemented as quartz oscillators and hardware counters which interrupt the CPU over a range from 100Hz to 1024Hz. Clock counts are often stored in a 64 bit register which gives a computer the potential for nanosecond resolution: UNIX clocks usually run with nanosecond resolution (although they require a patch to the kernel to realize this accuracy in a measurement timestamp). However, without correction, clocks can drift as much as seconds per day. Drift is based on clock quality and environmental conditions (moisture and temperature) which can increase or decrease drift.

A *reference clock* is a system which produces highly accurate time values. Reference clocks are typically either a cesium (atomic) clock or a system synchronized by GPS signals (GPS satellites all have cesium clocks). The Network Time Protocol (NTP) was designed to enable clock synchronization between systems connected over a network assuming that one of the systems is the reference system. NTP enables the client to synchronize with the server through a constant interval packet exchange adjusting time based on offset, jitter, frequency error and stability. Errors in synchronization using NTP are due to variations in network delay and variable hardware and software latencies in the host (this combination is referred to as jitter). The protocol uses a feedback loop, measured clock offset and estimated one-way transmission delay between the client and server to adjust the phase and frequency of the client's clock. Typically, NTP accuracy is 5ms to 100ms in the wide area and as good as 1ms on a LAN [113]. With Pulse Per Second (PPS) support from GPS signals, clocks can be synchronized on the order of nanoseconds.

In order to utilize the capabilities provided by NTP, a system's kernel must implement a clock model which can accept the adjustments provided by the protocol. Mills *et al.* describe such a clock model in RFC 1589 [81]. The model is a hybrid using both phase-lock loops and frequency-lock loops to carefully discipline the kernel clock using both feedback and current data. Recent implementations of this model in Linux and FreeBSD enable a nanosecond resolution system clock. The clock model also describes how to process PPS signals. Issues in PPS processing include how to capture the signal with high accuracy and how to apply it to the clock model.

PPS signals are emitted by all GPS satellites. If PPS is used to discipline the kernel, then PPS signal processing is also done in the kernel (as opposed to the application level which is where standard NTP runs). PPS drivers are installed in the kernel which direct the kernel in how to use the PPS signal for both time and frequency adjustments. PPS signals are typically received on the serial port of a system and reference drivers in NTP are specific to the GPS receiver that is being used.

5.2.3.2 GPS Hardware Configuration in WAWM v2.0

Configuring and installing GPS timers in the WAWM v2.0 systems was one of the most time consuming and logistically difficult parts of this project. While there is a wide range of GPS hardware products, there is little published on the use of GPS in Internet measurement projects and configuration is tricky. We learned many lessons in the process of setting our systems up and will be able to apply these lessons in future installations of WAWM systems.

After evaluating a number of candidates, we settled on the Motorola Oncore UT GPS receiver. The principal benefits of this unit are that it has a sophisticated NTP driver which has built in kernel clock disciplining capability (most other units require explicit command sequences in the NTP configuration file to enable the proper clock discipline). The second benefit of the Oncore is that it is relatively inexpensive ($350).

The GPS receiver is not useful by itself since it is provided by the manufacturer as a daughter card which must be mounted on a PCI card in order to actually use it in a computer. We selected the SATPAK-PCI card from Zeli Systems, Inc. This unit is easily installed and makes PPS signals available either on the PCI bus or on a serial port. The difference between the two is that the serial cable transmits the PPS signal to the CPU more slowly than the PCI interface which has the effect of adding a small (but predictable) offset. Unfortunately, at the time of writing there was no PCI driver available under Linux or FreeBSD for this unit, thus we used the standard serial port interface from the card to the computer [2].

After installing the GPS card in a system, the GPS must be connected to an antenna which receives the GPS signal from the satellites. In order to initially synchronize the systems clock, the antenna must have visibility of at least 3 GPS satellites. This typically requires that the antenna be set up on the roof of the building housing the reference system: this requirement limits the deployment of our client systems since roof access is not always possible. Once installed, the antenna (which is about the size of a softball) can effective transmit the GPS signal well over 100' without too much attenuation.

5.2.3.3 GPS Software Configuration in WAWM v2.0

Upon completion of the hardware installation, PPS support and NTP must be installed in the system. The PPS NANO kernel patch enables PPS clock disciplining and also gives the systems clock nanosecond precision. A clean kernel can be patched with the PPS kit and then built (enabling NTP) and installed on the WAWM systems. PPS features are then accessed via the PPS API which is exposed after the NANO kernel is installed.

The next step is to install NTP. The software must be configured for compilation using the `--enable-ONCORE` qualifier. After building and installing the NTP software, the NTP configuration file must specify that NTP should synchronize from the PPS signal. Finally, symbolic links to the serial port on which the PPS is received must be created.

After completion of the software set up, the GPS system must be synchronized by executing a site survey. This consists of letting the system run in survey mode until a positional fix is generated

[2] We plan to develop our own PCI driver for this unit in the future

```
51791 75053.799 0.000000609 16.613007 0.000013035 0.009817 6
51791 75119.800 -0.000000286 16.614990 0.000011292 0.008559 6
51791 75185.801 0.000000192 16.614990 0.000009781 0.007413 6
51791 75249.802 -0.000000460 16.613998 0.000008473 0.006439 6
51791 75313.804 0.000000138 16.613998 0.000007342 0.005576 6
51791 75379.805 0.000000020 16.608047 0.000006371 0.005672 6
51791 75443.806 -0.000000158 16.608047 0.000005532 0.004912 6
```

Figure 5.7: An example of a loopstats file from a WAWM v2.0 client

in one of the NTP log files (this can take as long as 24 hours to complete depending on satellite visibility). Once the site survey is complete, the positional data is hard coded into one of the NTP configuration files and at that point the system is complete.

5.2.3.4 GPS Timing Results in WAWM v2.0

Upon completion of site surveys on the client systems deployed in WAWM v2.0, we were able to realize very high accuracy in system clocks. There are a number of statistical file generated by NTP which list the various ongoing values used to adjust the clock. Of particular interest is the `loopstats` file which generates a column of values for drift as measured by NTP. The loopstats file example in Figure 5.7 has drift values in the third column. This column shows that the largest variation between time values is about 600 nanoseconds.

5.2.4 Management Architecture

In order to run tests, download data and manage systems which are part of the WAWM infrastructure, a secure management platform is required on all WAWM systems. We are evaluating third-party management platforms including the NIMI platform and Lucent's VitalAgent [56]. Both provide the essential management features and flexibility necessary for WAWM as well as built-in network performance measurement tools.

The management platform is an area of the system in which practical matters can dictate design choices. Therefore, in order to gain insight into the relevant issues, we developed our own management tool for the prototype study. The tool is written in Perl and is based on using Secure Shell [115] (SSH v1.2.26). This tool proved to be sufficient for both the prototype tests and the tests run in WAWM v2.0.

The data collected is cataloged and maintained in such a way as to facilitate analysis and reuse by other researchers. The WAWM data repository is maintained at the same central site from which WAWM tests are run. At this point in time a simple file structure is adequate to store the data. Each test run in our prototype study with only a single client resulted in about 9MB of compressed results data.

5.2.5 Testing Protocol

5.2.5.1 General Architecture

A WAWM *test* is a period of time over which a specific set of client systems make requests to a specific server. Specification of tests requires determining the following:

1. The server cluster which will be used in the test.
2. The local load which will be placed on the server.
3. The set of remote clients which will make requests during a test. The influence of distance from the server, Internet connection type and the number of clients participating in a test will all influence the outcome of measurements.
4. The time of day the test will be run. Due to the diurnal cycle of Internet traffic in North America, tests run during the day typically experience more pronounced influence from cross traffic than those run late at night.
5. The duration of the test. The test must be run long enough to collect a representative sample of data for the given network and server conditions.
6. The schedule for active network measurements during tests. Since active measurements could perturb HTTP performance, this schedule must be selected such that the resulting data gives a clear picture of the network's state while not influencing the HTTP tests.

5.2.5.2 Test Details

The testing protocol called for one set of tests to be run during the day (between 11:00am and 7:00pm) and one set of tests to be run at night (between 9:30pm and 5:00am). Initiating tests at these times provided a variety of network conditions for our data. Each test began by running `traceroute` to and from the server which gave us an indication of the current end-to-end routes during the tests. Each test was run for 15 minutes; we observed that this duration was always sufficient to transfer at least 40 of the 500KB files under heavy network load conditions and 4696 of the 1KB files under light network load conditions. `Zing` was run both to and from the server simultaneously with the HTTP requests for the duration of each test. Six separate tests were run during each test set. They consisted of making repeated requests for a single file (1KB, 20KB or 500KB) under both low and high server loads.

The protocol used in WAWM v2.0 was slightly different than in v1.0. First, NISTnet was configured for the local load generators using one way delay mean of 30ms, standard deviation of 3ms, and loss of 1%. Second, since OFF times were injected between files transfers to clients, tests were run for 20 minutes in order to collect additional data.

5.2.6 Analysis Protocol

5.2.6.1 General Architecture

Once test data is collected, it is reduced and analyzed. Automation of the analysis process is necessary due to the large amount of data that is gathered during each test. We have begun the development of a tool called `tcpeval` which will perform these tasks on the packet trace data. Our model is based on `tcpanaly` [103]. In addition to statistical approaches, use of visual analyses including both timeline and sequence plots enables detailed analysis of individual file transfers.

We also use `natalie` [3] to analyze `Zing` traces and intend to develop methods to correlate the data gathered through active and passive measurements.

5.2.6.2 Analysis Details

At the highest level we currently tabulate summary statistics and distributional statistics of file transfer delays, packet delays, packet loss and server performance. These results are presented in Chapter 6. `tcpeval` also generates a number of graphical aids to interpretation, including timeline diagrams which illustrate individual packet exchanges between a client and the server within a single HTTP transaction. In this thesis we focus on timeline diagrams (*e.g.,* Figures 6.2, 6.4, and 6.6) because they effectively represent the timing of events at both ends of a connection. Finally, we use `tcpeval` to do critical path analysis on the data that we gather. The details of CPA and results of the analysis are given in Chapter 7.

Part II

Internet Performance: Results and Analysis

Chapter 6

Summary Performance Measurements and Analysis

6.1 Overview

In this chapter we present the performance results from a set of measurements taken in WAWM v1.0. The focus is on file transfer latencies under different server and network load conditions. We also present the current status of the WAWM v2.0 infrastructure.

6.2 Performance Results from WAWM v1.0

In this section we report the path, file and packet level results obtained using the WAWM v1.0 measurement infrastructure described in Section 5.2.

6.2.1 Measurements of Path Characteristics in WAWM v1.0

The testing protocol called for `traceroute` to be run before and after each test. These measurements showed that the route to and from the remote site was asymmetric but fairly stable for all of the tests (13 hops from BU to DU and 15 hops from DU to BU). While there were occasional route changes between sets of test, we measured only one route change *during* one of the tests. The Denver site includes two commercial backbone providers (AlterNet and Qwest) in the path to BU.

6.2.2 Measurements of file transfer latency in WAWM v1.0

We present performance measurements for file transfers over combinations of server load, network load and file sizes used in the tests. While measurements in WAWM v1.0 were taken over a two week period, we present results of data sets which are typical of all of the data that was collected.

6.2.2.1 Performance under light network load

The details of the file transfer latency under light network load for all three files can be seen in Table 6.1. The table shows that the busy server adds approximately one second to the average file

File Sizes	Server Load	Files Trnsfd	Mean Latency	Std. Dev. Latency	tcpdump % pkt loss	tcpdump mean delay	Zing % pkt loss	Zing mean delay
1KB	low	4696	0.276	0.058	0.0	0.046	0.0	0.044
20KB	low	1521	0.678	0.273	0.1	0.050	0.1	0.045
500KB	low	365	2.552	0.283	0.0	0.056	0.1	0.047
1KB	high	768	1.256	1.171	0.1	0.045	0.1	0.043
20KB	high	572	1.662	1.239	0.1	0.049	0.1	0.044
500KB	high	223	4.125	1.584	0.0	0.062	0.1	0.045

Table 6.1: File transfer latency statistics under light network load

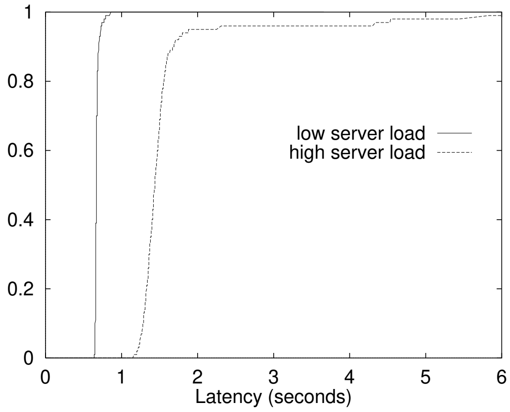


Figure 6.1: CDF for file transfer latency under light network load for the 20KB file

transfer latency for the 1KB and 20KB files and over 1.5 seconds to the average 500KB file transfer latency. Mean packet delay and loss characteristics under light network load as measured by both `Zing` and `tcpdump` can be seen in Table 6.1. Delay and loss measurements are for the outbound path from the server to the client. These network measurements show that the network conditions were fairly stable throughout the measurement period.

Under light network load and light server load, the file transfer latency had very low variability for all three file sizes. Under heavy server load, the variability increases slightly. The principal effect of server load when network load is low was to increase the average file transfer time by a relatively constant amount for each file size. These effects can be see in the cumulative distribution function (CDF) diagram for the 20KB file in Figure 6.1. This diagram also shows that for a small percentage of transfers, latency can increase over 6 times when server load is high.

Another way to analyze file transfers and the effects of varying server load for the lightly loaded network measurements is through the use of TCP time line plots. The time lines for typical transfers of the 20KB file can be seen in Figure 6.2. In these plots (as well as the rest of the time line plots in this section), the client is on the left hand side and the server is on the right. The time line plots highlight the delay incurred for all file transfers when the server is busy. After the TCP

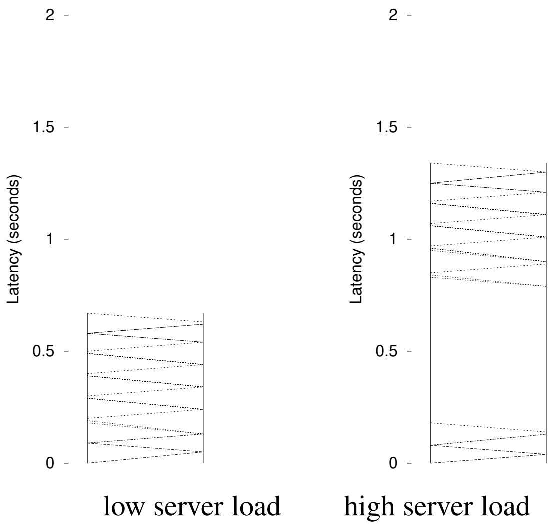


Figure 6.2: Time line diagram of typical 20KB file transfers under light network load

connection set up sequence, there is roughly a one second delay before the first data packet is sent by the server. This delay is due to the need for the request from the client to get up to user space and then for the response to get back down to the network. As can be seen in the lightly loaded case, this is nearly instantaneous when the server is lightly loaded. This data combined with the information from the CDF of file transfer latency leads us to conclude that if the network is lightly loaded, the maximum additional delay that most transfers might experience when accessing our busy server is about one second.

6.2.2.2 Performance under heavy network load

When packet delays and losses increase, transfer characteristics change. The details of the file transfer latency under heavy network load for all three files can be seen in Table 6.2. The network measurements from `Zing` indicate that network conditions were fairly stable throughout the measurement period. However, these measurements do not correlate closely with the mean delay and loss values extracted from the `tcpdump` traces. Packet transfer delays extracted from the `tcpdump` traces increase between 13% and 48% from light network load to heavy network load. The increase in packet delays is an indication of the level of congestion in the network. The table shows that under light server loads, the mean transfer latency increases in a range from 2.2 to 5 times the mean latency when the network is lightly loaded. It also shows that under heavy server loads, the mean transfer latency increases in a range from 1.3 to 2 times the mean latency when the network is lightly loaded. Perhaps the most interesting statistic to note is that transfer latency of the 500KB file under heavy network loads actually *decreases* when the server load is high versus when it is low. This effect is explored in more detail in Section 6.2.2.4.

File transfer delays are strongly influenced by both congestion and packet loss. Measurements taken during the day time periods for all data sets showed higher packet loss and delay character-

File Sizes	Server Load	Files Trnsfd	Mean Latency	Std. Dev. Latency	tcpdump % pkt loss	tcpdump mean delay	Zing % pkt loss	Zing mean delay
1KB	low	1543	0.728	1.373	4.2	0.052	5.0	0.051
20KB	low	656	1.503	1.827	3.8	0.056	5.9	0.051
500KB	low	70	12.977	4.501	3.2	0.083	6.4	0.053
1KB	high	610	1.598	1.329	2.7	0.051	6.4	0.052
20KB	high	444	2.127	2.169	1.9	0.056	4.3	0.050
500KB	high	110	8.297	2.993	1.5	0.085	5.8	0.051

Table 6.2: File transfer latency statistics under heavy network load

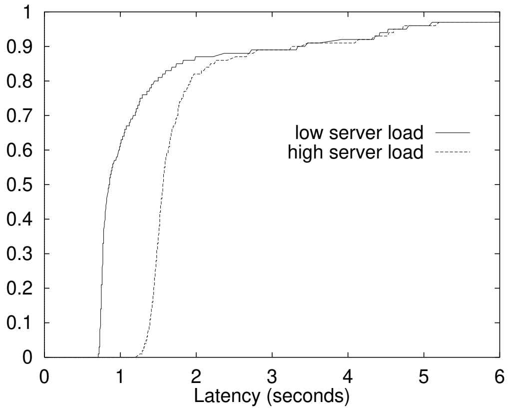


Figure 6.3: CDF for file transfer latency under heavy network load for the 20KB file

istics as would be expected. Under heavy network load, file transfer latency had generally higher delay and higher variability for all three file sizes regardless of server load. These effects can be seen by comparing Figure 6.1 with the corresponding figure for heavy network load (Figure 6.3). While in Figure 6.1, the knee in the CDF occured around the 90^{th} percentile, in Figure 6.3, the knee is much lower indicating that a much larger fraction of transfers experienced severe delays.

The time lines for typical transfers of the 20KB file when the network is heavily loaded can be seen in Figure 6.4. As in the lightly loaded network case, the delay in sending the first data packet from the server is evident. These diagrams show a general elongation in packet transfer times versus the lightly loaded network time lines, which is an indication of the level of congestion in the network during the measurement.

6.2.2.3 Effects of Packet Loss

Measurements made during high network load show the effects of both network congestion and packet loss. In this section we specifically examine the effects of packet loss on file transfer latency. The details of the effects of packet loss under heavy network load and high server load can be seen in Table 6.3. This table shows the differences between files transferred with and without packet

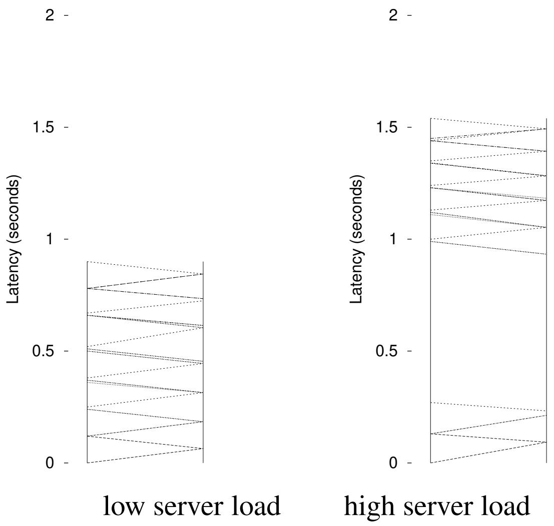


Figure 6.4: Time line diagram of typical 20KB file transfers under heavy network load

loss. File transfers for files which lose at least one packet are between 1.3 and 7.8 times slower than files transferred without packet loss. As might be expected, variability for files transferred without loss is also much lower than that of files transferred with loss.

Figure 6.5 compares the impact of packet loss on file transfer latency for the 1KB file and the 20KB file. As can be seen from the figure, packet loss can significantly increase both mean and variability of file transfer delay. When comparing this figure with Figure 6.3 it can be seen that almost all long transfer delays are due to packet drops. One notable feature of the figure is the cusp at about the 40% level for the 1KB file. This is due to lost SYN packets. There is a three second timeout period for the re-sending of a lost SYN packet in the Linux TCP implementation.

The time lines for transfers of a typical 20KB file with and without packet loss can be seen in

File Sizes	Server Load	Files w/o pkt loss	Mean Latency	Std. Dev. Latency	Files with pkt loss	Mean Latency	Std. Dev. Latency
1KB	low	1305	0.326	0.062	238	2.936	2.541
20KB	low	351	0.784	0.066	305	2.331	2.428
500KB	low	0	n/a	n/a	70	12.976	4.501
1KB	high	536	1.337	0.763	74	3.491	2.771
20KB	high	314	1.869	1.956	130	2.752	2.511
500KB	high	6	3.947	1.053	104	8.548	2.548

Table 6.3: File transfer statistics under heavy network load for files with no dropped packets or at least one dropped packet for 500KB file

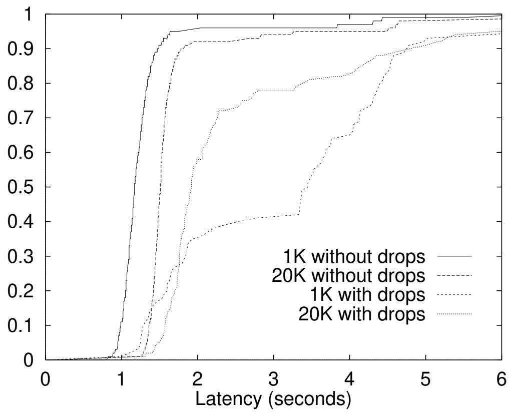


Figure 6.5: CDF for file transfer latency under heavy network load and heavy server load for files with and without packet loss

Figure 6.6. This figure shows the effect of a single lost packet during the data transfer phase of the HTTP transaction under light server load and heavy network load. The packet loss will cause TCP to restart slow start in order to transfer the remaining packets. The overall effect of this loss is the addition of about 0.5 seconds to the file transfer latency.

6.2.2.4 Effects of server load

Initial effects of server load on transfer latencies have been shown in the preceding sections. However, closer examination of the data from all ten data sets shows an interesting phenomenon. For transfers of the 500KB file, in seven out of ten of the data sets, packet loss actually goes *down* when server load is *increased* during the tests. Furthermore, in three out of those seven cases, file latency *decreases* when server load is *increased*. This effect can be seen in Figure 6.7 for one of these cases which is the same as the data in Table 6.2. This result can possibly be attributed to one of two things. First, the experiments for the 500KB file transfer are run approximately 45 minutes apart and the network activity could change enough during this time to cause performance to improve. However, active measurements indicate that the network was fairly consistent across all tests. Another possible explanation is that the heavy load on the server prevents it from producing large bursts of packets as it might normally do if it were lightly loaded. We discuss the possibility of investigating this hypothesis as future work in Section 8.6.

6.3 WAWM v2.0 Status

At the present time, WAWM v2.0 is deployed and operational. There are currently 6 GPS enabled client systems which are taking measurements of Web transaction performance using the protocol described in Section 5.2.5.2.

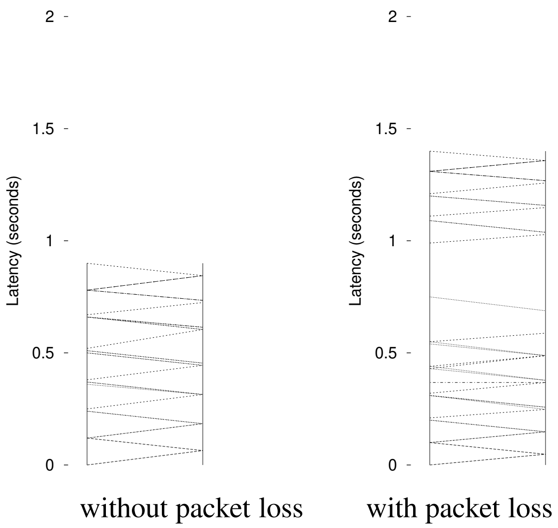


Figure 6.6: Time line diagram of a 20KB file transfers under heavy network load and low server load

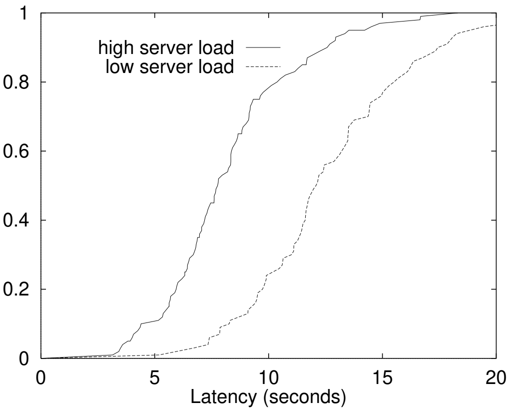


Figure 6.7: File transfer latency for the 500KB file transfers under heavy network load

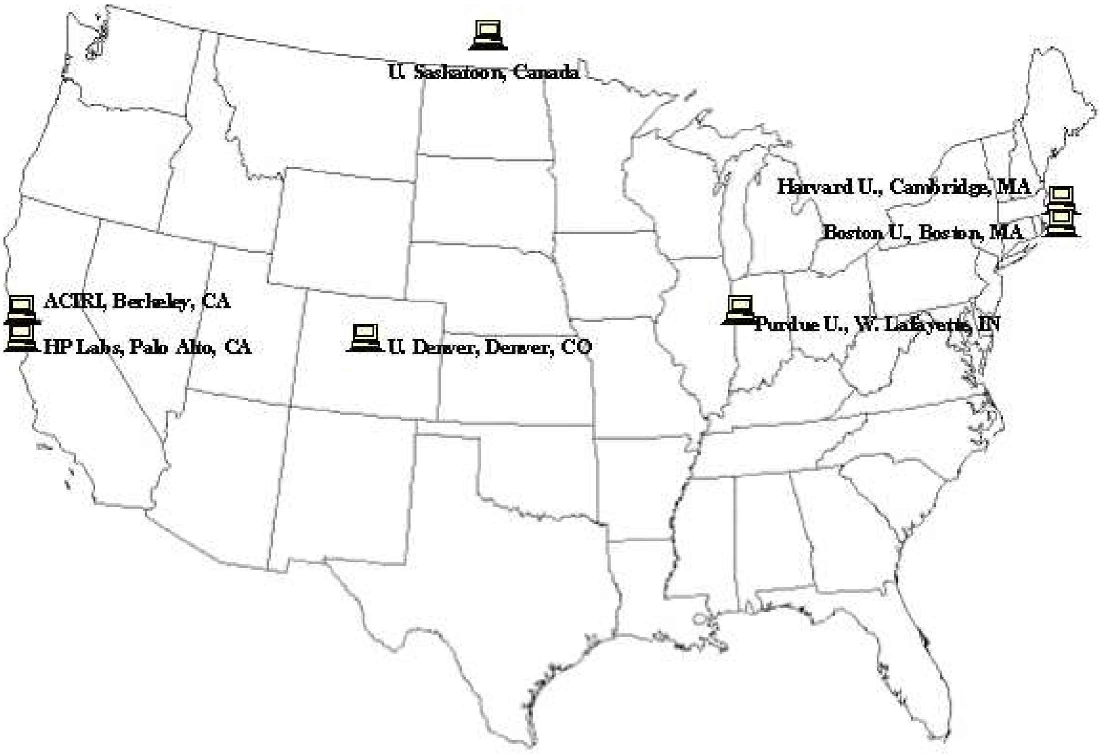


Figure 6.8: WAWMv2.0 System Map

6.3.1 Measurements of Path Characteristics in WAWM v2.0

We maintained our Denver client system in WAWM v2.0. The path characteristics with respect to Internet Service Provider (ISP), hop count and typical RTT for that system remained stable during the time between v1.0 and v2.0 of WAWM. The backbone carrier for this client is Qwest which is a commodity carrier. Our second client was located at Harvard University, 8 hops from the server cluster at Boston University. The path to the Harvard client was unique in that it travels over a metropolitan area network (MAN) and never reaches the Internet per-se. Our third client was located at Purdue University, our fourth client at the ACIRI lab in Berkeley, CA: both of these systems were connected via the Abiline high speed network which connects over 150 universities across the country. This backbone typically has much lower loss than commodity ISP's. Our fifth client at the University of Saskatchewan in Saskatoon: this client is connected via Qwest to CANET which is the Canadian equivalent of Abiline. Our sixth client was located at HP Labs in Palo Alto, CA and it's backbone carriers are both Qwest and BBN. The details of the path characteristics for each client are listed in Table 6.4. One aspect of using GPS timers is that the site survey used to initialize the timers gives the precise position of the system. Those positions are given in Table 6.4 along with their positions on the map in Figure 6.8. The combinations of distances and backbone providers give WAWM v2.0 a much more robust set of connection characteristics.

Client	Lat	Lon	Hight	Backbone	Distance (hops)	Typical RTT (ms)
BU Server	N 42d 20.94m	W 71d 6.26m	12.65m	n/a	n/a	n/a
ACIRI	N 37d 52.21m	W 122d 16.29m	62.46m	Abiline	16	73
Denver	N 39d 40.41m	W 104d 57.68m	1635.26m	Qwest	19	126
Harvard	N 42d 22.72m	W 71d 7.04m	12.24m	MAN	8	5
HP Labs	N 37d 24.72m	W 122d 9.00m	40.87m	Qwest/BBN	19	121
Purdue	N 40d 25.61m	W 86d 55.00m	181.35m	Abiline	9	33
U. Sask	N 52d 7.82m	W 106d 37.70m	496.5m	Qwest/CANET	16	135

Table 6.4: Client System Position and Path Characteristics in WAWM v2.0.

Chapter 7

Critical Path Analysis of TCP Transactions

7.1 Overview

In the last Chapter we saw that even in relatively simple settings (such as the retrieval of a single, static Web page) the complexities are great enough to allow room for inconclusive finger-pointing when delays arise. All too often, network providers are blaming the servers at the same time server managers are blaming the network.

In this Chapter we focus on methods that aid in pinpointing the sources of delays in Internet applications—allowing conclusive, accurate assignment of delays to either the network or the server. Furthermore, our methods are aimed at studying the network and the server together, as a complete end-to-end system. We focus in particular on Web transactions, looking to understand the relative and joint contribution that servers and networks make to overall transfer latency. Our approach is to apply *critical path analysis* to these transactions.

In general, critical path analysis identifies the precise set of events that determine an application's performance. The central observation of critical path analysis as applied to distributed systems is that only *some* of the component activities in a distributed application are responsible for the overall response time; many other activities may occur in parallel, so that their executions overlap each other, and as a result do not affect overall response time. In other words, reducing the execution time of any of the activities on the critical path will certainly reduce overall response time, while this is not necessarily true for activities off the critical path.

Recent work in [63] indicates that HTTP/1.0 is the dominant protocol used in the World Wide Web today. In this Chapter we apply critical path analysis to HTTP/1.0 transactions that use TCP Reno (as specified by RFC 2001 [121]) as their transport mechanism.[1] We show how to do critical path analysis for TCP Reno, and describe how we use the resulting critical path to determine the fractions of response time that are due to server delays, client delays, and network delays (including packet loss delay, propagation delay, and delay due to network variations such as queuing at routers,

[1] While our implementation is specific to TCP Reno, the methods discussed could easily be extended to encompass the packet loss recovery mechanisms of other versions of TCP such as New-Reno or SACK.

route fluctuations, etc.). The analysis has the appealing property that improvements in any of the delays we identify would surely have improved response time. This suggests that looking at the fraction of response time assigned to each category can provide a guide to which part of the system is most profitably improved. Despite the strengths of our approach, there are some inherent limitations which are described in Section 7.2.6.

Properly constructing the critical path depends on the details of the TCP implementation being used. However we have been able to develop a method that we believe applies to any TCP Reno implementation compliant with RFC 2001 for slow start, congestion avoidance, fast retransmit and fast recovery. Our approach does not require any instrumentation of end systems; we only require passively collected network traffic traces to be taken at the end points during the transaction. In this Chapter, we describe our method in detail and show its implementation in a tool called `tcpeval`. `tcpeval` takes as input endpoint traces from `tcpdump` along with implementation-specific constants and, by tracking the evolution of TCP's state as it controls packet flow, determines the critical path for a TCP flow. We then *profile* this critical path, assigning delays to various categories.

We used `tcpeval` to evaluate data collected in WAWM v1.0 to demonstrate its utility. We show the results of critical path analysis for Web transfers using both Linux and FreeBSD TCP implementations from our server cluster to the Denver and Harvard clients. The results show that critical path analysis can give considerable insight to the root causes of delays in Web transfers. In particular, we find that for the systems we have measured in WAWM v1.0, server load is the major determiner of transfer time for small files, while network load is the major determiner for large files; and we are able to characterize the points in the HTTP transaction where server delays are most often introduced. We also show that critical path analysis can pinpoint sources of variability in Web transfers: while packet losses are the primary contributor to variability of transfer time, we also find that the number of packets on the critical path can show very high variability.

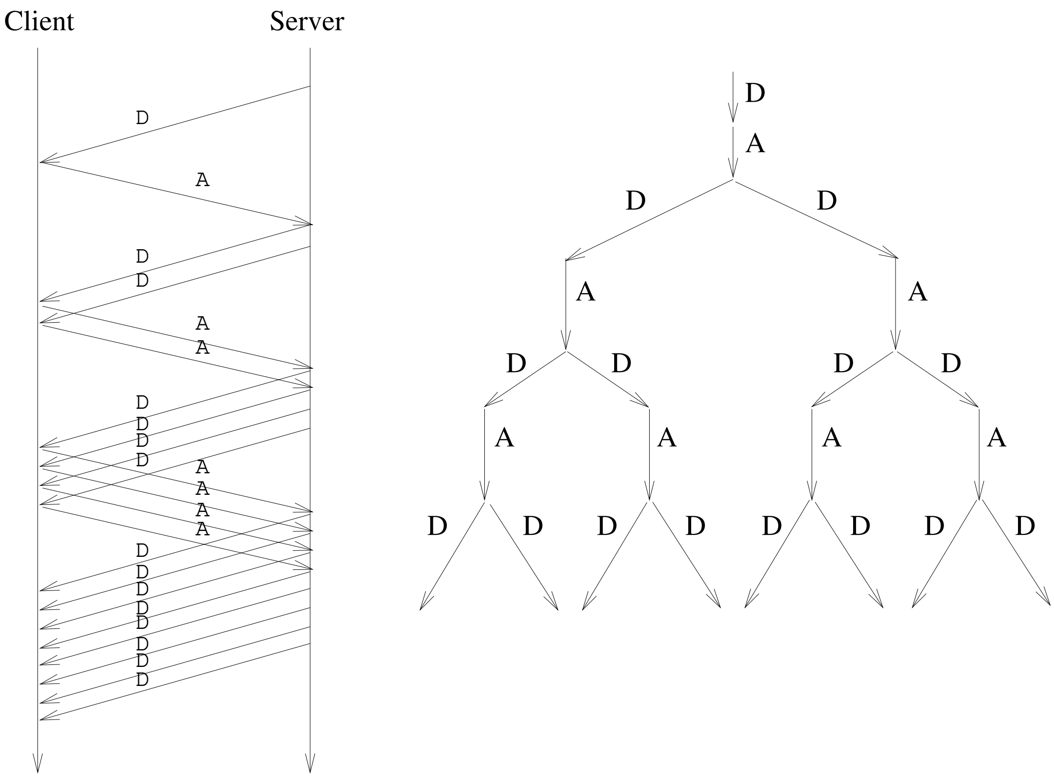
We also find that `tcpeval` can be useful as a diagnostic tool. In the results discussed in Section 7.3 we see that for 20KB files, delay caused by clients accounts for as much as 10% of the overall delay. This was surprising since the only activities that WAWM clients are engaged in is downloading a single document and running network measurement tests: all activities which should not tax those systems. A careful search of the literature of the Linux v2.0.30 TCP stack uncovered a known bug in the implementation that turned out to be the cause of these delays. In addition to TCP performance bugs, `tcpeval` could be applied to understand and debug performance problems in widely distributed systems.

7.2 Mechanics of Critical Path Analysis

In this section we describe how to discover the critical path for TCP flows; details of the algorithm we use; how critical path discovery can be employed in an HTTP setting; and how we profile the critical path to accurately assess the causes of transfer delays. We also describe the tool `tcpeval` which performs these analyses.

7.2.1 Discovering the Critical Path for a Unidirectional TCP Flow

We start by restricting our attention to the heart of our approach, the analysis of the data transfer portion of a unidirectional TCP flow; in the next section we will discuss bidirectional flows and



Time Line Diagram

Packet Dependence Graph

Figure 7.1: A time line diagram of a TCP flow and its reflection in a Packet Dependence Graph (PDG). The PDG shows which data packets (D) are liberated by individual ACKs (A).

application-level protocols. Critical path analysis of unidirectional flows is based on an examination of the *packet dependence graph* (PDG) for a particular TCP flow.

We define the packet dependence graph as a weighted, directed graph (in fact, a tree) in which each node represents the arrival or departure of a single packet at either endpoint. Thus, each packet that is not dropped en route corresponds to two nodes in the PDG. An illustration of the tree structure of the PDG's is shown in in Figure 7.1. This figure shows an idealization of the slow start portion of TCP. On the left is a traditional time line diagram where the dependencies between data and ACK packets are not explicit. On the right we draw the PDG where dependencies are explicit.

Arcs in the PDG represent a dependence relation corresponding to Lamport's *happened-before* relation [65]; each arc is weighted by the elapsed time between the events that form its start and end points. In this case, the dependence relation is that defined by the semantics of reliable data transport using TCP Reno. There are four kinds of arcs in the graph: first, for each distinct packet that is not dropped, we construct an arc from its departure at one endpoint to its arrival at the other endpoint. Second, for each acknowledgment packet (ACK), we find the data packet containing the last byte being acknowledged by the ACK. We construct an arc from the arrival of that data packet to the departure of that ACK packet, reflecting the fact that the ACK could not have been emitted before the corresponding data packet was received. The third type of arc is from ACK packets to data packets. For each ACK packet, we determine the data packets that it "liberated", and construct an arc from the arrival of the ACK to the departure of each liberated data packet. By "liberating" a data packet, we mean that the arrival of the ACK caused TCP to open its congestion window sufficiently to allow the transmission of the data packet. Tracking the liberation of data packets is central to our approach, and must be done carefully; we discuss it in detail in Section 7.2.3.

In the cases where no packet loss occurs, these three arc types are sufficient. However, when a packet is lost, eventually the packet is retransmitted, and there is a dependence relation between the two events. In this case, we construct an arc from the departure of the dropped packet to the departure of the earliest subsequent retransmission.

Having constructed the weighted tree described by this process, we define the critical path in the usual way; that is, the longest path in the tree starting from the root. Since we are considering a unidirectional flow, the root is the first data packet sent. The result is a (weighted) chain of dependence relations running from the connection's first data packet to its last data packet sent.

7.2.2 Discovering the Critical Path for an HTTP Transaction

The discussion so far has assumed that there are no application level dependencies among events. This is not the case in general; for example, it may be that the departure of data packets sent in one direction may depend on the arrival of data packets traveling in the other direction, due to application-level semantics. In this study we focus on HTTP/1.0 retrievals of static files (without persistent connections), for which this dependence only exists between the last packet comprising the HTTP GET and the first data packet sent in response. Nevertheless, to properly construct the critical path, this dependence must be taken into account. Nothing in our methods precludes the analysis of HTTP/1.1 however our implementation is specific to HTTP/1.0.

Since we do not want to restrict the applicability of our methods to any particular application, we do not attempt to explicitly incorporate application-level dependence into the PDG. Instead, we construct the complete critical path by breaking the entire HTTP transaction into four parts: 1) connection establishment — the SYN, SYN-ACK, ACK sequence; 2) the packet sequence delivering the HTTP GET and associated parameters; 3) the returning bulk data transfer; and 4) connection tear-down — the FIN, ACK, FIN, ACK sequence. This breakdown is shown in Figure 7.2 (details of the bulk transfer portion of the critical path are discussed in Section 7.2.3).

The advantage of this decomposition is that the parts do not overlap; the dependence between each part is clear; and that parts 1, 2, and 4 do not depend on the size of the file, and so can be easily analyzed with special-purpose code. Each of parts 1, 2, and 4 follows one of a small set of patterns, and the patterns only vary when packets are lost, which is easily accounted for. Thus, discovering the critical path for HTTP/1.0 transactions requires relatively straightforward analysis of the connection setup, file request, and connection tear down sequences, combined with the unidirectional-flow algorithm described in the previous section for bulk data transfer.

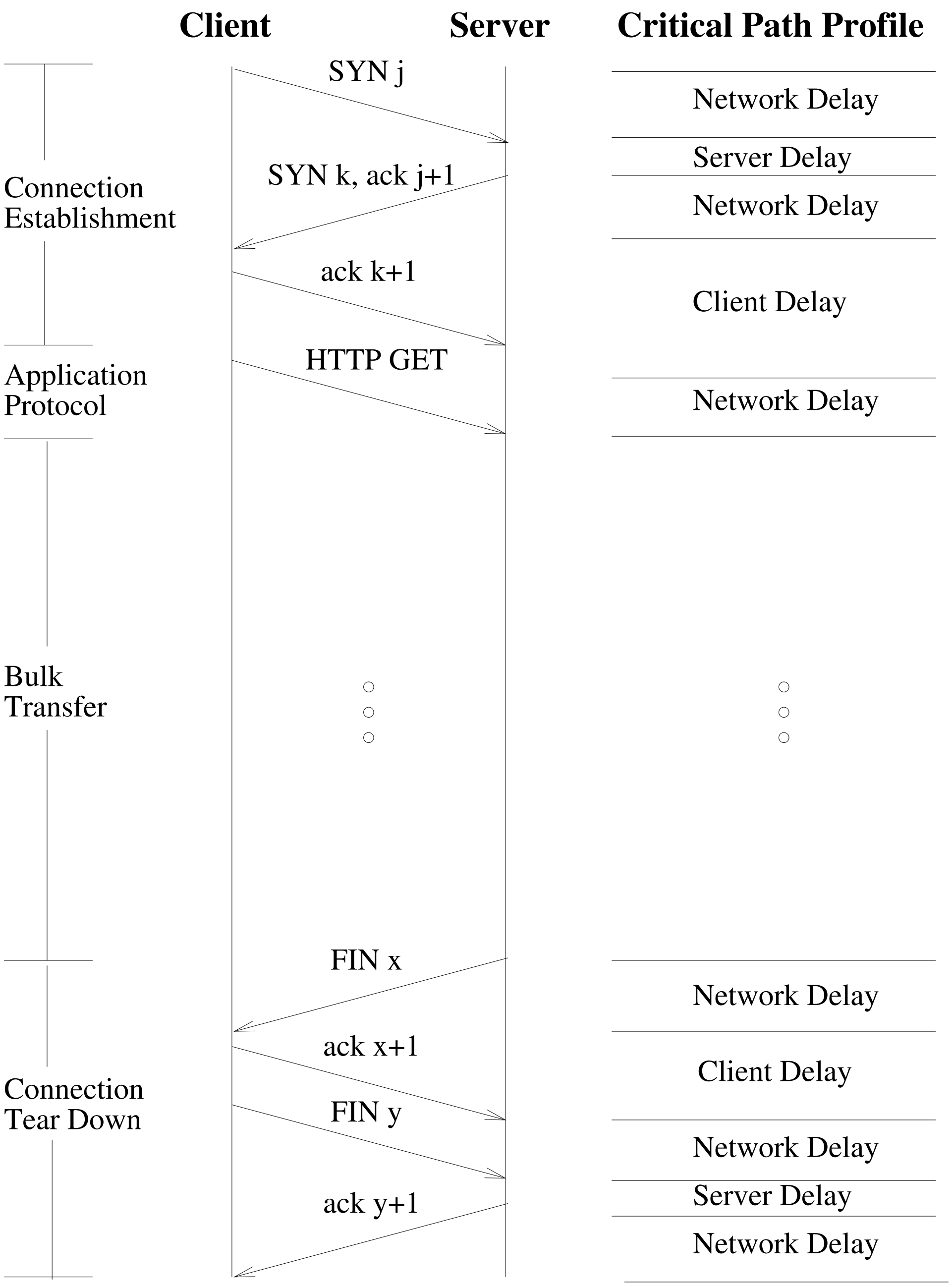


Figure 7.2: Critical path analysis for TCP connection setup, HTTP GET and TCP connection tear down.

7.2.3 Implementing Critical Path Discovery for Bulk Data Transport in TCP Reno

The critical path discovery algorithm for TCP Reno described in Section 7.2.1 can be implemented in two stages: a forward pass over the packet trace and a backward pass up the critical path.

The purpose of the forward pass is to construct a *round* for each non-duplicate ACK. Each round consists of the ACK and the data bytes which are liberated by that ACK. The data bytes liberated by a specific ACK are determined by the state of the receive and congestion windows on the server when the ACK was received. We identify the data bytes liberated by each ACK by simulating the receive and congestion window states based using the TCP Reno slow start/congestion avoidance/fast retransmit/fast recovery algorithms as specified in RFC 2001 [121] and described in [119]. The reason for this accounting is that an ACK may be received at the server before it has had a chance to transmit all of the packets permitted by the current window state; thus it is not safe to assume that each data packet was liberated by the most recent arriving ACK. After establishing the bytes liberated by an ACK, we determine the data packet that triggered the ACK (by comparing sequence numbers) and maintain a pointer to that data packet with the round for the ACK. The result is a tree of rounds.

After the forward-pass construction of all rounds, the critical path is determined by tracing back up the tree, starting from the last data packet (before the FIN is sent, or containing the FIN). At each round, we accumulate arcs onto the critical path, and then move to the parent round. This continues until reaching the first round, which will contain the first data packet.

An example bulk transfer for a Linux TCP stack is shown in Figure 7.3. Key steps in critical path analysis are left-to-right: capture of packet traces, analysis of rounds, construction of critical path, and profiling of the critical path. In this figure, rounds 1, 2, and 3 progressively consist of two, three, and four data packets, since the connection is in slow start [58]. The ACK for round 4 only acknowledges two of the four packets transmitted in round 3. Thus, round 4 is responsible for the release of only three more data packets—since from the server's perspective, there are still two data packets in flight and the current size of the congestion window is five—which illustrates the need to maintain the correct receiver and congestion window state to properly construct dependence arcs.

Maintaining the proper receiver and congestion window size in our simulation is complicated when packets are dropped, since the retransmissions signaled by coarse-grained time-outs and fast retransmits call for different congestion window size adjustments. Fortunately, we can distinguish between drops that are recognized by coarse-grained time outs and those which are recognized by the fast retransmit mechanism as follows: all retransmissions are visible (since we have traces from the sender's endpoint) and all duplicate ACK's are also visible; thus when a packet is retransmitted, we know whether it was in response to a coarse-grained timeout or in response to the receipt of 3 duplicate ACKs. We note that this means it is not necessary to simulate TCP's retransmission timer to determine when coarse-grained timeouts occur, which would be very difficult to reconstruct after the fact, and would introduce a number of TCP implementation dependent issues. One potential issue is that there may be rare instances where coarse-grained timeouts are the actual trigger of a retransmitted packet even though a third duplicate ACK has been received. We call these instances *hidden timeouts*. Our mechanism does not currently distinguish hidden timeouts. If they were to occur, our TCP simulation would have larger congestion windows leading to obviously incorrect client delays which can be detected by our critical path analysis tool, `tcpeval`. In the analysis of our sample data, we found no instances of hidden timeouts.

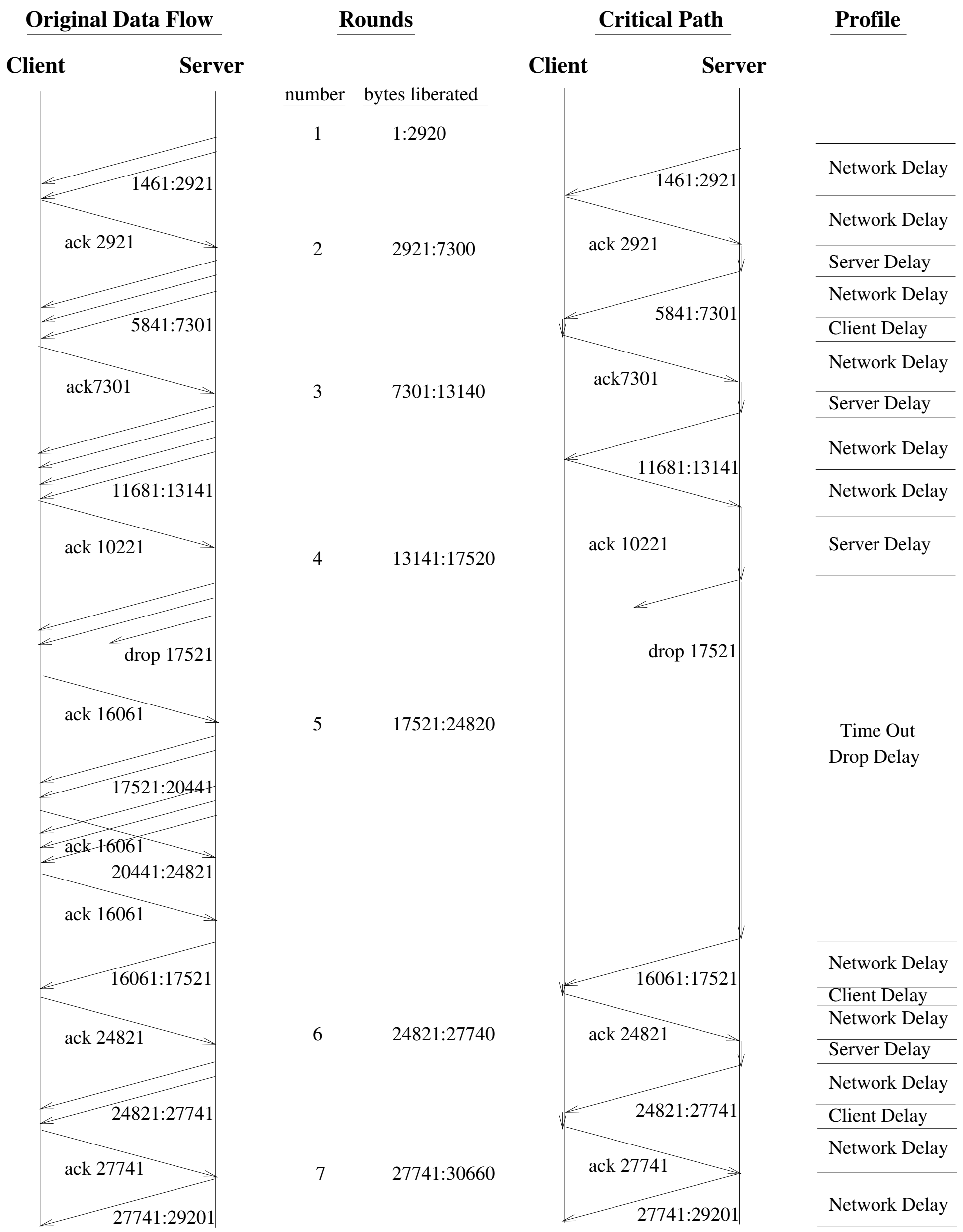


Figure 7.3: Details of critical path analysis for the bulk transfer portion of a unidirectional Linux TCP flow.

The strength of the critical path as an analytic tool is also evident from the figure. Consider once again the data transfer shown in Figure 7.3. The last packet (before the FIN) was released by the ACK in round 7, which is therefore on the critical path. The data which triggered the ACK for round 7 was released by round 6, which is also on the critical path. The ACK in round 6 was triggered by a retransmitted packet, which was originally liberated by round 4. This shows that none of the packets in round 5 are on the critical path, and indeed, even considerable variations in the delivery times of packets in round 5 would have had no effect on the eventual completion time of the overall transfer.

7.2.4 Profiling the Critical Path

Once the critical path has been constructed, we can use it to identify the sources of delays in the data transfer. To do so we assign a natural category to each arc type; the critical path profile is then the total contribution to each category made by arc weights on the critical path. We map arcs to categories as follows: arcs between differently-typed nodes (data → ACK or ACK → data) on the same endpoint are assigned to that endpoint (*i.e.,* **server** or **client**); arcs between equivalently-typed nodes (data → data or ACK → ACK) on opposite endpoints are assigned to the **network**; and arcs between data nodes on the same endpoint are assigned to **packet loss**.[2] These correspond to the intuitive sources of delay in each case. Sample delay assignments can be seen at the far right side of Figure 7.3.

We can refine this classification somewhat. First, it is possible to distinguish between packet losses recognized by coarse-grained timeouts, and those signaled by the receipt of 3 duplicate ACKs from the receiver (fast retransmits). Second, each network delay can be decomposed into two parts: propagation delay and network variation delay. We define propagation delay to be the minimum observed delay in each direction over *all* experiments in which the path did not change. Delay due to network variation is defined as the difference between network delay and propagation delay. It can be caused by a number of things including queuing at the routers along the end-to-end path and route fluctuation.

The result is a six-part taxonomy of delays in HTTP transfers, measured along the critical path: 1) server delays, 2) client delays, 3) propagation delays, 4) network variation delays (which include both delays due to queuing and delays due to any path fluctuations during tests), 5) losses recognized by coarse-grained timeouts, and 6) losses signaled by the fast retransmit mechanism.

7.2.5 Realization of Critical Path Analysis in `tcpeval`

The critical path construction method described in Sections 7.2.1 and 7.2.3 is general in the sense that it only depends on accurate tracking of TCP receiver and congestion window size—so the method works for any implementation that is compliant with RFC 2001. This robustness is evident in Section 7.3 which shows results from two different TCP implementations (Linux 2.0.30 and FreeBSD 3.3). In addition, while we focus on HTTP/1.0 in this paper, other protocols could be evaluated, given accurate modeling of application-level packet dependences.

[2]Some server delay is included in what we call delay due to packet loss since upon either coarse-grained timeout or receipt of a third duplicate ACK, a heavily loaded server could take some time to generate a retransmitted packet.

We have implemented the entire critical path analysis process (path construction and profiling) in the tool `tcpeval` consisting of approximately 4,000 lines of C code. In addition to critical path analysis, it provides statistics for higher level HTTP transaction study (such as file and packet transfer delay statistics) and produces time line and sequence plots for visual analysis.

The principal inputs to `tcpeval` are `tcpdump` traces taken at the client and the server. In addition to the packet traces, the only other inputs to `tcpeval` are the initial values for the congestion window (CWND) and the slow start threshold (SSTHRESH) since these can vary between TCP implementations.

A number of implementation issues arise due to our use of `tcpdump` traces and due to our need to accurately measure one-way packet transit times. These are discussed in the context of automated packet trace analysis by Paxson [103]. Specifically, these are the problems which can occur in packet filters such as the Linux Packet Filter (LPF) or the Berkeley Packet Filter (BPF) which are used by `tcpdump` to extract packet traces. First, packets can dropped by the filter. If `tcpeval` sees an ACK or other response for a packet which was supposedly lost, `tcpeval` concludes it was dropped in the filter rather than the network. Second, packets can be added to the trace by the filter. To address this, if `tcpeval` sees the same packet twice and it was not a retransmit, it simply discards the second copy of the packet. Third, packet filter can report packets in a different order than the occurred in reality. `tcpeval` insures the correct order by sorting packets by time stamp in a number of places during the analysis. Last, packet filters can incorrectly time stamp packets which causes a different type of reordering in the packet trace. At present, `tcpeval` does not account for this. The effect will be that packets will appear to be sent beyond the congestion window which is flagged as an error by `tcpeval`.

With respect to accurate measurements of one-way delays, we have carefully configured NTP [80] (using frequent hard resets and syncing with multiple servers where needed) in our experimental setup to obtain the closest possible clock synchronization. In this way we have found that for the two experimental configurations in Section 7.3, the resulting difference in time between nodes was on the order of 1 ms as reported by NTP. Nonetheless, the nature of our approach demands highly synchronized clocks, so we will configured our systems with GPS-synchronized clocks (enabling synchronization on the order of microseconds) for experiments in WAWM v2.0.

7.2.6 Limitations

There are a number of limitations to the use of critical path analysis in this setting. First, there are some details of the web transactions which we do not take into account. The most obvious is that we do not consider the delay due to domain name server (DNS) resolution. `tcpeval` easily could be enhanced to do this by tracking DNS packets. We also do not consider client browser parse times in our experiments since we simply transfer files as is explained in Section 7.3. The effect of omitting this from our analysis is that delays due to clients could be greater than we report in this thesis.

There are also some aspects of our application of CPA to TCP transactions that limit our ability to use the strengths of CPA. The most fundamental is the inability to do "what if" analysis after establishing the CPA for a TCP transaction. Although one could explore the effects of changing delays or drops on a particular critical path, in reality, changing delays or drop events can change the dependence structure of the entire transaction, (*i.e.,* the PDG). For example, when a drop occurs, it changes the congestion window size and thus the number of packets which are liberated by subsequent ACKs. This fact limits the extent of “what-if” analysis to *simulating* what would happen in the remainder of a transaction instead of actually being able to know exactly what would have happened.

7.3 Critical Path Analysis Applied to HTTP Transactions

In this section we apply critical path analysis to measurements of HTTP transactions taken in WAWM v1.0.

7.3.1 CPA Results for Data Collected in WAWM v1.0

We first examine the resulting critical path profiles for transfers to the Denver client. These results are shown in Figure 7.4. Figure 7.4(a), (b), and (c) show small (1KB), medium (20KB), and large (500KB) file sizes respectively; within each chart the particular experiments are denoted by H or L for high or low load and N or S for network and server.

A number of observations are immediately evident from the figure. At the highest level, it is clear that the category that is the most important contributor to transaction delay depends on a number of factors, including file size and server load. This shows that no single aspect of the system is generally to blame for long transfer times, and that the answer is rather more complicated.

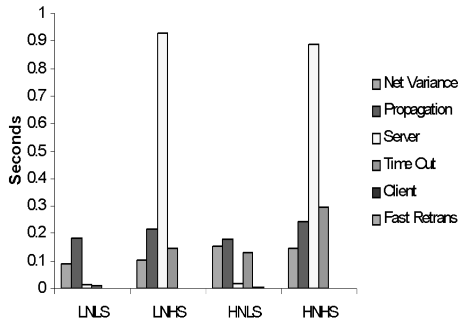
In addition, the figure shows that propagation delay is often the most important overall contributor to transaction delay (*e.g.,* for large files, and for all file sizes when server load is low). This is an encouraging statement about the design of the end-to-end system, since this component of the data transport process is rather hard to avoid. When propagation delay dominates transfer time it would seem that only by opening the congestion window more aggressively could overall end-to-end response time be significantly reduced (while recognizing the difficulty of doing so without contributing to congestion) [98].

Overall, we note that when server load is low, the delays on the client side and the server side are roughly comparable. Client delays throughout tend to be quite low; since the remote site consists of a PC simply transferring a single file, this is consistent with expectations. In addition, this data confirms that there are many more retransmissions triggered by time-outs than by the fast retransmit mechanism in all our cases. This situation has been observed in other studies [13, 50, 96].

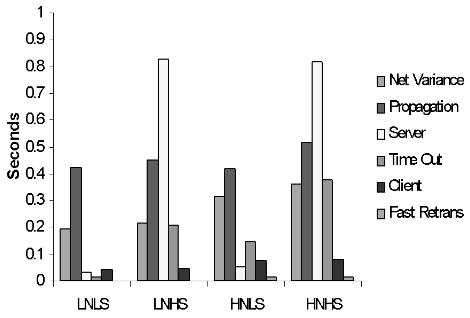
7.3.1.1 Effect of File Size.

Comparing critical path profiles across file sizes, we can make the following observations from Figure 7.4.

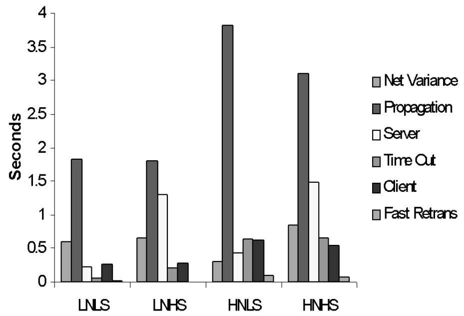
Small files are dramatically affected by load on the server. When server load is low, network delays (network variation and propagation) dominate. However, when server load is high, server delays can be responsible for more than 80% of overall response time. Since small files are so common in the Web, this suggests that performance improvements quite noticeable to end users should be possible by improving servers’ delivery of small files. For *medium sized files,* network delays also dominate when server load is low; but when server load is high, the contribution to delay from the network (network variation and propagation) and the server are comparable. So



(a)



(b)



(c)

Figure 7.4: Aggregate mean contributions to transfer delay from Denver client for (a) small file, (b) medium file and (c) large file. LNLS = low network load, low server load, HNHS = high network load, high server load.

for these files, neither component is principally to blame. Finally, for *large files,* delays due to the network dominate transfer time, regardless of server load.

7.3.1.2 Effect of Server Load.

By examining individual critical paths, we can observe that the delays due to server load are not spread uniformly throughout the transfer. In fact, for small and medium files, nearly all of the server delay measured in our HTTP transactions occurs between the receipt of the HTTP GET at the server and the generation of the first data packet from the server. In Figure 7.5 we plot some typical critical paths for small and medium files, comparing the cases of low and high server load (network load is high throughout).

Each diagram in the figure shows the critical path for the transfer of a single file, with time progressing downward from the top, and lines representing dependence arcs in the PDG. In each diagram, the client is on the left (sending the first packet, a SYN) and the server is on the right. The figure shows that when server load is high, there is a characteristic delay introduced between the moment when the HTTP GET arrives at the server (which is in the second packet from the client, because the second handshake ACK is piggybacked), and when the first data packet is sent out in response. We note that this server start-up delay may be specific to the Apache 1.3.0 architecture; we are currently investigating the case for other servers and versions of Apache.

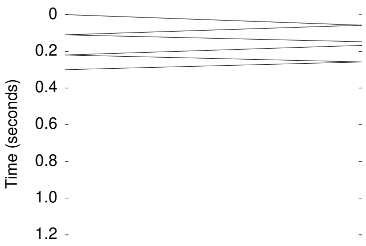
Although these figures clearly show that substantial server delay is introduced at connection start-up, we can also determine that for large files, a busy server introduces considerable delays later in the transfer as well. This can be seen from Figure 7.4 comparing (a) or (b) with (c); the total server-induced delay is twice as large in (c), although the startup delay is independent of file size.

Taken together, these results on the file size and server load indicate that while both servers and networks show startup costs at the beginning of transfers, the startup cost on our server (when under load) is much higher. On the other hand, when approaching steady-state (*i.e.,* during long transfers) the network introduces more delays than does the server.

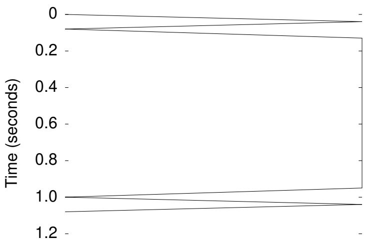
7.3.1.3 Effect of Network Load.

Propagation delay as we have defined it is proportional to number of round-trips on the critical path. For small files, there are 6 packets on the critical path; for medium files, there are typically 14 packets on the critical path; and for large files, there are typically close to 56 packets on the critical path (all of these values are in the absence of losses). One-way delay in both directions between our site and the Denver client was determined (using minimum filtering) to be about 32ms.

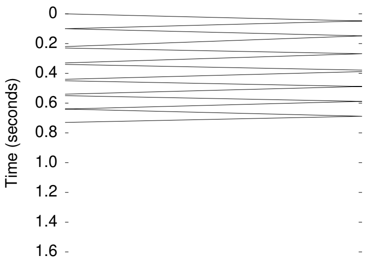
Figure 7.4 shows that for small and medium sized files, propagation delay is independent of network load or server load, which agrees with intuition. However, for large files, propagation delays are higher under high network load. On first glance one might assume this is because of a difference in routes used during tests. This however is not the case. Upon detailed examination of the critical paths, it is apparent that there are actually more round-trips on the critical path on average in the cases of heavy network load. This is because TCP reduces its congestion window dramatically after coarse-grained timeouts. If coarse-grained timeout drops are rare, the effect will be that the congestion window will have time to grow large between drops. If however coarse-



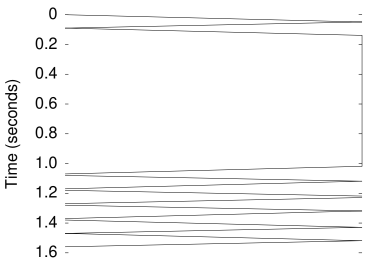
(a)



(b)

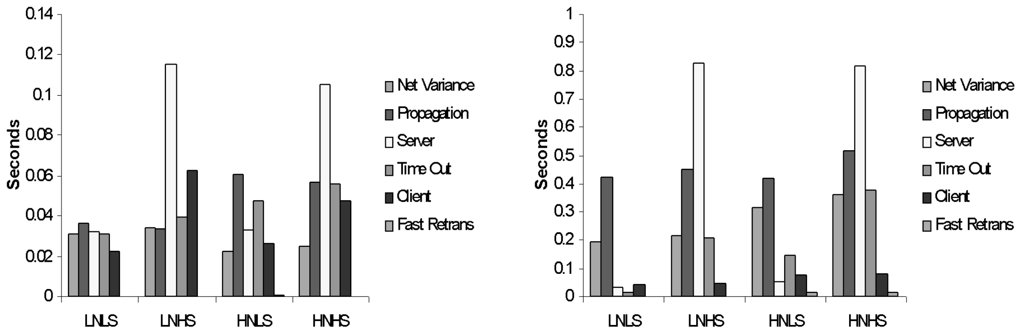


(c)



(d)

Figure 7.5: Critical path diagrams for the transfer of (a) small file, light server load, (b) small file, heavy server load, (c) medium file, light server load, (d) medium file, heavy server load.



(a) (b)

Figure 7.6: Aggregate mean contributions to transfer delay (a) from Harvard client (short path, FreeBSD) and (b) from Denver client (long path, Linux) for medium sized file.

grained timeouts are frequent, the congestion window has less chance to open fully, and so more round-trips appear on the critical path.

In contrast to propagation delay, delay due to network variation generally increases when the network is heavily loaded. However, network variation delay overall is a less important contributor to transfer delay than is propagation delay; and between network variation delay and packet loss (both effects of congestion) neither is generally more significant than the other.

7.3.1.4 Effect of Path Length.

To study the effect of path length on the critical path profile, we ran experiments to the Harvard client. Results for medium sized files only are shown in Figure 7.6; on the left of the Figure we show results for the Harvard client (running Free BSD); on the right of the Figure we repeat the results for the Denver client (running Linux) for comparison purposes.

The figure shows that file transfers occurred almost an order of magnitude faster over the much-shorter path to the Harvard client. In general this increases the importance of server delays to overall transfer time; for example, for the LNLS case, transfers to the Denver client were affected equally by propagation delay and by server delay, while the same transfers to Denver were dominated by propagation delays.

Interestingly, we find that server delays are also radically reduced in the Harvard/FreeBSD setting; this may be due to the different systems software used. We are currently investigating the cause of this effect.

7.3.1.5 Causes of Variability in Transfer Duration.

Finally, while Internet users are typically interested in mean transfer duration, variability in transaction duration is also a source of frustration. Our results so far have presented only mean statistics;

however by studying the variability of each category in our critical path profile we can also shed light on the root causes of variability in transfer duration. Figure 7.7 plots the standard deviation of each category of delay, for all three file sizes transferring to Denver (*i.e.,* this figure is analogous to Figure 7.4).

This figure exposes a number of important aspects of transfer time variability. First, coarse-grained timeout losses are the overwhelming contributor to the variability of transfer delay for small and medium sized files. This is because the duration of even a single timeout is often much larger than the typical total transfer time in the absence of timeouts for these file sizes; thus when coarse-grained timeouts occur, they are very noticeable.

A surprising result is that for large files, when the network is heavily loaded, a significant cause of transfer time variability is propagation delay. This means that the number of packets on the critical path is showing high variability under heavily loaded network conditions. This can be confirmed by examining the distribution of critical path lengths for large files, comparing conditions of low network load with high network load. These histograms (along with each distribution's minimum, mode, and mean) are shown in Figure 7.8.

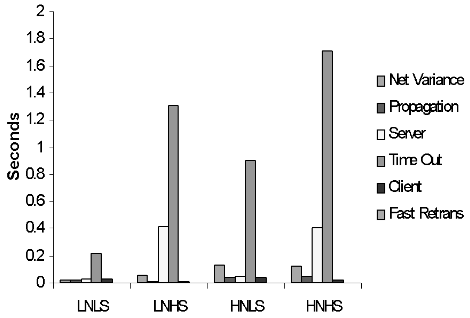
This Figure helps explain why mean propagation delay under high network load is larger (as discussed above); the summary statistics show that the common cases (the distributional modes) are the same, but that under heavy network load the critical path length distribution shows much longer tails.

7.3.2 CPA in WAWM v2.0

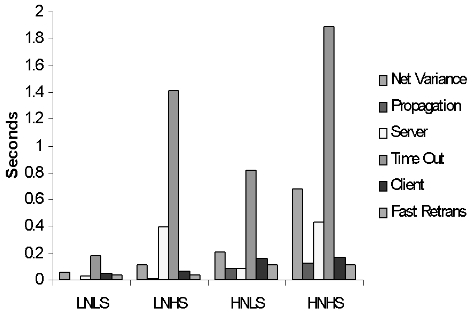
CPA results from measurements taken in WAWMv2 were not yet available at the time of writing. The primary reason for this is that Linux v2.2.14 which is installed on the WAWMv2 client systems implements both the SACK and the NewReno functionality in TCP. SACK is described in [75] and NewReno is described in [50] and specified in RFC 2582 [41]. The SACK option can be toggled in WAWM client systems but NewReno requires a modification to `tcpeval` in order to do CPA on the new data.

NewReno deals specifically with the fast recovery mechanism in TCP. The modification improves the performance of standard Reno without the addition of SACK. The NewReno modification deals with a sender's behavior during fast recovery when multiple packets have been lost. In standard Reno, if the sender receives a *partial ACK* during fast recovery, it is treated as an indication that more than a single packet was lost and the sender reinitiates slow start retransmitting the lost data usually after a timeout. When partial ACK's are received in NewReno, it does not take TCP out of fast recovery. Instead, partial ACK's are treated as a signal that the packet directly following the ACK was lost and the sender retransmits that packet. This enables multiple lost packets to be retransmitted without timeout and without window deflation until all of the lost packets have been acknowledged.

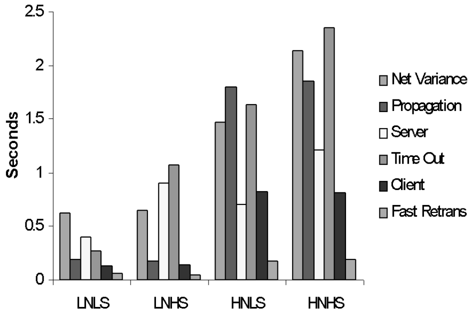
While the NewReno mechanism in principal is a simple change to the window management algorithm in `tcpeval`, it will require careful evaluation to insure that the TCP simulation mechanism in that tool is operating correctly before results can be generated. Since SACK is also available in Linux, will both NewReno and SACK as options in the next version of `tcpeval`.



(a)

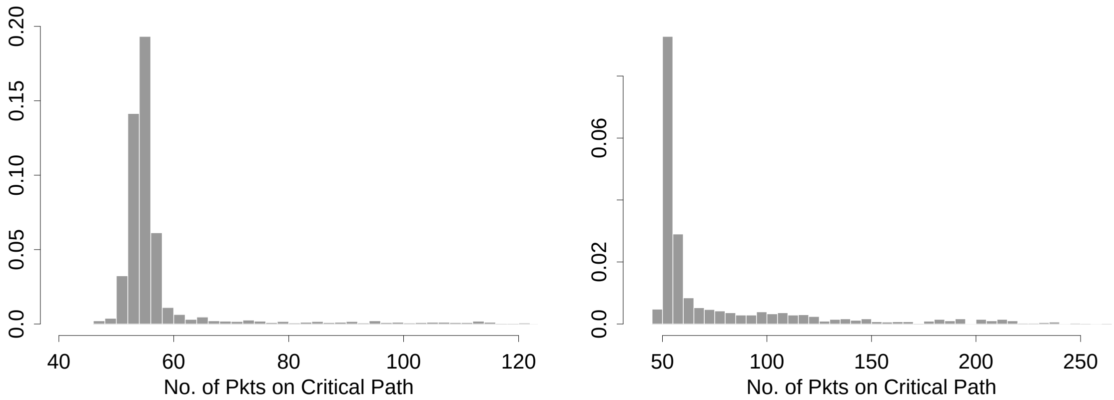


(b)



(c)

Figure 7.7: Standard deviations of delay components for (a) small file, (b) medium file and (c) large file.



(a: min: 47, mode: 55; mean: 58) (b: min: 48 mode: 54; mean: 76)

Figure 7.8: Histograms of Number of Packets on Critical Path for large file: (a) Low network load (b) High network load (both are for the Denver client, high server load).

Chapter 8

Summary and Conclusion

In this thesis we address the problem of determining root causes of performance delays in Web transactions. Until now, researchers have addressed this problem by considering Web components (clients, servers, HTTP and the underlying network) in isolation. While important performance improvements have been made using this approach, future improvements will require that clients, servers HTTP and network be considered as an integrated system. Studying this problem using standard methods such as modeling and simultion is difficult due to the size and complexity of the Web. Our approach was to develop novel methods for taking measuring Web transactions in the wide area and then analyzing them using CPA. In this Chapter we summarize our work and discuss conclusions and future work.

8.1 Workload Generation

We described a tool for generating representative Web requests that is based on analytical models of Web use. We described the characteristics that we feel are important to capture in Web workloads, and why they are important. We showed that satisfying this large set of requirements presents challenges, but that these challenges can be addressed. The resulting tool, SURGE, incorporates the idea of user equivalents as a measure of workload intensity in addition to six distributional characteristics in order to create representative Web workloads.

Our work draws on the wide range of previous research that has characterized Web usage patterns. In addition we present some new characterizations of Web usage that are needed to complete the SURGE model.

We showed that a workload that satisfies all of these characteristics exercises servers in ways quite different from the most commonly used Web workload generator, SPECweb96. In particular, the workload generated by SURGE maintains a much larger number of open connections than does SPECweb96, which results in a much higher CPU load. In addition, we show that SURGE exercises networks differently than SPECweb96. At high loads, SURGE generates network traffic that is self-similar, which does not appear to be true for SPECweb96. Thus SURGE's workloads are more challenging than SPECweb96's to networks as well. These results suggest that accurate Web workload generation is important, since comparison with realistic workloads indicate that traditional workload generators like SPECweb96 may be optimistic in their assessment of system performance.

8.2 HTTP Evaluation

We measured and compared the relative performance impact of HTTP/1.0 and HTTP/1.1 in terms of both transfer latency and file throughput for two popular Web servers. We conducted our experiments in a LAN using an updated version of SURGE that has a more detailed file size model, a session model and eliminates the active OFF time model. We measured the network demands of each protocol by taking packet traces during tests, and used detailed performance monitoring to understand the server demands.

Our study is limited to performance effects that occur in a LAN in which round trip times are short and packet loss is rare. For that reason we do not observe some performance improvements of HTTP/1.1 that would occur in a wide area network setting. However, we do measure quantities such as the server congestion window that help us infer how HTTP/1.1 performs in a wide area setting.

Our measurements confirm and quantify the degree to which HTTP/1.1 reduces network load compared to HTTP/1.0. We show that the persistent connection feature of HTTP/1.1 lowers the number of SYN, FIN and ACK packets transferred significantly. Our measurements of server congestion window size indicate that HTTP/1.1 should enable more efficient utilization of network bandwidth in the wide area. On the other hand, our measurements of the number of data packets transferred and the number of data bytes per packet indicate that the pipelining feature of HTTP/1.1 has very little impact on the network. Our measurements indicate that pipelining does not significantly reduce file transfer latency which differs from previously reported results in [48]. This difference can be traced to the shorter round trip times in a LAN environment and the smaller Web object size used in our study.

We show that there is relatively little difference in performance between HTTP/1.0 and HTTP/1.1 when the CPU is the bottleneck in the system. These results show that the new features added to HTTP/1.1 do not significantly reduce the server's CPU processing load per byte transferred.

We conducted experiments in which we forced the disk system to become the bottleneck. We find that when there is insufficient RAM to accommodate the *entire* data file set, that performance using HTTP/1.1 can be much worse than HTTP/1.0. This is because when using persistent connections, there are many more open connections on average on the server. We show that per connection memory cost can be as high as 81KB per connection on Apache v1.3.0 which means that paging is more likely to occur in HTTP/1.1 when memory is limited. We show that multiple connections per client causes server throughput to degrade compared to a single connection per client.

Our observation of the underutilization of persistent connections in HTTP/1.1 leads us to propose a connection management policy. This policy, which we call early close, closes connections at the end of a Web object transfer. The intention of early close is to increase server throughput, while maintaining the beneficial network effects of persistent connections, when the disk system is the bottleneck. We measure the effect of the early close policy by having clients close their HTTP connections after the transfer of each Web object. Our measurements show that early close generates approximately 17% more network packet traffic than standard HTTP/1.1. However, our measurements show that when the disk system is the bottleneck, server throughput is greatly increased by using early close.

8.3 WAWM Infrastructure

We described two versions of the WAWM infrastructure and test methodology and argued that it represents an approach to wide area measurement experiments that is both novel and useful. Its novelty arises from its treatment of the server and network as an integrated system. Its utility comes from the ability to explore the interaction between server behavior and network behavior, and to identify cases in which the two do not interoperate well.

The WAWM v1.0 infrastructure consisted of a single client system located at the University of Denver which could generate HTTP requests and take active/passive measurements. A server cluster located at Boston University consisted of a Web server, local load generators and measurement systems. The local load generators were connected directly to the LAN. Time synchronization was done by using NTP, synchronizing from the passive measurement system in the server cluster.

The WAWM v2.0 infrastructure was constructed based on lessons learned in WAWM v1.0. The major enhancements consisted of adding 4 new client systems, adding GPS timing to all systems, and running local load generators through the WAN emulator NISTnet. We found that the addition of WAN emulation to the server cluster placed a more variable load on the server in both the light and heavy server load conditions. We found that the addition of GPS timers, while difficult from a deployment perspective (due to the external antenna requirement), enabled clock synchronization three orders of magnitude more accurate than standard NTP. The additional client systems provided pathways that are more diverse from the perspective of latency, backbone carrier and congestion characteristics. This enables the study of a much wider variety of Internet conditions than was available in WAMM v1.0.

8.4 Summary of Performance Results

Tests run in the WAWM v1.0 show that this kind of wide area infrastructure can yield informative and sometimes unexpected measurements results. We showed that (for our server) the main effect of server load on typical transfers is to delay the *first* data packet sent. We show that in many of our experiments, servers under high load suffered significantly *less* packet loss than those under low load. This leads to overall file transfer delays being less for the heavily loaded servers than for the lightly loaded server.

8.5 Critical Path Analysis

We have described methods and a tool for critical path analysis of Internet flows. We have shown how to construct the critical path for unidirectional TCP flows, and how to extend this method to HTTP transactions. Furthermore, we have developed a taxonomy of kinds of delays that can occur in TCP flows and used this taxonomy to profile the critical path.

An important goal of traffic analysis tools is wide applicability, and we have shown that the method we use for constructing the critical path applies to a range of TCP implementations, *i.e.,* those that conform to TCP Reno. Our tool does not require any intrusive instrumentation of end systems, only passively collected packet traces from both endpoints.

Applying this method to HTTP transfers in the Internet helps answer questions like "What are the causes of long Web response times?" We have shown that for the systems we have measured, server load is the major determiner of transfer time for small files, while network load is the major determiner for large files. If found to be general, this would lend support to the development of server enhancements to speed the transfer of small files. Furthermore, critical path analysis uncovers a number of more subtle points. For example, the contribution of propagation delay to the critical path is often greater than that network variability (*e.g.,* queuing); this suggests that it may be difficult to improve transfer latency in some cases without more aggressive window opening schemes such as those discussed in [6, 98] or that better retransmission algorithms which avoid timeouts and slow start [21]. In addition, the dominant cause of variability in transfer time is packet loss; but surprisingly, even the number of packets on the critical path for a constant-sized file can show a very long distributional tail.

Analyzing data taken in WAWMv2 is complicated by the fact that the Linux implementation used in that infrastructure implements the NewReno version of TCP with SACK. While SACK can be turned off on systems, NewReno modifies the fast recovery portion of the protocol such that the current implementation of `tcpeval` will not properly simulate the congestion window size when multiple losses take place in the same transmission window. We are in the process of updating `tcpeval` so that it can run CPA on Reno, NewReno and SACK.

8.6 Future Work

The focus of this work was on developing test methods, a test infrastructure and analysis methods in order to better understand Web performance. While at the highest level, we have provided some insight on Web performance, there is still a long way to go. The clear next steps are to build upon this work and use the infrastructure and tools to do long term measurement and analysis of Web performance. More specifically our future goals are:

1. Broadening the test infrastructure. At present we have deployed 6 client systems which give us an interesting but very small environment from which to run measurement experiments. Expanding the WAWM infrastructure to include a larger number of systems (both clients and additional server clusters) along a wider variety of paths will give us a much more robust and representative environment for taking measurements.

2. Broadening the kinds of measurements made in the infrastructure. At present we take measurements of end-to-end behavior and measurements of end-system behavior. A clear next step is to include measurements taken within the network and at the routing level. This will enable us to analyze exactly where delays and loss are introduced in TCP flows and to understand the effect that routing has on performance.

3. Running tests for a much longer period of time. In order for WAWM measurements to be meaningful on an on-going basis, they need to be taken continuously. This will require new methods of analysis and data reduction since the amount of data collected will be large.

Bibliography

Internet RFCs are published by the USC Information Sciences Institute at Marina Del Rey, CA, and can be obtained from `http://www.isi.edu/publications.html`.

[1] WebBench 2.0. http://www.zdnet.com/zdbop/webbench/webbench.html.

[2] G. Abdulla, E. Fox, and M. Abrams. Shared user behavior on the World Wide Web. In *WebNet97*, Toronto, Canada, October 1997.

[3] A. Adams, J. Mahdavi, M. Mathis, and V. Paxson. Creating a scalable architecture for Internet measurement. *IEEE Network*, 1998.

[4] A. Aggarwal, S. Savage, and T. Anderson. Understanding the performance of TCP pacing. In *Proceedings of IEEE INFOCOM '00*, Tel Aviv, Israel, March 2000.

[5] M. Allman. A Web server's view of the transport layer. *Computer Communications Review*, 30(5), October 2000.

[6] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's initial window. IETF RFC 2414, September 1998.

[7] J. Almeida, V. Almeida, and D. Yates. Measuring the behavior of a World Wide Web server. In *Proceedings of the Seventh IFIP Conference on High Performance Networking (HPN)*, White Plains, NY, April 1997.

[8] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of 1996 International Conference on Parallel and Distributed Information Systems (PDIS '96)*, pages 92–103, December 1996.

[9] N. Anerousis, R. Cáceres, N. Duffield, A. Feldmann, A. Greenberg, C. Kalmanek, P. Mishra, K. Ramakrishnan, and J. Rexford. Using the AT&T labs PacketScope for Internet measurement. AT&T Services and Infrastructure Performance Symposium, November 1997.

[10] J. Apisdorf, K. Claffy, K. Thompson, and R. Wilder. OC3MON: flexible, affordable, high performance statistics collection. In *Proceedings of the USENIX Tenth System Administration Conference (LISA X)*, Chicago, IL, September 1996.

[11] M. Arlitt and T. Jin. Workload characterization of the 1998 World Cup Web site. *IEEE Network*, 14(3):30–37, May 2000.

[12] M. Arlitt and C. Williamson. Internet Web servers: workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, October 1997.

[13] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, and R. Katz. TCP behavior of a busy Internet server: Analysis and improvements. In *Proceedings of IEEE INFOCOM '98*, San Francisco, CA, March 1998.

[14] G. Banga and P. Druschel. Measuring the capacity of a Web server. In *Proceedings of the USENIX '97*, Monterey, CA, December 1997.

[15] G. Banga and J. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the USENIX '98*, New Orleans, LA, June 1998.

[16] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in Web client access patterns. *Special Issue on World Wide Web Characterization and Performance Evaluation; World Wide Web Journal*, December 1998.

[17] P. Barford and M. Crovella. Generating representative workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS '98*, pages 151–160, Madison, WI, June 1998.

[18] P. Barford and M. Crovella. A performance evaluation of hyper text transfer protocols. In *Proceedings of ACM SIGMETRICS '99*, Atlanta, GA, May 1999.

[19] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. IETF RFC 1945, October 1995.

[20] J. Bolot. End-to-end packet delay and loss behavior in the Internet. In *Proceedings of ACM SIGCOMM '93*, San Francisco, Setpember 1993.

[21] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of ACM SIGMETRICS '96*, Philadelphia, PA, May 1996.

[22] H. Braun. A simple method for testing goodness of fit in the presence of nuisance parameters. *Journal of the Royal Statistical Society*, 1980.

[23] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE INFOCOM '99*, New York, NY, March 1999.

[24] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *Proceedings of IEEE INFOCOM '00*, Tel-Aviv, Israel, March 2000.

[25] S. Cheng, K. Lai, and M. Baker. Analysis of HTTP/1.1 on a Wireless Network. Technical report, Stanford University, 1998.

[26] HTTP Client. http://www.innovation.ch/java/HTTPClient/.

[27] The Standard Performance Evaluation Corporation. SPECWeb96. http://www.specbench.org/org/web96/.

[28] M. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[29] C. Cuñha, A. Bestavros, and M. Crovella. Characteristics of WWW client-based traces. Technical Report TR-95-010, Boston University Department of Computer Science, April 1995.

[30] R. D'Agostino and M. Stephens, editors. *Goodness-of-Fit Techniques*. Marcel Dekker, Inc., 1986.

[31] S. Deng. Empirical model of WWW document arrivals at access link. In *Proceedings of the 1996 IEEE International Conference on Communication*, June 1996.

[32] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proceedings of ACM SIGCOMM '00*, Stockholm, Sweden, September 2000.

[33] dummynet. $http://www.iet.unipi.it/\sim luigi/ipdummynet$, 2000.

[34] A. Erramilli, O. Narayan, and W. Willinger. Experimental queueing analysis with long-range dependent packet traffic. *IEEE/ACM Transactions on Networking*, 4(2):209–223, April 1996.

[35] A. Feldmann. Modelling characteristics of TCP connections. Technical report, AT&T Laboratories, 1996.

[36] A. Feldmann. Continuous on-line extraction of HTTP traces from packet traces. Position paper: W3C Web Characterization Workshop, November 1998.

[37] A. Feldmann, R. Cáceres, F. Douglas, and M. Rabinovich. Performance of Web proxy caching in hetrogeneous bandwidth environments. In *Proceedings of IEEE INFOCOM '99*, New York, NY, March 1999.

[38] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: Methodology and experiences. In *Proceedings of ACM SIGCOMM '00*, Stockholm, Sweden, September 2000.

[39] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. IETF RFC 2068, January 1997.

[40] S. Floyd. TCP and explicit congestion notification. *ACM Computer Communications Review*, 24(5):10–23, October 1994.

[41] S. Floyd and T. Henderson. The NewReno modification to TCP's fast recovery algorithm. RFC 2582, 1999.

[42] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[43] S. Floyd and V. Paxson. Why we don't know how to simulate the Internet. In *Proceedings of the 1997 Winter Simulation Conference*, December 1997.

[44] National Laboratory for Applied Network Research. http://www.nlanr.net, 1998.

[45] Cooperative Association for Internet Data Analysis. http://www.caida.org, 1998.

[46] A Distributed Testbed for National Information Provisioning. http://ircache.nlanr.net/Cache/.

[47] H. Frystyk-Nielsen. Libwww. http://www.w3.org/Library/.

[48] H. Frystyk-Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Wium-Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1 and PNG. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, Setpember 1997.

[49] The NPD Group. http://www.npd.com/, 1998.

[50] J. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *Proceedings of ACM SIGCOMM '96*, Palo Alto, CA, August 1996.

[51] J. Hollingsworth and B. Miller. Parallel program performance metrics: A comparison and validation. In *Proceedings of Supercomputing '92*, Nov 1992.

[52] B. Huberman, P. Pirolli, J. Pitkow, and R. Lukose. Strong regularities in World Wide Web surfing. *Science*, 280:95–97, 1998.

[53] C. Huitema. Internet quality of service assessment. ftp://ftp.telcordia.com/pub/huitema/stats/quality_today.html.

[54] Keynote Systems Inc. http://www.keynote.com, 1998.

[55] Lucent NetCare Inc. net.Medic. http://www.lucentnps.com/software/medic/datasheet/index.asp, 1997.

[56] Lucent NetCare Inc. VitalAgent. http://www.lucentnps.com/software/vitalsuite/vitalagent/datasheet/index.asp, 1998.

[57] Internet Protocol Performance Metrics. http://www.advanced.org/ippm/index.html, 1998.

[58] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM '88*, pages 314–332, August 1988.

[59] V. Jacobson. traceroute. ftp://ftp.ee.lbl.gov/traceroute.tar.Z, 1989.

[60] S. Karandikar, S. Kalyanaraman, P. Bagal, and B. Packer. TCP rate control. *Computer Communications Review*, 30(1):45–58, January 2000.

[61] S. Khaunte and J. Limb. Statistical characterization of a World Wide Web browsing session. Technical Report GIT-CC-97-17, College of Computing, Georgia Institute of Technology, 1997.

[62] L. Kleinrock. *Queuing Systems, Volume II: Computer Applications*. John Wiley & Sons, 1976.

[63] B. Krishnamurthy and C. Willis. Analyzing factors that influence end-to-end Web performance. In *Proceedings of the Ninth International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[64] T. Kwan, R. McGrath, and D. Reed. User access patterns to NCSA's WWW server. Technical Report UIUCDCS-R-95-1934, University of Illinois, Department of Computer Science, February 1995.

[65] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[66] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, pages 2:1–15, 1994.

[67] B. Liu and E. Fox. Web traffic latency: Characteristics and implications. In *WebNet98*, Orlando, FL, November 1998.

[68] Z. Liu, N. Niclausse, and C. Jalpa-Villanueva. Web traffic modeling and performance comparison between HTTP/1.0 and HTTP/1.1. In *System Performance Evaluation: Methodologies and Applications*, CRC Press, August 1998.

[69] Z. Liu, N. Niclausse, and C. JalpaVillenueva. WAGON Web trAffic GeneratOr and beNchmark. $http://www-sop.inria.fr/mistral/personal/Zhen.Liu/$, 1999.

[70] K. Lockyer. *Introduction to Critical Path Analysis*. Pitman Publishing Co., New York, N.Y., 1964.

[71] B. Mah. An empirical model of HTTP network traffic. In *Proceedings of INFOCOM '97*, Kobe, Japan, April 1997.

[72] C. Maltzahn, K. Richardson, and D. Grunwald. Performance issues of enterprise level Web proxies. In *Proceedings of ACM SIGMETRICS '97*, Seattle, WA, June 1997.

[73] S. Manley, M. Courage, and M. Seltzer. A self-scaling and self-configuring benchmark for Web servers. Harvard University, 1997.

[74] S. Manley and M. Seltzer. Web facts and fantasy. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[75] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. IETF RFC 2018, 1996.

[76] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communications Review*, 27(3), July 1997.

[77] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques and storage hierarchies. *IBM Systems Journal*, 9:78–117, 1970.

[78] R. McGrath. Measuring the performance of HTTP daemons. http://www.ncsa.uiuc.edu/InformationServers/Performance/Benchmarking/bench.html, 1996.

[79] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[80] D. Mills. Network time protocol (version 3): Specification, implementation and analysis. IETF RFC 1305, 1992.

[81] D. Mills. A kernel model for precision time keeping. IETF RFC 1589, 1994.

[82] D. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networking*, 3(3):245–254, June 1998.

[83] Mindcraft. WebServer Comparison: Microsoft Windows NT Server 4.0 on a Compaq ProLiant 3000 and Sun Solaris 2.6 with Sun WebServer 1.0 on a Sun Ultra Enterprise 450. http://www.mindcraft.com/whitepapers/.

[84] J. Mogul. The case for persistent-connection HTTP. Technical Report WRL 95/4, DEC Western Research Laboratory, Palo Alto, CA, 1995.

[85] J. Mogul. Network behavior of a busy Web server and its clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA, 1995.

[86] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, Setpember 1997.

[87] S. Moon, P. Skelley, and D. Towsley. Estimation and removal of clock skew from network delay measurements. In *Proceedings of IEEE INFOCOM '99*, New York, NY, March 1999.

[88] D. Mosberger and T. Jin. httperf – a tool for measureing Web server performance. In *SIGMETRICS Workshop on Internet Server Performance '98*, Madison, WI, June 1998.

[89] E. Nahum, J. Almeida, M. Rosu, and S. Seshan. The effects of wide-area conditions on WWW server performance. Currently under review for publication, November 2000.

[90] Netcraft. Netcraft Web server survey. http://www.netcraft.co.uk/, August 1998.

[91] Collaborative Advanced Interagency Research Network. http://www.cairn.net.

[92] Coral: Passive network traffic monitoring and statistics collection. $http://www.caida.org/tools/coral.$

[93] NLANR Acitve Measurement Program - AMP. http://moat.nlanr.net/AMP.

[94] NLANR Passive Measurement and Analysis - PMA. http://moat.nlanr.net/PMA.

[95] University of Minnesota. Gstone version 1. http://web66.coled.umn.edu/gstone/info.html.

[96] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of ACM SIGCOMM '98*, Vancouver, Canada, Setpember 1998.

[97] J. Padhye and S. Floyd. Identifying the TCP behavior of Web servers. Technical report, ACIRI, 2000.

[98] V. Padmanabhan and R. Katz. TCP fast start: A technique for speeding up Web transfers. In *Proceedings of the IEEE GLOBECOM '98*, November 1998.

[99] V. Padmanabhan and J. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28:25–35, December 1995.

[100] V. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[101] K. Park, G. Kim, and M. Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *Proceedings of the Fourth International Conference on Network Protocols (ICNP '96)*, pages 171–180, October 1996.

[102] V. Paxson. Empirically-derived analytic models of wide-area TCP connections. *IEEE/ACM Transactions on Networking*, pages 316–336, August 1994.

[103] V. Paxson. Automated packet trace analysis of TCP implementations. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[104] V. Paxson. End-to-end Internet packet dynamics. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[105] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California Berkeley, 1997.

[106] V. Paxson. On calibrating measurements of packet transit times. In *Proceedings of ACM SIGMETRICS '98*, pages 11–21, Madison, WI, June 1998.

[107] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. IETF RFC 2330, 1998.

[108] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[109] S. Pederson and M. Johnson. Estimating model discrepancy. *Technometrics*, 1990.

[110] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipies in C*. Cambridge University Press, second edition, 1992.

[111] Apache HTTP Server Project. http://www.apache.org, 1998.

[112] The Surveyor Project. http://www.advanced.org/csgippm/, 1998.

[113] Network Time Protocol Server Frequently Asked Questions. $http://www.eecis.udel.edu/\sim ntp/ntpfaq/$.

[114] M. Schroeder and M. Burrows. Performance of firefly RPC. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, Litchfield Park, AZ, December 1989.

[115] SSH Communications Security. $http://www.ssh.fi$, 1999.

[116] Network Time Protocol Server. $http://www.eecis.udel.edu/\sim ntp$.

[117] T. Shepard. TCP packet trace analysis. Master's thesis, Massachusetts Institute of Technology, 1990.

[118] Skitter. http://www.caida.org/tools/measurement/skitter.

[119] W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[120] W. Stevens. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. Addison-Wesley, 1996.

[121] W. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. IETF RFC 2001, January 1997.

[122] The Linuxtrace Toolkit. http://www.linuxtrace.org, 2000.

[123] G. Trent and M. Sake. WebSTONE: The first generation in HTTP server benchmarking, February 1995. Silicon Graphics White Paper.

[124] UCB/LBNL/VINT Network Simulator - ns (version 2). http://www.isi.edu/nsnam/ns/, 2000.

[125] Route Views. University of oregon. http://www.antc.uoregon.edu/routeviews.

[126] WebCompare. http://webcompare.iworld.com/.

[127] Webjamma. http://www.cs.vt.edu/~chitra/webjamma.html.

[128] NISTnet wide area network emulator. http://snad.ncsl.nist.gov/itg/nistnet/, 2000.

[129] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson. Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, February 1997.

[130] M. Yajnik, S. Moon, J. Kurose, and D. Towsley. Measurement and modeling of temporal dependence in packet loss. In *Proceedings of IEEE INFOCOM '99*, New York, NY, March 1999.

[131] C. Yang and B. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of 8th International Conference on Distributed Computing Systems*, San Jose, CA, June 1997.

[132] G. Zipf. *Human Behavior and the Principle of Least-Effort*. Addison-Wesley, Cambridge, MA, 1949.

Vita

Paul Robert Barford was born in Milwaukee, Wisconsin U.S.A. on June 12, 1963 to Judith Ann Barford and Robert Barford. Paul was followed by his brother Miles Daniel, his brother Ian Peter and his sister Mary Florence Virginia.

After living briefly in Milwaukee, WI; Sioux City, IA; and Bloomington, IN, the Barfords settled in Charleston, IL where Paul attended primary and secondary school. Paul spent his freshman year, and parts of his sophomore and junior years attending the The Anglo-American School of Casteldefells, Spain.

After high school graduation in 1981, Paul entered Illinois State University in Normal, IL as a freshman in the pre-engineering program. At ISU, Paul served as an undergraduate teaching assistant in both the department of economics and the department of mathematics. Paul was a member of the ISU honors program and a number of undergraduate honor societies. Paul transferred to the University of Illinois in Urbana, IL in the fall of 1983, and entered the electrical engineering program. Paul graduated from the University of Illinois with a B.S. in electrical engineering with honors in May, 1985.

Paul worked as an instructor teaching calculus in the mathematics department at the University of Illinois in Urbana, IL from 1985 through 1987. During that time, Paul also pursued a masters degree in electrical engineering at U of I completing all requirements except his thesis.

In the summer of 1987, Paul began working for Digital Equipment Corporation in Maynard, MA as a senior engineer in their electrical computer aided design group. Paul worked on various projects during his tenure at DEC and received a number of awards for outstanding contributions.

In the summer of 1991, Paul began working for DeGeorge Financial Services Corporation in Cheshire, CT as their director of corporate planning and research. For the next four years, Paul worked on a variety of both technical and non-technical projects.

In the Fall of 1995, Paul entered the post-bachelor's Ph.D. program in the Department of Computer Science at Boston University. Paul was a member of the OCEANS, Commonwealth and Wing research groups at BU and was supported by a number of different grants. Paul also joined the technical advisory boards of epicRealm, Inc., and Netheartbeat, Inc., and consulted for a number of other companies while at BU. His Ph.D. defense took place in December of 2000.

Athletics have always been a major part of Paul's life. He competed as a professional triathlete and duathlete from 1987 to 1992 and has continued to compete as an amateur in both sports since. Paul is a two time overall national duathlon champion, a 16 time USA Triathlon All American, a 5 time US National Team member and has been ranked first overall in duathlon and second in triathlon nationally. Paul also served for three years on the USA Triathlon board of directors.

Paul will become a member of the Computer Science faculty at the University of Wisconsin at Madison in January, 2001.

`barford@cs.bu.edu`
December 15, 2000
Boston, Massachusetts