LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-494

TCP Packet Trace Analysis

Timothy Jason Shepard

February, 1991

545 TECHNOLOGY SQUARE; CAMBRIDGE, MASSACHUSETTS 02139 (617) 253-5851

TCP Packet Trace Analysis

by

Timothy Jason Shepard

Submitted to the Department of
Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements
for the Degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
June, 1990

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the National Aeronautics and Space Administration under Contract No. NAG2-582.

TCP Packet Trace Analysis

by

Timothy Jason Shepard

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1990
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

ABSTRACT

Examination of a trace of packets collected from the network is often the only method available for diagnosing protocol performance problems in computer networks. This thesis explores the use of packet traces to diagnose performance problems of the transport protocol TCP. Unfortunately, manual examination of these traces can be so tedious that effective analysis is not possible. The primary contribution of this thesis is a graphical method for displaying the packet trace which greatly reduces the tediousness of examining a packet trace.

The graphical method is demonstrated by the examination of some packet traces of typical TCP connections. The performance of two different implementations of TCP sending data across a particular network path is compared. Traces many thousands of packets long are used to demonstrate how effectively the graphical method simplifies examination of long complicated traces.

In the comparison of the two TCP implementations, the burstiness of the TCP transmitter appeared to be related to the achieved throughput. A method of quantifying this burstiness is presented and its possible relevance to understanding the performance of TCP is discussed.

Thesis Supervisor: David D. Clark
Title: Senior Research Scientist

Acknowledgments

A reimplementation of the Sun `nit` socket in 4.3BSD Unix by *Mark Rosenstein* of Project Athena was used in the packet trace collection system.

Van Jacobson identified the appropriate two-dimensional space in which to plot.

My thesis advisor *David Clark* provided guidance, support, a stimulating environment, and the freedom to do my own thesis. Working with him and in his group has been an enriching experience.

I have had many enjoyable discussions with many interesting people in LCS on a wide range of topics. These distractions and mini-projects are an important part of my education, and I thank everyone who has been a part of the stimulating environment around me.

Andrew Heybey, my officemate for three years, has tolerated my randomness and distracting behavior.

James Davin proofread the bulk of the thesis and pointed out many grammatical errors and twisted sentences.

My dear friend *Jennifer Gleason* heard "No. I can't. I have to work on my thesis." far too many times.

Many thanks to my parents, *Ralph* and *Carol Shepard*, and sister, *Cynthia Shepard*, for their support, love, and continued interest in me. I am very fortunate to have a wonderful family.

This document was typeset with the LaTeX Document Preparation System.

Contents

This page intentionally left blank.

Chapter 1

Introduction

This thesis presents some new tools and methods of data presentation that enable empirical analysis of a widely used transport protocol: the DARPA Internet's Transmission Control Protocol (TCP). Diagnosis of TCP performance problems using this method of presentation is faster than with previous tools (or more accurately, the lack of tools) and will allow a much more comprehensive and detailed understanding of an analyzed connection than was previously possible.

1.1 Perspective

An engineering cycle can be used to describe the evolution of a technology. The typical cycle involves observing what currently exists, deciding what is wrong with it or what improvements can be made, designing the improved version, and then testing and deploying the new system. Wide area packet-switched networks for computer communication such as the Arpanet and its successor, the DARPA Internet, have existed for over 20 years, but have only been through about two engineering cycles. These systems have long engineering cycles because of the scale of these systems and the large amount of time required to deploy the necessary network interfaces and software in a wide variety of computer systems.

Because of current efforts at standardization of the next generation of network interfaces and communication protocols such as the ISO/OSI protocols and the CCITT B-ISDN standards it is timely to take a close look at the operation of current networks and protocols. This thesis contributes to the process of examining the performance of existing protocols.

1.2 TCP and Performance

The widely used DARPA Internet Protocols (which are also known as *the TCP/IP protocol suite*) were originally designed to allow various network technologies to be interconnected into one large internet, the *DARPA Internet*. TCP is the principal transport protocol in this suite.

TCP provides for an end-to-end reliable byte stream network connection over a datagram network. A single TCP connection provides a pair of byte streams between the two end points of the connection, one in each direction. A TCP connection is identified by the IP addresses and the port numbers at each end of the connection. TCP modules, one at each end of a TCP connection, communicate with each other by sending *TCP segments* between themselves. TCP segments are carried by IP (Internet Protocol) packets through the packet-switched network. The TCP modules provide the end-to-end reliable byte stream by arranging for the transmission, sequencing and acknowledgment of bytes. They also provide automatic detection and recovery from lost packets by using timers and retransmissions. TCP is fully described in [16], and IP is fully described in [14].

Because TCP needed to work over a wide variety of networks and was expected to provide a variety of types of service, the TCP protocol specification leaves some of the details to be decided by each implementor. Interoperability was the chief goal of the TCP specification. Performance issues such as window sizes, how quickly segments should be sent, and whether to try to batch acknowledgments by dallying were left almost entirely to the designer of a particular implementation [5].

The *performance* of a TCP connection is the collection of behavior of the TCP implementations involved in the connection that may affect data throughput, efficient use of bandwidth, timely recovery from lost packets, and interaction with other resources in the network. *Performance problems* are the result of poor behavior of an implementation on one or more of these points.

TCP has indeed achieved a high degree of interoperability. Today it is rare to find two different TCP implementations that are unable to connect and carry data between them. However, it is not as rare to find TCP connections performing poorly. Poor performance can be caused by a variety of problems. The assumptions made by the implementor might not match the the network being used or the assumptions made by the implementor of the other TCP. The network may even be failing in some way to deliver enough of the packets to the remote TCP. When there is a problem short of a total failure of the net to deliver packets, TCP will continue to interoperate and the only usual indication that there is a problem will be reduced performance.

The only measures of performance typical computer network users have are how quickly responses return from a remote element in the network (such as remote echoing of typing) and the total time taken to transfer a some amount of data from one host to another. These correspond roughly to the round trip time and the achieved total throughput. These do give the user some means of observing performance, but only the most sophisticated users of a large wide-area network or a large campus network would likely know what performance to expect from the network.

TCP's robustness can effectively hide failures, even bugs, of the network and of the TCP implementations themselves. Evidence presented in Chapter 3 will show that this type of hidden failure does occur, perhaps somewhat frequently. Worse yet, most all TCP implementations hide these failures silently. There are no red warning lights that light up when TCP takes action to retransmit a packet or has to send every packet two, three, or more times to get it through. Users probably would not want to see repeatedly such warning message and few users would be able to take action to correct the problem.

Thus there is a problem today in networks using the TCP/IP protocols. Bugs manifest themselves as performance problems. Few users know what performance can be expected of a network and therefore are unaware of the presence of bugs. TCP's ability to continue to service a connection when things are going wrong allows bugs to remain hidden. Detecting these bugs requires methods of observing and determining the performance of TCP connections.

This thesis explores the examination of packet traces collected from a network to determine in detail the performance of TCP connections.

1.3 Organization of this Thesis

In Chapter 2 a method of displaying the trace of a TCP connection graphically is used to greatly reduce the time and effort required to understand in detail the activity of the connection. This method of displaying the trace is the chief intellectual contribution of this thesis. Many packet traces have been examined using this graphical tool, and some of the more interesting packet traces will be presented in Chapter 3 to demonstrate the usefulness of this tool and to highlight this tool's ability to allow rapid discovery of interesting phenomena buried deep inside of packet traces. Chapter 4 addresses this tool's chief limitation: it still requires the human to examine and view the packet trace as a sequence of events in time. Chapter 4 looks in detail at the burstiness of the TCP transmitter and presents a second graphical method which displays the burstiness of TCP transmissions. Chapter 5 concludes the thesis with a few suggestions for network implementors, managers, and troubleshooters and points to where future research in transport protocol transmission methods should be directed.

An important part of a network analyst's assets is his or her box of tools. Appendix

A details some of the tools constructed while carrying out the research for this thesis. An important property of these tools were that the packet gathering ran continuously throughout this project so that the capture for analysis of a particular packet trace did not have to be premeditated. Also important was that these tools ran under Unix. This allowed use of the standard Unix data manipulation utilities to rapidly prototype utilities.

1.4 Related Work

Studies of Networks and Protocols

Related work includes studies to characterize the traffic of operational networks and systems built to collect data for such studies. In [10], Jain and Routhier developed a model for traffic on a token ring network using data collected by a monitoring system built by Feldmeier [6]. More recently, Braden and DeSchon have developed a system NNStat [3] for gathering statistics from the Internet for traffic studies. All of these studies have been concerned with studying traffic in the network as a whole and do not examine traces of individual connections in detail.

The performance of transport protocols has been studied in simulation. Hashem [7] studied the effect of gateway policy on performance in a simulated network carrying TCP connections using Jacobson's slow-start and other algorithms.

More closely related to this thesis are two projects which studied the performance of transport protocols by collecting and examining data of actual protocol operations. Sanghi et. al. instrumented a particular TCP to collect a trace of its state [17]. Aronoff et. al. built an instrumented testbed where transport protocols can be developed and studied [1]. Both of these systems allow the conduct of experiments and the collection of data for analysis but are not directly applicable to general monitoring of operational networks.

Improvements to TCP

Another category of related work includes studies to improve the understanding of transport protocols by using collected data. Motivated by reports of poor performance in the Internet, Jacobson and Karels developed a collection of algorithms, including the slow-start algorithm, for improving the behavior of the TCP in 4.3BSD Berkeley Unix [9]. The graphical method presented in Chapter 2 was inspired by time-sequence plots created by Jacobson. Mankin and Thompson studied the performance of the slow-start algorithm using data collected from a host and an Internet gateway [12].

Packet Trace Analysis

The most closely related work is that of Hitson [8]. Hitson recognized the difficulty of packet trace analysis (as is shown in Chapter 2 of this thesis) and tackled the difficulty using expert-system techniques. Hitson's goals are the same as goals of this thesis, but the techniques are different. Hitson uses automated analysis and encodes the knowledge necessary to do the analysis. In this thesis the use of tools to enhance the effectiveness of manual analysis is emphasized.

Chapter 2

TCP Packet Trace Analysis

This chapter will examine the process of analyzing a packet trace to determine the performance behavior of a Internet's transport layer protocol, TCP. Analysis of the packet trace is often the only method of determining why a TCP connection is behaving or performing oddly. This chapter will show that analysis of a TCP packet trace is problematic because of its tediousness. This chapter will then show that a novel transformation of the packet trace into a graphical form makes the analysis much easier.

TCP (Transmission Control Protocol) is defined by RFC793. It is a protocol which provides a bidirectional reliable byte stream on top of IP (Internet Protocol), an unreliable datagram service. TCP is an appropriate protocol to analyze because its use is common among workstations and timesharing computers at universities and research laboratories.

2.1 Packet Trace

A common method of performance problem diagnosis of TCP connections is manual examination of a trace of the relevant TCP packets collected from either a metered implementation of TCP or from some sort of network monitoring device. The relevant fields of the packets in the trace along with a timestamp are usually printed. By examining such a trace, the skilled analyst can reconstruct much of the story behind the trace and infer the reasons for the packets and the contents of their control fields. Often, insight into the cause of performance problems can be gained from constructing this story.

This method leaves the human analyst to do all of the data reduction of the analysis. The analyst must reconstruct the story from raw data. Hitson [8] discusses the difficulties of leaving the human with the packet trace in this raw form and states that it took human experts about 20 minutes to examine such traces in detail. My early experience showed that when faced with a real performance problem, two experts working together could easily spend an hour attempting to understand a section of a TCP packet trace no more than a few dozen packets long.[1]

The problem is that when examining the packet trace, the human spends much time trying to reconstruct the purpose of each packet and how each packet fits in with the packets near it in the trace. An example of this process would be identifying the packet that carries the first acknowledgment covering the data carried in some previous packet. Another example would be identifying which packets carry retransmissions of data already sent by a previous packet. One of the harder things to reconstruct from a line-by-line listing of the packet trace is the relative temporal relationships between all the packets. This sort of manual analysis of packet traces leaves for the human the tedious task of reconstructing these relationships between the packets.

One approach to solving this problem would be to attempt to describe the process the human expert uses precisely enough so that the expert's knowledge and methods can be captured and used by some sort of automated analysis tool. This tool would

[1] But TCP packet traces thousands of packets long are typical for connections involving bulk data transfer. It would be hard to claim after an hour that the few dozen packets examined were representative of the entire connection without looking at the remaining packets. A thorough analysis would take a long time.

scan the packet trace, identifying and classifying common localized phenomena in the trace, and could either annotate or produce a higher level description of the trace. Carrying this approach further hopefully would lead to an expert system capable of assisting non-experts in diagnosing and isolating performance problems of TCP.

The difficulty with this approach lies in the difficulty of describing how the human expert performs the task. The performance of an expert-system used to perform this task would depend on the coverage of the knowledge base. Experience examining packet traces with the tools presented in this thesis suggests that this database would never be complete.

The fundamental problem is that the packet trace is an unwieldy ocean of numbers when it is presented in its raw form. The analyst, when confronted with this sea of numbers, suffers from information overload. This information overload motivates others to reduce the amount of information presented to the user by means such as statistical analysis and expert systems.

The approach presented here is to improve the form in which the data is presented to the human analyst, without trying to embed knowledge about the analysis into the tools. By improving the form in which the trace is presented to the analyst and by providing the analyst with some tools to manipulate this form, the problem of information overload can be dramatically reduced.

2.2 Details of TCP

An IP network provides a simple unreliable message (datagram) delivery service where each packet is routed independently and the network maintains no important state about the connections through it. An IP network makes no guarantee to deliver a packet reliably. IP datagrams may be lost, reordered, corrupted, or even duplicated. By using checksums, sequence numbers, acknowledgments, and windows, TCP provides a reliable byte stream with end-to-end flow control.

A TCP connection provides a bidirectional connection where the acknowledgments and window updates in one direction can be carried along with the data going the other direction. The TCP modules at each end of the connection communicate by using IP to send TCP *segments* to each other. A TCP segment consists of the IP and TCP headers and some amount of carried data, possibly none.

The data bytes (or *octets*) in the stream going each direction are conceptually numbered sequentially so that each octets has its own 32-bit *sequence number*. A TCP header contains a 32-bit sequence number field. This field contains the sequence number of the first data octet carried by the segment. If there is no data to be sent in a segment, then the sequence number in the TCP header is set by the sender to the sequence number of the first octet not yet sent. Packets with no data are sent when control information needs to be conveyed to the other end of the connection and there is no data to be sent.

The 32-bit *acknowledgment* field in the TCP header is used to indicate that all octets up to but not including the byte whose sequence number is carried in the acknowledgment field has been received and no longer needs to be retransmitted.

The 16-bit *window* field in the TCP header is used to indicate how many octets beyond the acknowledged octet the receiver is prepared to accept. This is used to implement a simple end-to-end window flow control on each direction of the data connection.

These three fields of the TCP header (the *sequence* and *acknowledgment* numbers and the *window*) along with the number of data octets carried by the TCP segment[2] are the most important elements from the packet for performance analysis. In addition to these fields from the packet, a timestamp and some means to identify which direction the packet was sent need to be included in the trace.

There are other fields from the TCP and IP headers which are important but on

[2]The number of data octets carried in a TCP segment is not explicitly carried in a separate field of the header but can be computed from other fields of the TCP and IP headers.

```
timestamp                     seq   ack   win  length
---------                     ---   ---   ---  ------
 0:59:59     HOST-A   <---- 168    52    20      0           HOST-B
 1:00:08                       52   168   100      6  ---->
 1:00:15              <---- 168    58    20      0
 1:00:17                       58   168   100     10  ---->
 1:00:19                       68   168   100     10  ---->
 1:00:30              <---- 168    68    20      0
 1:00:32                       78   168   100     10  ---->
 1:00:41              <---- 168    68    20      0
 1:00:51                       68   168   100     10  ---->
 1:01:01              <---- 168    88    20      0
 1:01:05                       88   168   100     10  ---->
 1:01:16              <---- 168    98    20      0
```

This is a synthetic trace constructed to demonstrate the basic operation of TCP. HOST-A is sending some data to HOST-B and HOST-B is sending acknowledgments and window updates back to HOST-A. The timestamps are relative to HOST-A.

Figure 2.1: A simple TCP packet trace

most functioning TCP connections there are no surprises in these fields and they do not convey much information about the performance of the TCP connection.

In Figure 2.1 is shown a synthetic TCP packet trace. This trace was contrived to demonstrate the basic operation of TCP. In this trace, data is only sent in one direction, from HOST-A to HOST-B. This can be seen by observing that the sequence number in the packets from HOST-B to HOST-A never change and that the acknowledgment number in the packets from HOST-A to HOST-B never change. The second line of the trace shows a packet seen at 1:00:08 which contains six bytes of data, all of which are beyond the point acknowledged by the packet on the first line. Seven seconds later, an acknowledgment is received acknowledging all of the data carried by the packet on the second line. The packets seen at 1:00:17 and 1:00:19 each carry 10 new data bytes. The packet on the sixth line seen at 1:00:30 acknowledges the first of the 10 byte segments (sent in the packet shown on the fourth line), but does not acknowledge the remaining ten bytes. However, since the window is extended by this packet, the sender (HOST-A) can send 10 more bytes of data and in fact does so in the packet sent at 1:00:32. At 1:00:41, another acknowledgment is received which does not acknowledge any new data, and was probably sent by HOST-B when it received the packet sent at 1:00:32 and had not received the packet sent at 1:00:19. (HOST-B saw a gap in the data, and resent the acknowledgment.) At 1:00:51 HOST-A retransmits the packet originally sent at 1:00:19 and a short while later, an acknowledgment is received for all outstanding data. Host A continues by sending another 10 bytes of data and HOST-B acknowledges this data with the last packet shown.

This trace demonstrates some of the basic features found in TCP packet traces. The packet at 1:00:15 *acknowledges* the packet seen at 1:00:08. The packet seen at 1:00:41 is a *duplicate acknowledgment* because it acknowledges the same point in the bytestream as the packet seen at 1:00:30. The packet at 1:00:51 is a *retransmission* of the packet sent 1:00:19.

2.3 Manual Packet Trace Analysis is Difficult

The first 100 packets of a packet trace captured and analyzed as an early part of this research are shown in Figures 2.2 and 2.3. The packet trace is of a 4.3BSD Unix filesystem dump to a remote tape drive which was observed to be progressing very slowly. The packet trace was collected using FTP Software's LANWatch program. The trace was transferred to a Unix workstation where it was formatted and printed.

About an hour was spent manually analyzing this packet trace trying to understand why this connection was performing poorly. This trace contains 1330 packets. It was not possible to examine in detail each packet and its relationship to the packets around it in only and hour, but it was possible to reach a few conclusions after examining in detail a few different parts of the trace. Often the machine with the

Timestamp	Source addr	port	Dest addr	port	seq	ack	win	len
12:06:12.306	18.26.0.115	1023	18.26.0.92	514	2382364546	0455093405	4096	126
12:06:12.309	18.26.0.92	514	18.26.0.115	1023	0455093405	2382364672	0	0
12:06:12.312	18.26.0.92	514	18.26.0.115	1023	0455093405	2382364672	0	0
12:06:12.424	18.26.0.115	1023	18.26.0.92	514	2382354306	0455093405	4096	1024
12:06:13.041	18.26.0.92	514	18.26.0.115	1023	0455093405	2382351234	13438	0
12:06:14.230	18.26.0.115	1023	18.26.0.92	514	2382353282	0455093405	4096	1024
12:06:17.312	18.26.0.115	1023	18.26.0.92	514	2382364672	0455093405	4096	1
12:06:17.315	18.26.0.92	514	18.26.0.115	1023	0455093405	2382364673	65535	0
12:06:17.318	18.26.0.92	514	18.26.0.115	1023	0455093405	2382364673	65535	0
12:06:17.320	18.26.0.115	1023	18.26.0.92	514	2382364673	0455093405	4096	1024
12:06:17.324	18.26.0.92	514	18.26.0.115	1023	0455093405	2382365697	64511	0
12:06:17.327	18.26.0.92	514	18.26.0.115	1023	0455093405	2382365697	64511	0
12:06:17.329	18.26.0.115	1023	18.26.0.92	514	2382365697	0455093405	4096	1024
12:06:17.332	18.26.0.115	1023	18.26.0.92	514	2382366721	0455093405	4096	1024
12:06:17.334	18.26.0.92	514	18.26.0.115	1023	0455093405	2382366721	63487	0
12:06:17.366	18.26.0.115	1023	18.26.0.92	514	2382368769	0455093405	4096	1024
12:06:17.369	18.26.0.115	1023	18.26.0.92	514	2382369793	0455093405	4096	1024
12:06:17.370	18.26.0.92	514	18.26.0.115	1023	0455093405	2382368769	61439	0
12:06:17.373	18.26.0.115	1023	18.26.0.92	514	2382370817	0455093405	4096	897
12:06:17.378	18.26.0.92	514	18.26.0.115	1023	0455093405	2382369793	0	0
12:06:17.392	18.26.0.92	514	18.26.0.115	1023	0455093405	2382369793	60415	0
12:06:17.397	18.26.0.115	1023	18.26.0.92	514	2382369793	0455093405	4096	1024
12:06:17.401	18.26.0.92	514	18.26.0.115	1023	0455093405	2382370817	59391	0
12:06:17.404	18.26.0.92	514	18.26.0.115	1023	0455093405	2382370817	59391	0
12:06:17.407	18.26.0.115	1023	18.26.0.92	514	2382370817	0455093405	4096	897
12:06:17.412	18.26.0.92	514	18.26.0.115	1023	0455093405	2382371714	58494	0
12:06:17.417	18.26.0.92	514	18.26.0.115	1023	0455093405	2382371714	58494	0
12:06:17.500	18.26.0.92	514	18.26.0.115	1023	0455093405	2382371714	58494	7
12:06:17.511	18.26.0.115	1023	18.26.0.92	514	2382371714	0455093412	4096	7
12:06:17.514	18.26.0.92	514	18.26.0.115	1023	0455093412	2382371721	58487	0
12:06:17.516	18.26.0.92	514	18.26.0.115	1023	0455093412	2382371721	58487	0
12:06:17.519	18.26.0.92	514	18.26.0.115	1023	0455093412	2382371721	58487	0
12:06:17.521	18.26.0.92	514	18.26.0.115	1023	0455093412	2382371721	58487	0
12:06:17.523	18.26.0.92	514	18.26.0.115	1023	0455093412	2382371721	58487	0
12:06:18.157	18.26.0.92	514	18.26.0.115	1023	0455093412	2382404489	25719	7
12:06:18.168	18.26.0.115	1023	18.26.0.92	514	2382404489	0455093419	4096	7
12:06:18.171	18.26.0.92	514	18.26.0.115	1023	0455093419	2382404496	25712	0
12:06:18.175	18.26.0.92	514	18.26.0.115	1023	0455093419	2382404496	25712	0
12:06:18.177	18.26.0.92	514	18.26.0.115	1023	0455093419	2382404496	25712	0
12:06:18.180	18.26.0.92	514	18.26.0.115	1023	0455093419	2382404496	25712	0
12:06:18.182	18.26.0.92	514	18.26.0.115	1023	0455093419	2382404496	25712	0
12:06:18.185	18.26.0.92	514	18.26.0.115	1023	0455093419	2382404496	25712	0
12:06:18.187	18.26.0.92	514	18.26.0.115	1023	0455093419	2382404496	25712	0
12:06:18.189	18.26.0.92	514	18.26.0.115	1023	0455093419	2382404496	25712	0
12:06:18.202	18.26.0.115	1023	18.26.0.92	514	2382404496	0455093419	4096	1024
12:06:18.204	18.26.0.115	1023	18.26.0.92	514	2382405520	0455093419	4096	1024
12:06:18.207	18.26.0.115	1023	18.26.0.92	514	2382406544	0455093419	4096	1024
12:06:18.209	18.26.0.115	1023	18.26.0.92	514	2382407568	0455093419	4096	1024
12:06:18.212	18.26.0.115	1023	18.26.0.92	514	2382408592	0455093419	4096	1024
12:06:18.214	18.26.0.115	1023	18.26.0.92	514	2382409616	0455093419	4096	1024

This is from a real packet trace collected from a monitor on the network. The first 50 packets are shown here. The next 50 are shown in Figure 2.3.

Figure 2.2: A real TCP packet trace

Timestamp	Source addr	port	Dest addr	port	seq	ack	win	len
12:06:18.216	18.26.0.92	514	18.26.0.115	1023	0455093419	2382407568	22640	0
12:06:18.220	18.26.0.115	1023	18.26.0.92	514	2382411664	0455093419	4096	1024
12:06:18.223	18.26.0.115	1023	18.26.0.92	514	2382412688	0455093419	4096	1024
12:06:18.225	18.26.0.115	1023	18.26.0.92	514	2382413712	0455093419	4096	1024
12:06:18.229	18.26.0.115	1023	18.26.0.92	514	2382414736	0455093419	4096	1024
12:06:18.231	18.26.0.115	1023	18.26.0.92	514	2382415760	0455093419	4096	1024
12:06:18.233	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.237	18.26.0.115	1023	18.26.0.92	514	2382417808	0455093419	4096	1024
12:06:18.239	18.26.0.115	1023	18.26.0.92	514	2382418832	0455093419	4096	1024
12:06:18.242	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.246	18.26.0.115	1023	18.26.0.92	514	2382420880	0455093419	4096	1024
12:06:18.249	18.26.0.115	1023	18.26.0.92	514	2382421904	0455093419	4096	1024
12:06:18.252	18.26.0.115	1023	18.26.0.92	514	2382422928	0455093419	4096	1024
12:06:18.255	18.26.0.115	1023	18.26.0.92	514	2382423952	0455093419	4096	1024
12:06:18.257	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.261	18.26.0.115	1023	18.26.0.92	514	2382426000	0455093419	4096	1024
12:06:18.264	18.26.0.115	1023	18.26.0.92	514	2382427024	0455093419	4096	1024
12:06:18.266	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.270	18.26.0.115	1023	18.26.0.92	514	2382429072	0455093419	4096	1024
12:06:18.273	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.276	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.279	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.282	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.285	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.288	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.291	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.294	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	0	0
12:06:18.296	18.26.0.92	514	18.26.0.115	1023	0455093419	2382408592	21616	0
12:06:18.301	18.26.0.115	1023	18.26.0.92	514	2382408592	0455093419	4096	1024
12:06:18.305	18.26.0.92	514	18.26.0.115	1023	0455093419	2382409616	20592	0
12:06:18.308	18.26.0.92	514	18.26.0.115	1023	0455093419	2382409616	20592	0
12:06:18.310	18.26.0.115	1023	18.26.0.92	514	2382409616	0455093419	4096	1024
12:06:18.313	18.26.0.115	1023	18.26.0.92	514	2382410640	0455093419	4096	1024
12:06:18.315	18.26.0.92	514	18.26.0.115	1023	0455093419	2382410640	19568	0
12:06:18.318	18.26.0.92	514	18.26.0.115	1023	0455093412	2382375817	54391	0
12:06:18.321	18.26.0.115	1023	18.26.0.92	514	2382381961	0455093412	4096	1024
12:06:18.326	18.26.0.92	514	18.26.0.115	1023	0455093419	2382412688	17520	0
12:06:18.328	18.26.0.115	1023	18.26.0.92	514	2382415760	0455093419	4096	1024
12:06:18.334	18.26.0.115	1023	18.26.0.92	514	2382417808	0455093419	4096	1024
12:06:18.336	18.26.0.115	1023	18.26.0.92	514	2382418832	0455093419	4096	1024
12:06:18.339	18.26.0.115	1023	18.26.0.92	514	2382419856	0455093419	4096	1024
12:06:18.341	18.26.0.115	1023	18.26.0.92	514	2382420880	0455093419	4096	1024
12:06:18.343	18.26.0.92	514	18.26.0.115	1023	0455093419	2382413712	0	0
12:06:18.351	18.26.0.92	514	18.26.0.115	1023	0455093412	2382380937	0	0
12:06:18.354	18.26.0.92	514	18.26.0.115	1023	0455093412	2382380937	0	0
12:06:18.366	18.26.0.92	514	18.26.0.115	1023	0455093419	2382413712	0	0
12:06:18.368	18.26.0.115	1023	18.26.0.92	514	2382427024	0455093419	4096	1024
12:06:18.370	18.26.0.115	1023	18.26.0.92	514	2382428048	0455093419	4096	1024
12:06:18.370	18.26.0.115	1023	18.26.0.92	514	2382397321	0455093412	4096	1024
12:06:18.373	18.26.0.115	1023	18.26.0.92	514	2382429072	0455093419	4096	1024

This is the continuation of the trace shown in Figure 2.2.

Figure 2.3: More of a real TCP packet trace

tape drive was offering a window of zero. Furthermore, the machine with the tape drive was often taking back previously offered window.

For example, at 12:06:17.370, the end of the window (determined by adding together the acknowledgment number and the window field of the packet) was at $2382368769 + 61439 = 2382430208$, but eight milliseconds later the window field of the packet is zero, and the acknowledgment number is only at 2382369793 which is 60415 bytes short of the previously offered end of window.

These observations led to the conclusion that the machine that was receiving the data to be dumped was not behaving ideally and that the problem seemed to lie in the TCP implementation of that host.

This packet trace is interesting because it was the first real problematic connection traced and analyzed as part of this project and served to motivate the development of some better way of presenting the trace for analysis. It is not an unusually long packet trace. Many other collected traces of TCP connections were longer, some as much as twice as long as this one. However, a trace of this length is very cumbersome to analyze manually. This trace demonstrated the need for some means of dealing with the large amount of information presented in a packet trace.

Doing the analysis of this trace demonstrated that such analysis can be useful even if it is cumbersome and tedious. Useful information was found in the trace and some conclusions were reached about what might be the cause of the poor performance of the system using the TCP connection.

One of the most difficult parts of manual analysis of TCP packet traces arises from the need to understand how packets relate to each other in time and in sequence or acknowledgment numbers. Extracting these relationships from the trace probably took most of the time spent analyzing this trace.

```
timestamp                    seq   ack   win  length
---------                    ---   ---   ---  ------
 0:59:59     HOST-A   <----        52    20                HOST-B
 1:00:08                      52               6  ---->
 1:00:15              <----        58    20
 1:00:17                      58              10  ---->
 1:00:19                      68              10  ---->
 1:00:30              <----        68    20
 1:00:32                      78              10  ---->
 1:00:41              <----        68    20
 1:00:51                      68              10  ---->
 1:01:01              <----        88    20
 1:01:05                      88              10  ---->
 1:01:16              <----        98    20
```

This is the same as Figure 2.1 but with the fields which were relevant only to the data sent from HOST-B to HOST-A removed.

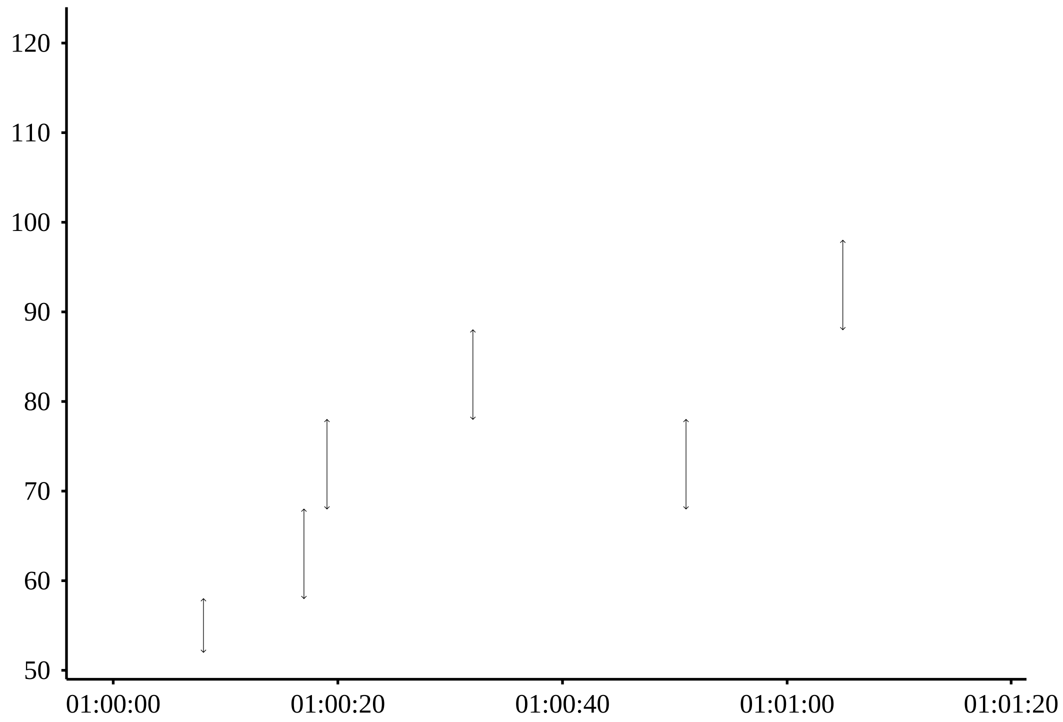
Figure 2.4: Trace information for half of the duplex TCP connection

2.4 Displaying the Trace Graphically

This section will show how to present the packet trace in a graphical format so that these relationships will be evident as simple spatial relationships. The method of displaying the trace presented in this section increases the rate at which TCP packet traces can be analyzed by humans by a factor of 100 to 1000.[3]

The key idea to this graphical form is to display the packet trace as a *time-sequence plot* where the horizontal axis is indexed by time and the vertical axis is indexed by sequence number. In [9], Jacobson first applied this idea to TCP connections when he used it to show the progress of a TCP connection by plotting the sequence number field of packets versus the time each packet was sent. The graphical method used to display packets in this thesis expands on this idea and displays all of the information

[3]An hour's worth of manual analysis may take anywhere from 5 to 30 seconds when using the tools presented here. Someone experienced at viewing the plots can often grasp what transpired at first glance, much more quickly than the story could be expressed verbally.

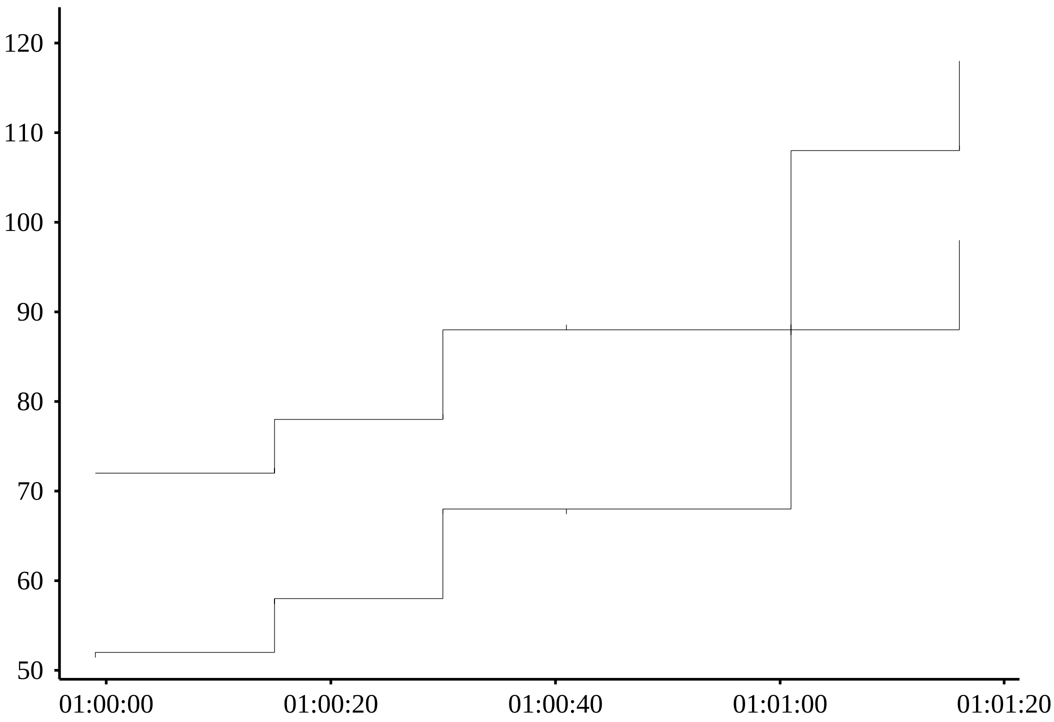


This time-sequence plot shows the six packets sent from HOST-A to HOST-B in Figure 2.4. The horizontal axis is time (in hours, minutes and seconds) and the vertical axis is the TCP sequence number.

Figure 2.5: A time-sequence plot

contained in a packet trace of the sort shown in the previous section graphically so that the analyst need not refer back to a printed packet trace for any information.

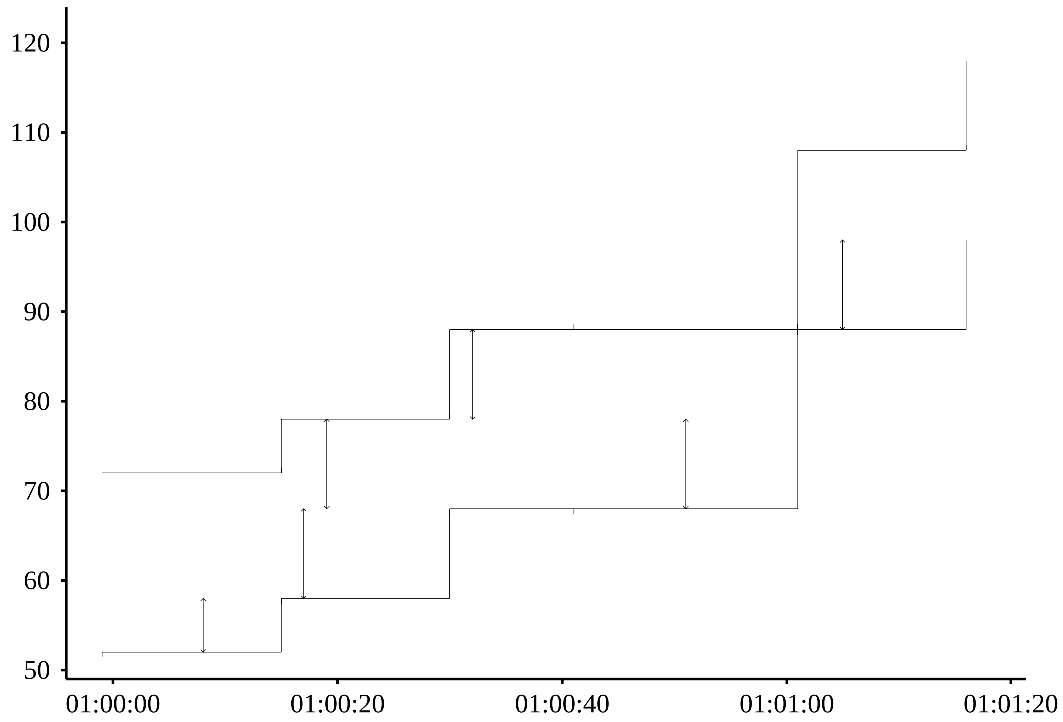
The packet trace in Figure 2.1 will be used to illustrate the use of a time-sequence plot to display a packet trace. A TCP connection provides a byte stream service in each direction. A single time-sequence plot displays information about the data flow of a single direction only. To show exactly what data is transferred to the plot, Figure 2.4 contains a copy of trace shown in Figure 2.1 but including only the fields of the connection relevant to understanding the process of carrying data from HOST-A to HOST-B. The relevant fields are the sequence and data-length fields of packets sent by HOST-A (the *sender* of the data) and information from the acknowledgment and window fields of the packets sent by HOST-B (the *receiver* of the data).



This time-sequence plot shows the ack lines and window lines. The axis are the same as in Figure 2.5.

Figure 2.6: Time-sequence space plot of information in returning packets

The packets sent by the sender are plotted by placing a vertical line segment in time-sequence space starting at the sequence number contained in the sequence field of the packet and extending upwards for the length of the packet. To make these segments easier to recognize and to make it possible to see zero-length segments, an arrow is affixed to each end of the line segment. Zero length segments appear only as two arrows facing each other, which looks like letter X since the heads of the arrow are placed in the same location. Figure 2.5 contains a time-sequence plot with just this information shown. Notice that the six vertical line segments correspond to the six packets sent from HOST-A to HOST-B in the packet trace. The location relative to the horizontal access corresponds to the timestamp on the packet in the trace. Notice that all three relevant numbers from the trace (the timestamp, the sequence



This is the detailed time-sequence plot for the data sent from HOST-A to HOST-B of the trace shown in Figure 2.1. The horizontal axis is time (in hours, minutes and seconds) and the vertical axis is the TCP sequence number.

Figure 2.7: The complete time-sequence plot

number, and the length) could be recovered from this diagram.

In Figure 2.6 the packets sent by the receiver are plotted by extracting the acknowledgment and window fields, computing an end-of-window number by adding these two fields together, and plotting points at the location of the (time-of-packet, acknowledgment) and (time-of-packet, end-of-window) in time-sequence space. The space between these points can be thought of as the window and to make this clearer, over-and-up stair steps are plotted from the each of the points plotted by the previous inbound packet to the corresponding new point. The stair-step produced by plotting the acknowledgment fields is called the *ack line* and the region on the plot between the ack line and the line at the end of the window (the *window line*) can be thought of as the window. In order to make visible inbound packets which contain the same acknowledgment or window numbers as the previous packet, down tick marks are placed on the ack line and up tick marks window lines. An example of this occurs at 1:00:41.

When the plots of both receiver-sent and sender-sent packets are combined, the result is the complete time-sequence plot. This result is shown in Figure 2.7. Notice how the packets sent by the sender naturally lie in the window and how easy it is to tell when the first acknowledgment which covers a packet sent by the sender is received. All of the time and sequence number relationships are easily extracted visually from this plot without the need to examine the original trace or handle numbers.

Timestamps are Relative

Messages take time to travel through the network. Because of this, traces of a given trace collected from different points in the network would have different timing relationships. Most often the analyst is interested in understanding the retransmission strategies of the sender of data so packet traces are usually taken relative to the sender of data, often directly from an instrumented TCP module running in the sender.

The packet traces collected for this project were taken from the Ethernet, independent of either end of the connection. The timestamps in the packet traces do not necessarily match the timing at either end of the TCP connection. Understanding this can be important when analyzing a packet trace of a connection between two hosts. There are four main configurations. One is when both end points of the TCP connection are on the same ethernet as the monitor. In this case, the round trip time between the two hosts is very short and timestamps are best thought of as being relative to a midpoint between the two machines. A second configuration is when the monitor is located on the same ethernet as the sender of data and the recipient of the data is located at a distance in the network. In this case, the timestamps on the packets captured by the monitor can be thought of as being the same as the sender sees. These first two cases were the configuration for almost all traces examined as part of this research. Since much of the complexity in TCP is deciding when to transmit, these two configurations are probably the most useful because the analyst's view will be the same as that of the transmitter. A third configuration is where the sender of data is distant in the network and the monitor is near the recipient. In this case the packets are seen effectively when the receiver sees them. This can produce strange results. An example of this is shown in Figure 3.15. The final configuration would be where the monitor is on a transit network and is not near either of the end points of the connection.

If traces collected away from the sender are to be analyzed, it needs to be understood that the observed timing of packets is partly a result of the network as well as the host that sent them. This applies as well to analyzing the acknowledgments and window updates when the traces are collected away from the receiver.

Time-Sequence Plots Preserve Information Content

Time-sequence plots of TCP packet traces have a nice information preserving property. They preserve the information content of the original trace. Figure 2.7 contains all of the information in Figure 2.4. This is key to eliminating the need for the human analyst to examine the printed trace or any other presentation of data about the trace.

There are other fields in the TCP packet which might be of interest to the analyst. The fields not included in the packet traces and time-sequence plots in this chapter which might be of interest are the the six TCP flag bits (URG, ACK, PSH, RST, SYN, and FIN), the urgent pointer, and the options. Also, the checksum field was ignored and could not have been verified without the entire TCP segment. The entire TCP segment was not usually available because the system used to capture the packets only saved the headers. The SYN and FIN bits do occupy sequence number space, and are included in the computation of the length fields of the packets shown, but no explicit indication of these bits was included.

The information in these other fields has not seemed necessary for diagnosing performance problems. No effort has been made to include this information in the traces or diagrams in this thesis. If it were necessary to include this information for general debugging of the TCP protocol, it should be possible to annotate the time-sequence plot with this information. For example, text could be placed directly on the diagram in the appropriate places to indicate the settings of the flag bits when they do not have the usual values.

Chapter 3

Observation of TCP

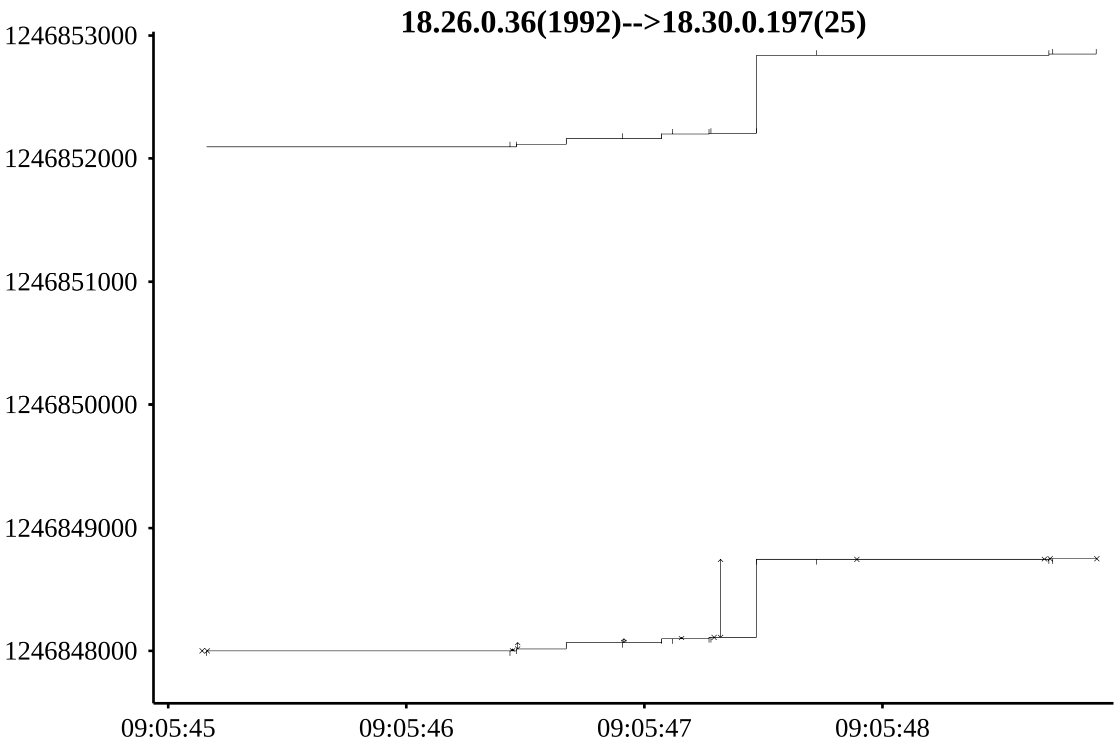
By presenting a collection of examples, this chapter will demonstrate that the graphical method presented in the previous chapter enables the analysis of TCP packet traces.

Over a hundred TCP connections collected from an operational Ethernet in LCS were examined in detail using time-sequence plots. Only a few will be presented here. In the first part of this chapter, time-sequence plots which represent the most typical TCP connections will be presented. Second, the performance of two TCP connections carrying bulk data are examined in detail. Then at the end of this chapter, some of the most interesting TCP connections observed using time-sequence plots will be presented.

3.1 Typical TCP Connections

Figures 3.1 through 3.6 are examples time-sequence plots of typical TCP connections collected from the operational network. Each is shown here to demonstrate the operation of some of the most common uses of TCP connections: the transfer of computer mail, remote terminal connection, and bulk data transfer.

Figures 3.1 and 3.2 show the behavior of both halves of a TCP connection trans-

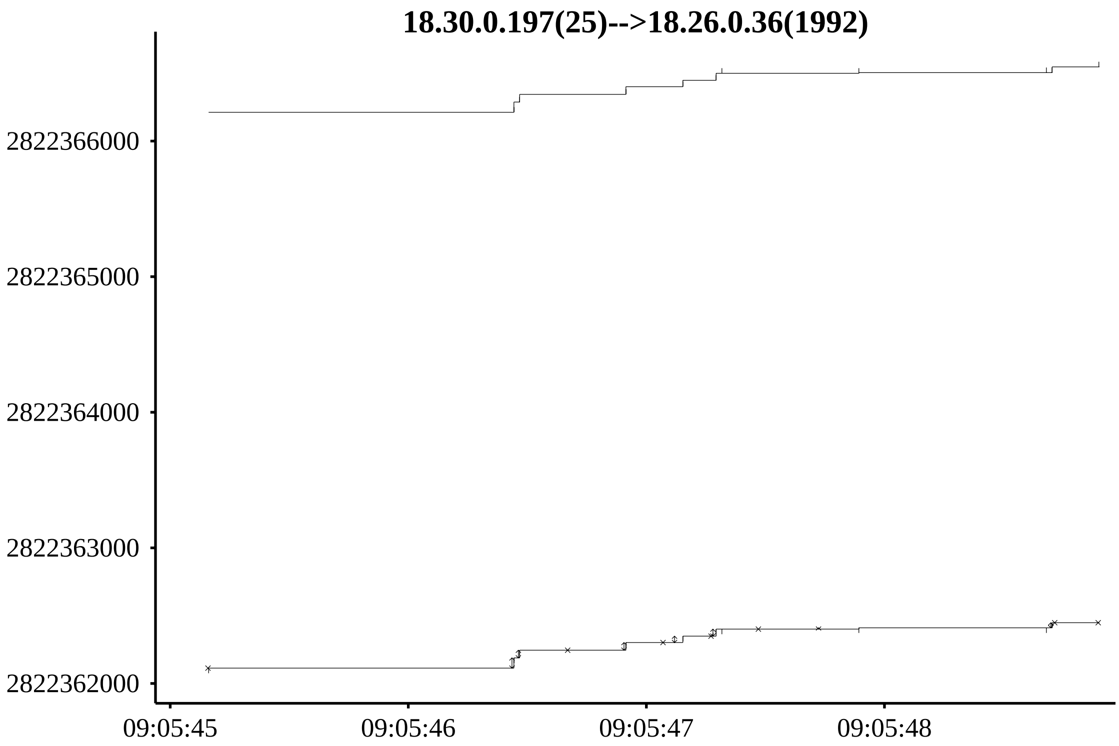


This is a time-sequence plot of a TCP connection to an SMTP port. This plot shows how the data was sent to the SMTP server. The entire connection from SYN to FIN is shown here. Figure 3.2 shows the other half of this connection.

Figure 3.1: SMTP TCP connection

ferring a computer mail message using the SMTP protocol. The SMTP protocol is described in [15]. The SMTP protocol first exchanges some short messages identifying the hosts involved and the intended recipients of the message. After these have been sent, verified, and acknowledge by the SMTP server the message is transferred to the SMTP server. Once the entire message has been received and safely stored by the SMTP server, it sends a reply acknowledging that the message has been successfully transferred.

Figure 3.1 is a time-sequence plot of the sender to SMTP server side of the connection. The large data packet soon after 09:05:47 is the transfer of the actual body of the message. The much smaller packets before and after are the SMTP commands being



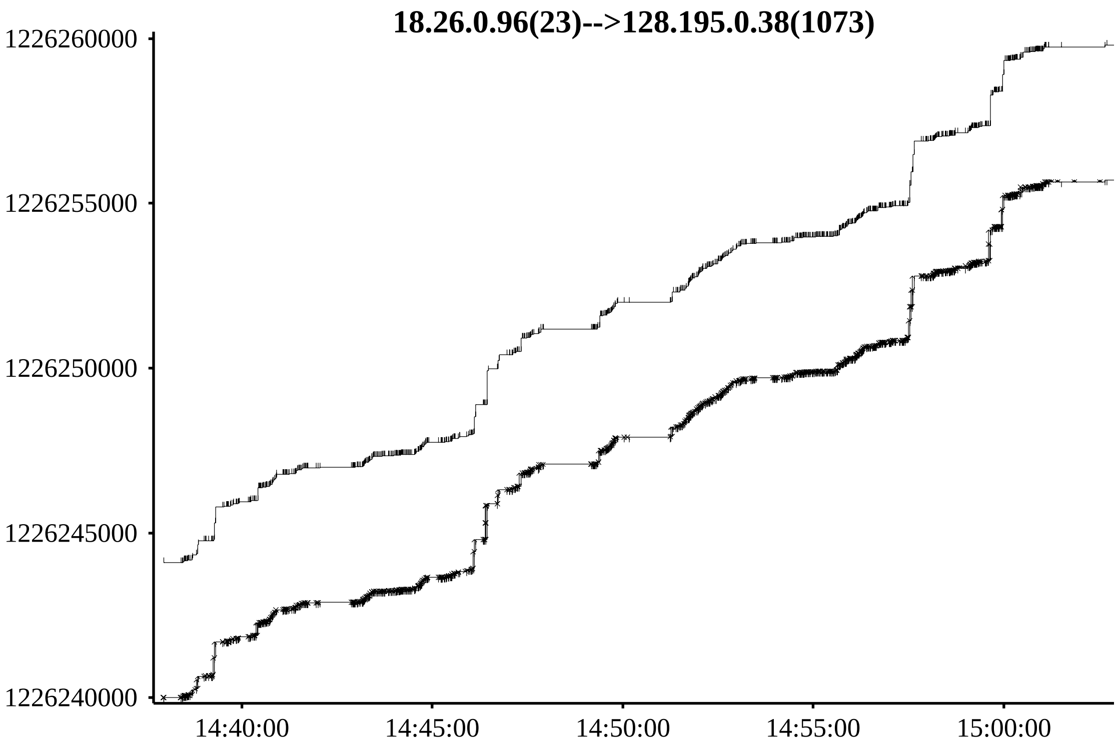
This is a time-sequence plot of the other half of the SMTP TCP connection shown in Figure 3.1.

Figure 3.2: Other half of SMTP TCP connection

exchanged. Figure 3.2 is a time-sequence plot of the other half of the same SMTP connection. Only short server responses are sent on this half of the connection.

The body of the message transferred was apparently less than 1000 bytes in length and fit into a single data packet. Longer messages would be broken into multiple packets. The transfer of an extremely long message might begin to look like one of the bulk data transfer connections presented later in this chapter.

The next two figures, Figure 3.3 and Figure 3.4, are both examples of interactive traffic. Figure 3.3 is a plot of a TELNET connection, and Figure 3.4 is a plot of an X11 window system connection. Both plots are of the side of the connection



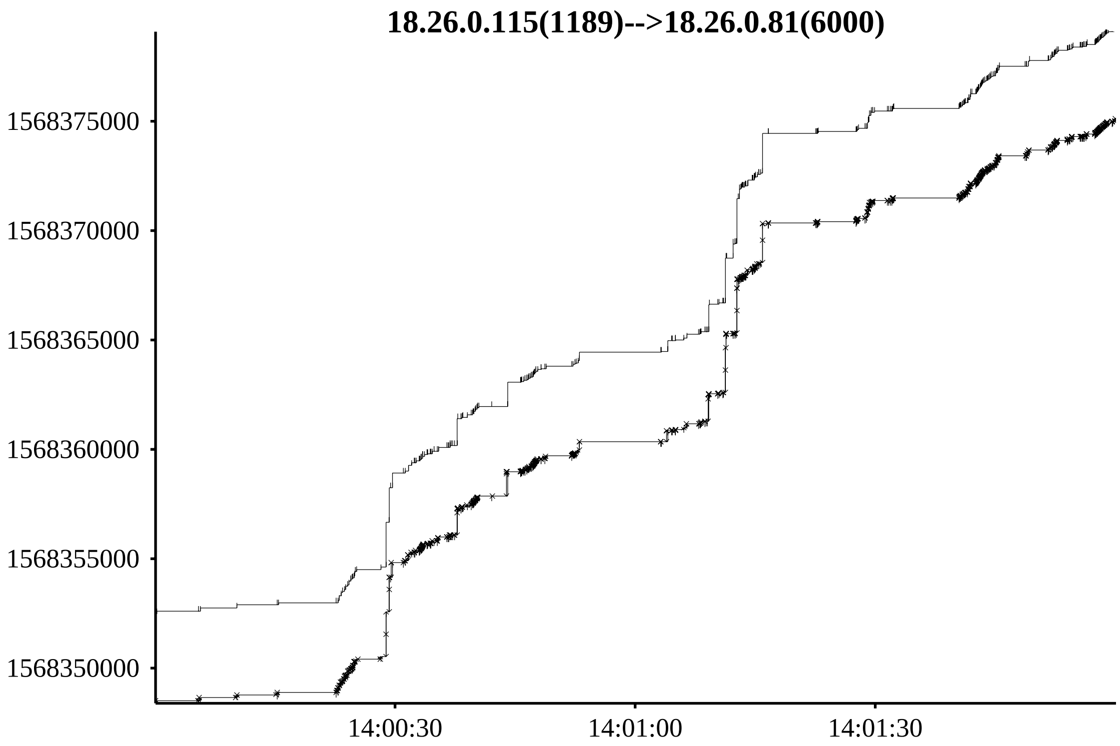
This is a time-sequence plot of the host-to-user half of a typical TELNET connection from a user on a distant host. Over twenty minutes of the connection is shown here. Most of the TCP data segments are short and probably correspond to remote echoes of typed characters. At this scale, these short segments are all blurred together on the ack line. Occasionally larger bursts of traffic can be seen. These are probably output from a program.

Figure 3.3: Telnet connection

returning output to the user.[1] Idle periods, periods of typing where the packets are very short and somewhat frequent, and short bursts of output where larger packets are sent can be seen in both of these figures. Alternating between these three modes is characteristic of interactive traffic.

The four TCP connections presented thus far are typical of most TCP connections

[1]In the case of TELNET, this is from the server to the client, but in the case of the X11 window system this is from the client to the server since the data flowing *to* the X11 window system server contains the output to be displayed to the user.



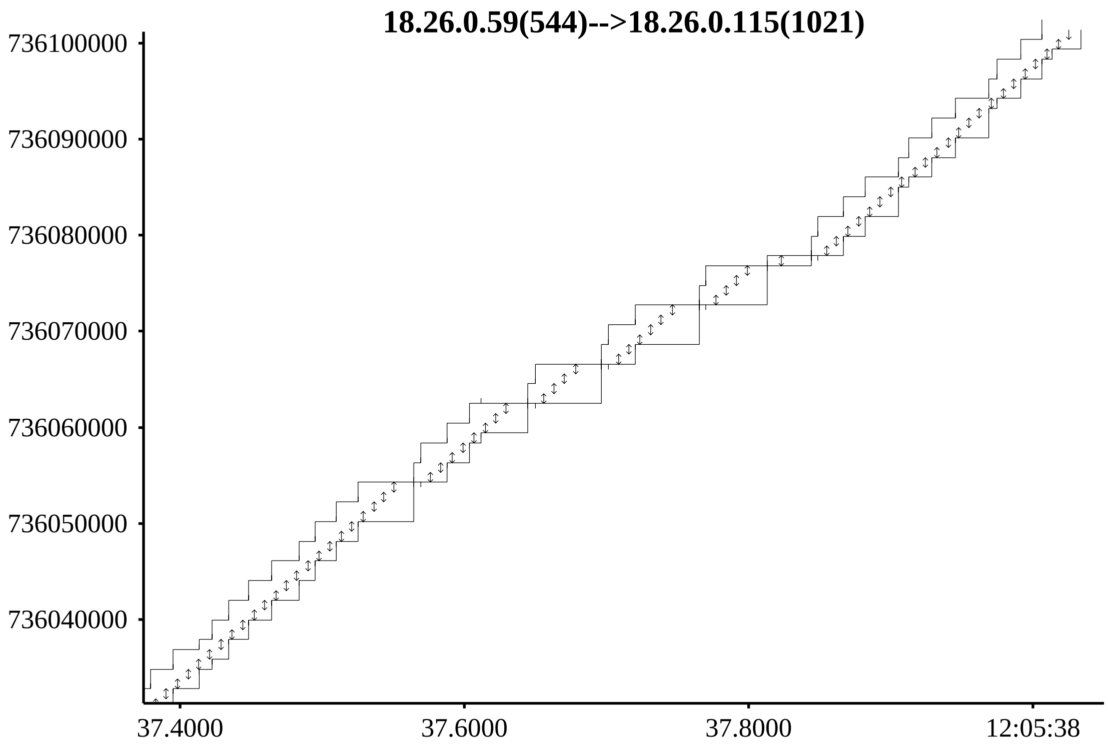
This shows the client to server (program to user) half of an X11 window system connection. This appears very similar to the trace of a telnet connection shown in Figure 3.3 which suggests that this X11 client was some sort of interactive program.

Figure 3.4: X11 window system connection

which are not used for bulk data transfer. They all share a few characteristics: they consist mostly of small packets, retransmissions are very rare, and the TCP window is rarely filled. No performance problems are evident in these TCP connections. Since the performance demands on TCP are low on these connections, it is not surprising that no performance problems are evident.

TCP connections carrying bulk data are not as common, but are much more interesting to observe. The next two figures are examples of TCP connections carrying bulk data.

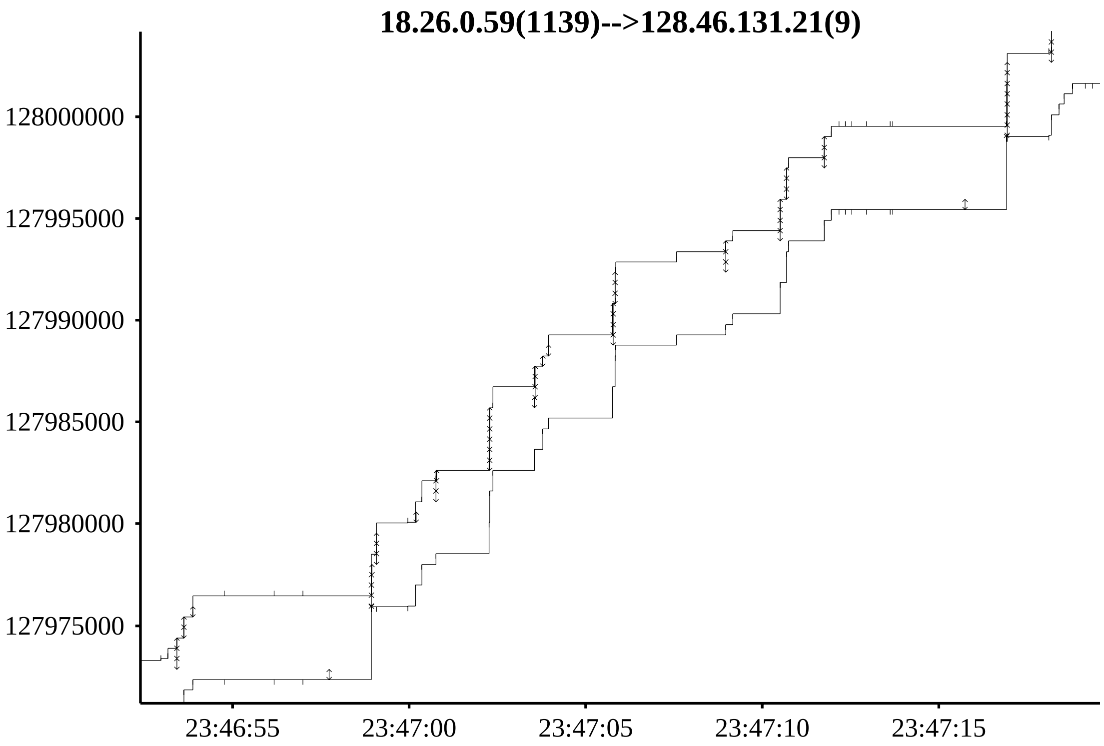
Figure 3.5 shows the behavior of a part of a TCP connection carrying data for a



This time-sequence plot shows the behavior of a bulk data transfer to a host on the same local area network.

Figure 3.5: Local bulk data transfer

file transfer program to another host on the same local area network (an Ethernet). Each of the packets is roughly 1000 bytes and over 60 packets can be seen in less than 0.6 second. This accounts for less than 10% of the capacity of an Ethernet, so the performance of the Ethernet does not appear to limit throughput. There were no retransmissions in this portion of the connection, though there were a few times when the transmitter apparently could not send because the TCP window was full. Almost all returning packets updated both the acknowledgment and the end of window, though a few returning packets which only updated one or the other can be seen. It appears that the progress of this connection is usually limited by the ability of the transmitter to send the packets, though sometimes it is held up by the receiver



This plot shows the behavior of a TCP connection carrying bulk data to a distant host.

Figure 3.6: Bulk data transfer to a distant host

not keeping the window open.

Figure 3.6 is an example of a TCP connection carrying bulk data to a distant host across a slower long-haul network. Here the window is often filled completely, and the progress of the connection is limited by the end-to-end flow control implemented by the TCP window mechanism. Retransmissions of presumably lost packets can also be seen in this connection. Prior to both retransmissions, duplicate acknowledgments can be seen. These were probably sent when packets beyond the missing packet were received.

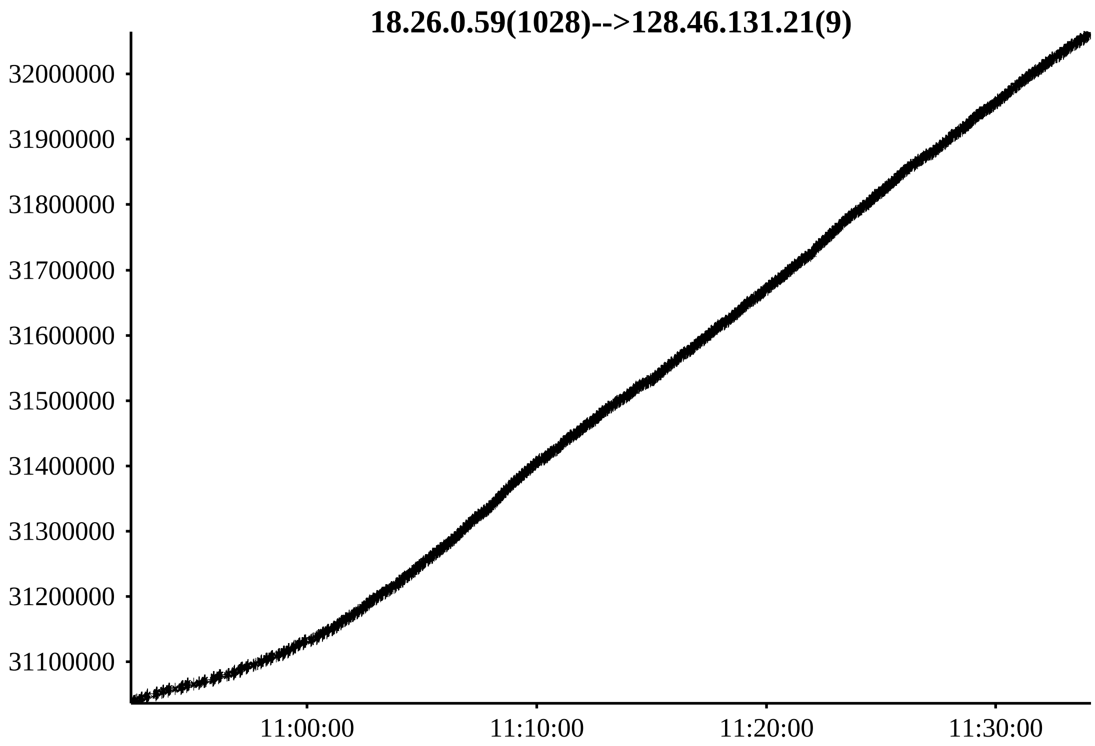
In Figures 3.1 through 3.4, the throughputs of these TCP connections were limited by the applications using the connections. The TCP connection shown in Figure 3.5

was limited by the host's ability to send the packets quickly on the Ethernet and occasionally by the receiving host's ability to keep the window open. The connection shown in Figure 3.6 is fundamentally different because the performance of the network plays a significant role in determining the performance of the connection.

3.2 Two TCP Connections

Long distance networks often use lower bandwidth trunks, have longer delays, and are shared among more users than local area networks. They are more heavily used (in proportion to capacity) than local area networks. The longer delay and congestion (which causes packets to be either dropped or delayed even further) caused by heavy use can cause the performance of a TCP connection to be sensitive to the algorithms used to control the transmission of data.

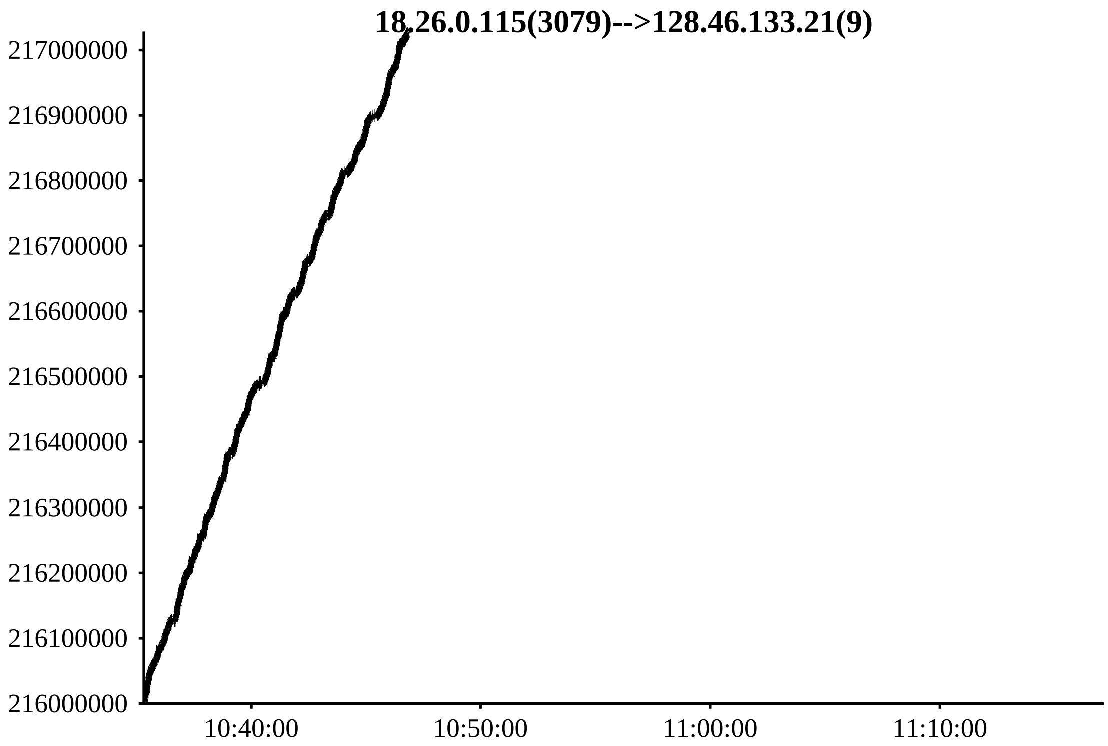
Several revisions have been made to the Berkeley Unix TCP retransmission strategy. The 4.2BSD TCP would retransmit all segments held when the retransmit expired. This burst of retransmissions on each expiration of the retransmit timer was undesirable for a few reasons. If only the first unacknowledged segment is missing, this would lead to many unnecessary retransmissions. If the other host is currently unreachable, then sending more than one segment on each expiration of the retransmit timer is pointless and only contributes to congestion. In the 4.3BSD TCP, this algorithm was improved to send only the first unacknowledged segment when the retransmit timer expired. Once an acknowledgment is received which covers previously unacknowledged data, any segments on the retransmit queue which have not yet been acknowledged are retransmitted [11]. Further improvements to the 4.3BSD TCP retransmission algorithm, including the slow-start algorithm, were later made by Jacobson and Karels [11]. In the slow-start algorithm, further acknowledgments after the retransmission of an initial single segment are used to clock out an exponentially growing number of segments in each round trip time interval.



This is a time-sequence plot of an entire TCP connection which transferred 1,024,000 bytes, and lasted over 40 minutes.

Figure 3.7:

Packet traces of two different TCP connections will be used here to demonstrate how the behavior of the TCP transmitter can seriously affect performance over long paths in the network. Both connections were the result of running the same test program on two different machines at MIT with essentially identical hardware configuration. One machine, at IP address 18.26.0.59, was running Ultrix 2.2 (whose TCP is probably based on the standard 4.3 BSD TCP) and the other, at IP address 18.26.0.115, was running 4.3 BSD Unix with Jacobson's TCP improvements which are described in [9]. The test program opens a TCP connection to a remote sink port and calls `write()` on the socket with a 1,024,000 byte buffer. Both connections were



This is a time-sequence plot of a TCP connection from an improved TCP implementation transferring the same amount of data across the same network as the connection shown in Figure 3.7. The scale is the same as the scale in Figure 3.7 so the overall performance can be easily compared.

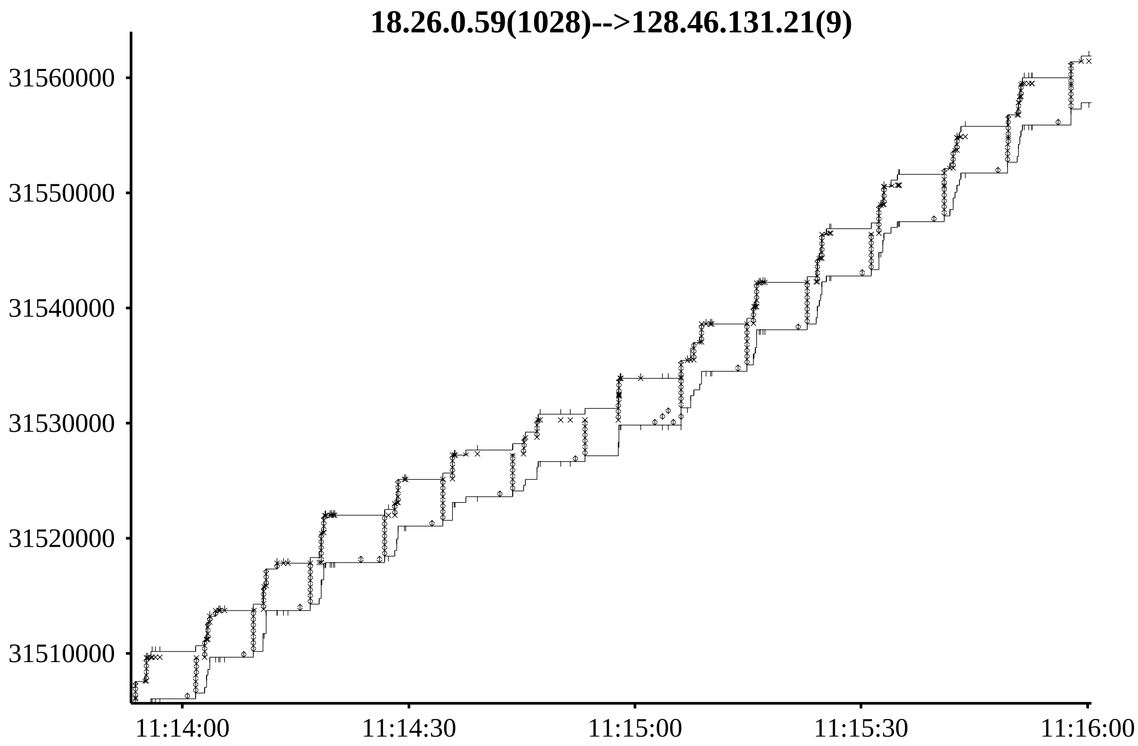
Figure 3.8:

to the sink port on the same remote host, a machine at Purdue University.[2]

The traces were collected from the Ethernet at MIT which is directly connected to the two source machines. The timing in the packet traces are essentially the same as seen by the sender of the data.

Figure 3.7 and Figure 3.8 are time-sequence plots of traces of the entire connections. The scale in Figure 3.8 matches the scale in Figure 3.7. Not much detail can

[2]Unfortunately, the hostname did not resolve to identical IP addresses on the two machines so different IP addresses for the same target machine were used. The routes used for both connections left MIT via a connection to the ARPANET. The routes used by the two connections are believed to be essentially identical.



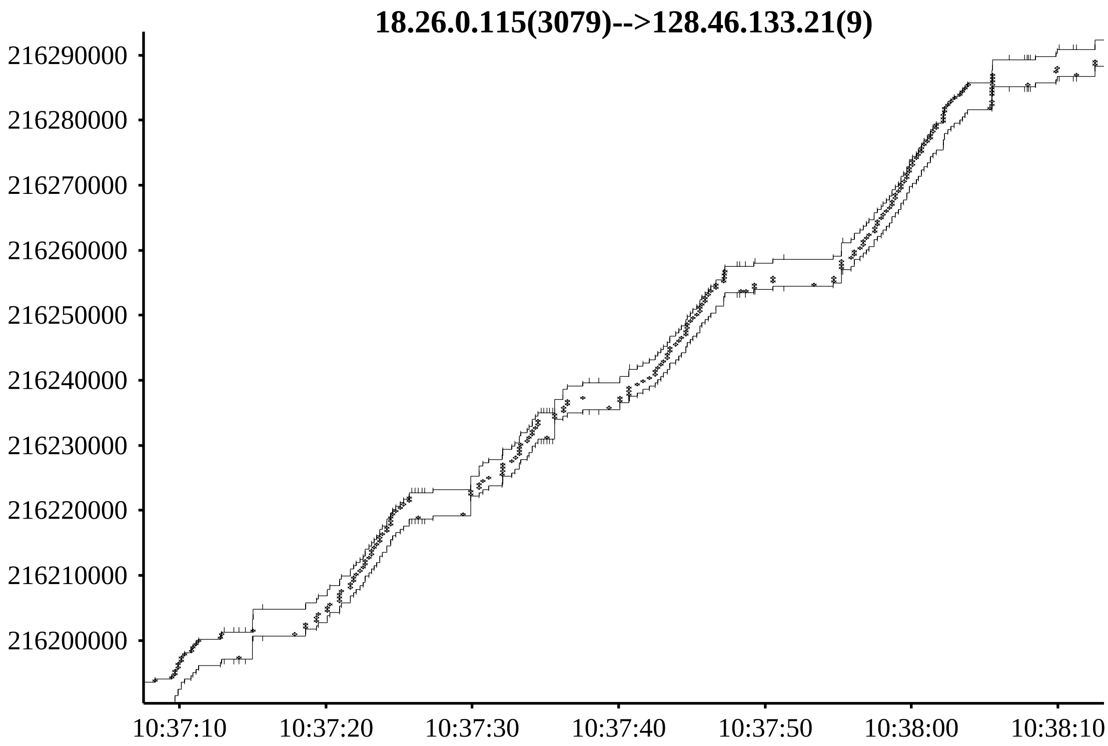
This is a enlarged view of a portion of the trace shown in Figure 3.7. (Examining a further enlarged view in Figure 3.11 before this figure might aid in deciphering the details of this figure.)

Figure 3.9:

be seen at this scale, but the overall performance (from the users perspective) of the improved TCP is obviously much better since it transferred the same amount of data in about one fourth the time.

Figure 3.9 and Figure 3.10 are enlarged views of the same two connections.

Poor performance is evident in Figure 3.9. The TCP is sending most segments of data twice. There are pauses of about 10 seconds between each salvo which are probably caused by waiting for the retransmit timer. About one window's worth of data is successfully acknowledged on each salvo, but a segment near the end of the initial window is lost every time, requiring the retransmission. When the first retransmit occurs, only one packet is sent, but once it is acknowledged, another salvo



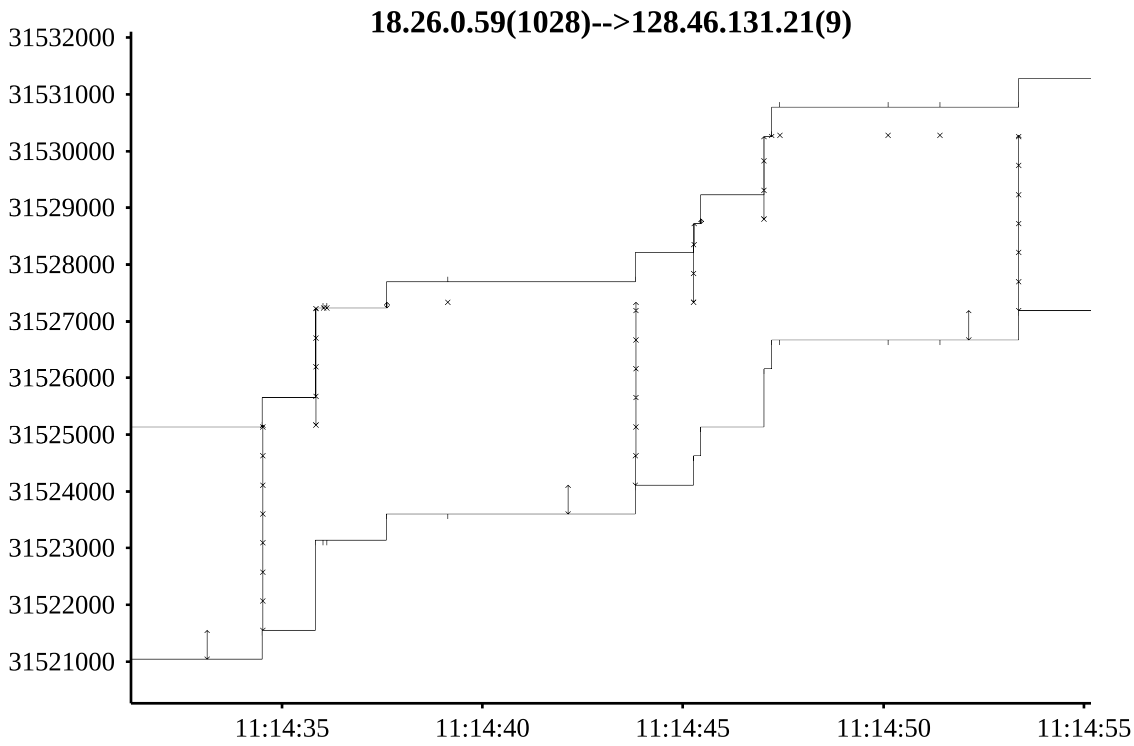
This is a enlarged view of a portion of the trace shown in Figure 3.8. (Examining a further enlarged view in Figure 3.12 before this figure might aid in deciphering the details of this figure.)

Figure 3.10:

is launched. This is probably the algorithm used in the 4.3BSD Unix [11] where once a retransmit timer has gone off, the TCP refrains from dumping the entire retransmit queue until it receives an acknowledgment covering previously unacknowledged data.

This behavior is remarkably periodic, suggesting that there is some process which causes the network always to drop a packet which is usually at about the same place in the salvo. This, and the success of the other TCP connection, suggests that this TCP implementation is the cause of its own troubles.

In the closer view of the operation of the improved TCP in Figure 3.10 the basic behavior of the combined slow-start and congestion-avoidance algorithm [9] is evident. Whenever a packet needs to be retransmitted, the improved TCP refrains from



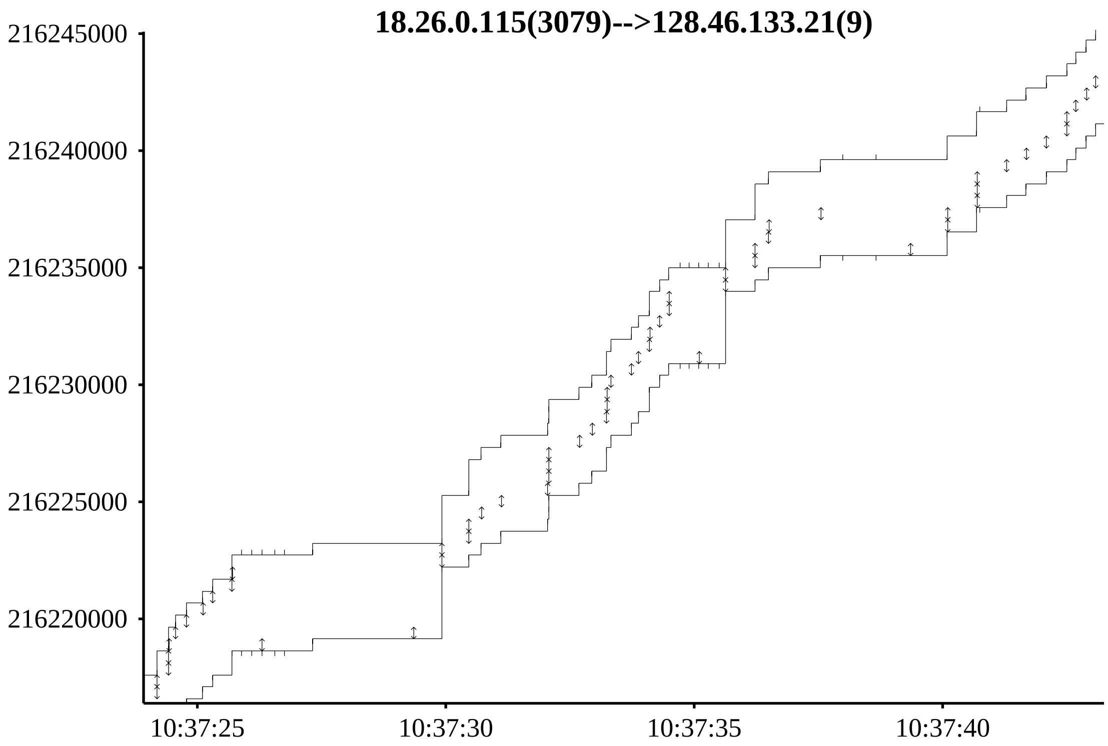
This is an enlarged view of a portion of the trace shown in Figure 3.9.

Figure 3.11:

sending bursts of packets. Instead, it sends just one packet, and allows one more outstanding packet per round-trip time depending on whether it is in the slow-start or congestion-avoidance phase. The upward curl at the start of each episode is characteristic of the combined slow-start and congestion-avoidance algorithms used in the improved TCP. (These episode boundaries can even be seen as bumps in Figure 3.8.)

The overall performance difference between these two connections can also be seen at this scale if the difference in scale between the two different time-sequence plots is noted.

Figure 3.11 and Figure 3.12 are even closer views from the previous two figures. At this level of detail, packet by packet analysis (as was done for Figure 2.1) is possible, yet these two plots each contain about a factor of eight more information than the



This is an enlarged view of a portion of the trace shown in Figure 3.10.

Figure 3.12:

trace in Figure 2.1.

At this scale, the bursts of segments in Figure 3.11 still appear as if all the packets were sent at once.[3] Also apparent here is that some of the segments sent near the end of the window are not as long as the rest of the packets. This is surprising since the test program wrote the entire buffer all at once and the TCP should include some mechanism of silly-window-syndrome[4] (SWS) avoidance, although no SWS is apparent. When the sender has refrained from sending completely to the end of the window, zero length packets are sent in response to the duplicate acknowledgments.[5]

[3]On a much closer view not included here they appear about 2ms apart.

[4]The silly-window-syndrome is described in [4].

[5]It is hard to distinguish zero length packets from very short packets at this scale, but those are indeed zero length.

The detailed behavior of the slow-start and congestion avoidance algorithms can be seen in Figure 3.12. There are three slow-start episodes which get beyond the first packet and one which appears as only a single packet. Six packets in Figure 3.12 are retransmissions. Two appear to be caused by retransmit timers going off, two appear to be sent in response to an acknowledgment, and two appear to be caused by the fast-retransmit algorithm[6].

The analysis of these two packet traces points to a serious performance problem in the unimproved TCP. The behavior of this TCP transmitter (its burstiness and its haste in retransmitting all of its retransmit queue) seriously affected throughput and efficient use of the network.

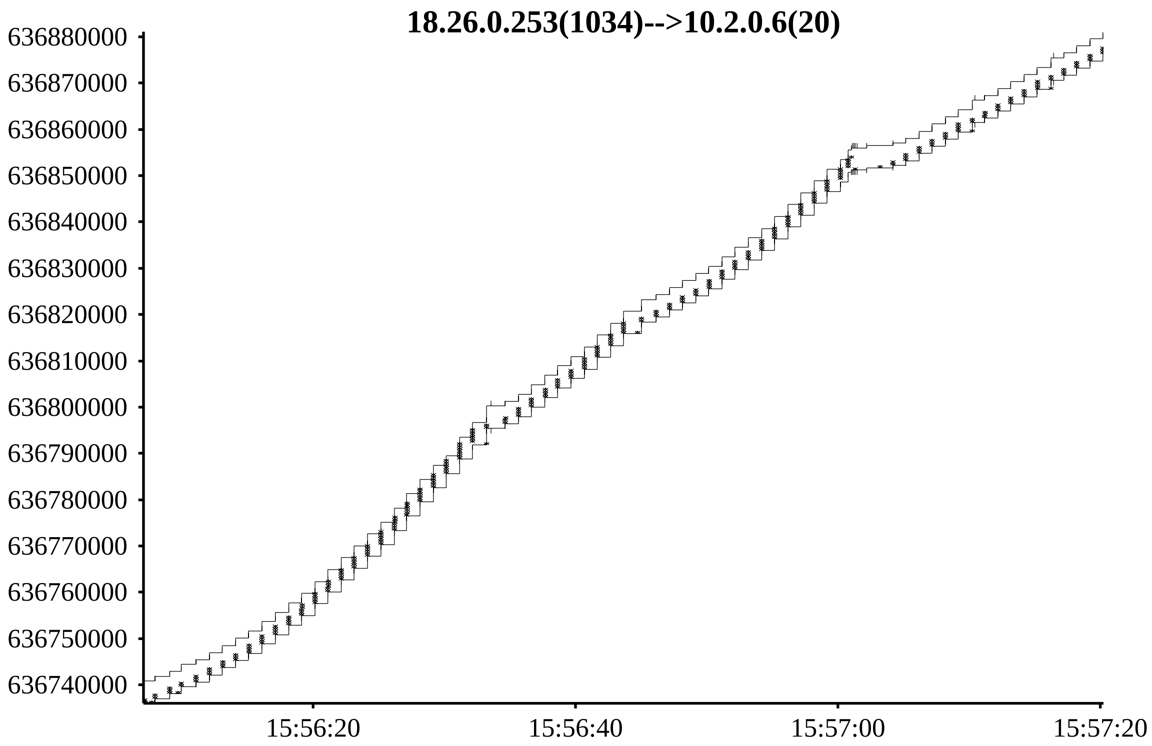
3.3 Interesting Plots

Figure 3.13 is a time sequence plot of a TCP using Jacobson's slow-start algorithm sending to a host which implemented a 200 millisecond dally timer.[7] A dally timer is a feature which can be implemented in a TCP receiver to delay the transmission of a segment to acknowledge received data when more data is likely to arrive soon, or when a segment containing data is likely to be returned soon. The receiving TCP must eventually time-out and return a segment acknowledging the data even if no further data is received, hence a dally timer is set. When the dally timer expires, a segment is sent covering all unacknowledged data.

An alert user had complained that a file transfer from a machine whose TCP had been updated to use the slow-start and congestion avoidance algorithms performed poorly compared to the previous performance under the older TCP which did not incorporate the slow-start and congestion avoidance algorithms. Examination of a

[6]The fast-retransmit algorithm is described in [9].

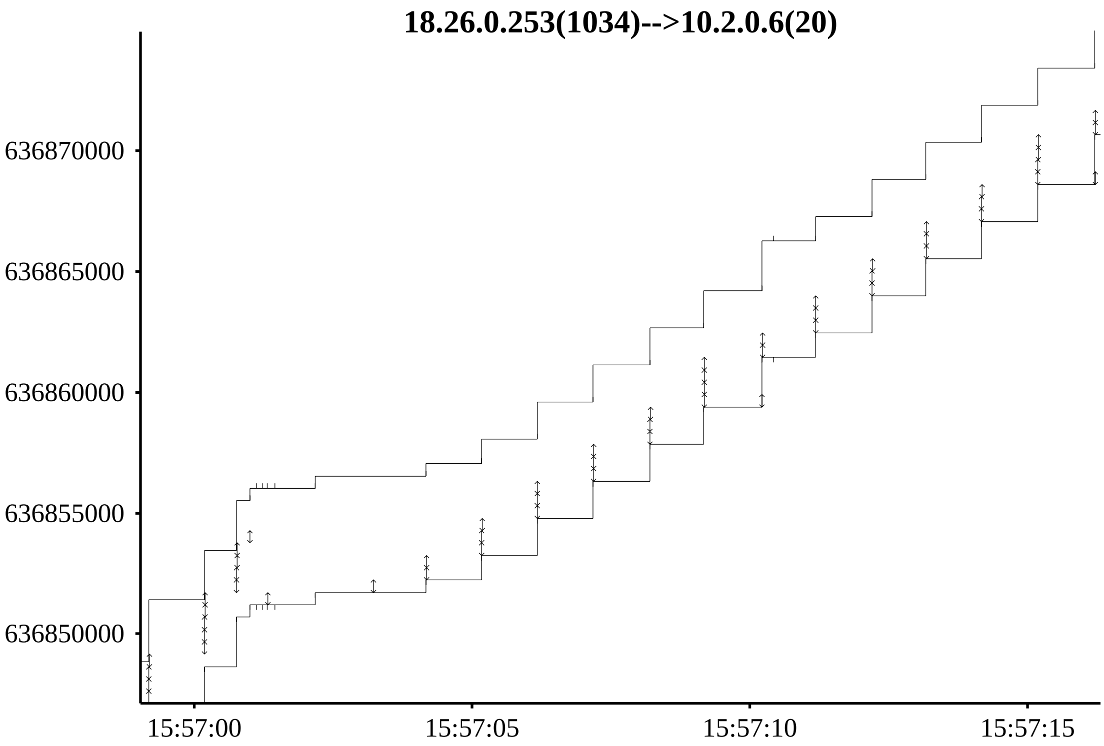
[7]It is currently recommended that a TCP send at least one acknowledgment for every two data-carrying segments received[2]. This TCP would always wait the full 200 milliseconds and send just one acknowledgment regardless of how many packets arrived.



This plot shows what can happen when a TCP using the slow-start algorithm is sending to a TCP which implements a dally timer.

Figure 3.13:

packet trace using time-sequence plots revealed an interesting interaction between the algorithms used in the newer TCP and the dally timer on the receive TCP. Since the newer TCP uses the spacing of the returning acknowledgments to clock out segments carrying data, the dally timer effectively winds up controlling the transmitter. Also, since the opening of the congestion window (a window kept internally by the transmitter in the newer TCP) is controlled by the reception of new acknowledgments and since the dally timer reduces the number of distinct new acknowledgments received, the congestion window was opened much more slowly than intended. Furthermore, the newer TCP uses an improved round trip time (RTT) estimator which models the variance in RTT as well as the RTT itself. Since the apparent RTT was dominated by the dally timer and since the variance in apparent round trip time was very low,

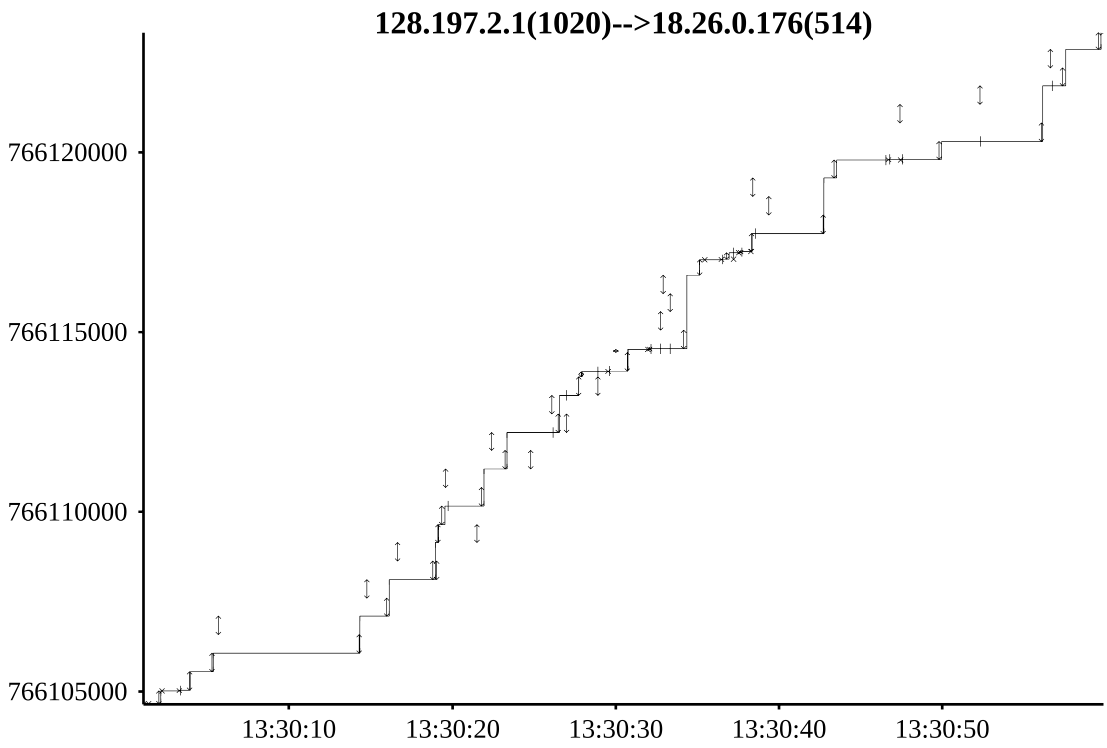


A closer view of a section of Figure 3.13.

Figure 3.14:

often an acknowledgment which was only very slightly later than usual would not arrive in time to prevent the expiration of a retransmit timer and the consequential retransmission and reset of the congestion window used in the slow start algorithm. An example of this can be seen in Figure 3.14 slightly after 15:57:10.

Figure 3.15 is an example of a time-sequence plot of a trace collected near the receiver of a bulk transfer of data. The window line has been omitted in this plot. This plot contains examples of out of order segments and the arrival of duplicate segments which had been previously acknowledged. Previously acknowledged segments appear below the ack line.



This is a time-sequence plot of a trace collected near the receiver with the window line omitted. Out of order packets and duplicate packets can be seen. This demonstrates the relativity of time in a data communication network.

Figure 3.15:

Chapter 4

Bursty Behavior of the TCP Transmitter

Chapter 3 has demonstrated the effectiveness of the graphical tool presented in Chapter 2. This chapter discusses the limits of this tool, and gives an example of a more specialized tool which was used to further explore and quantify the bursty behavior of the TCP transmitter.

The problem of analyzing a TCP packet trace has not been completely solved by the time-sequence plot. An expert who understands TCP and the network is still required to interpret the time-sequence plot and infer what is happening. Even when an expert analyst is available, a detailed analysis of a time-sequence plot still requires the analyst to view the TCP connection as a progression of events in time. This can be too burdensome if one is trying to examine many packet traces.

For example, imagine the manager of a large network faced with the problem of determining if the TCP connections using the network are performing well. The manager might understandably want to know if the resources in the network network are being wasted. The manager would need to perform a survey of TCP connections and somehow screen for those TCP connections which are behaving poorly. Once the poorly performing connections have been identified, the cause of the poor performance would need to be determined so that appropriate steps could be taken to fix the problem.

One task is to survey a large number of connections to determine how well they are performing. Another task is to diagnose the connections which are found to be performing poorly to isolate and diagnose the problems. Time-sequence plots can be effectively used to tackle the second task, but not the first.

A single ethernet might carry 500,000 TCP packets in a single hour. It could theoretically carry 10,000,000 or more in a single hour. A survey for poorly performing TCP connections would ideally look at each of these half million packets. The expert human analyst might be able to analyze 100 packets in an hour manually, and 10,000 packets in an hour by using Time-Sequence plots. (After zooming in to an appropriate scale, the human expert would then walk through the entire trace.) This means that only 2% of the TCP packets on the net could be screened in real time. (Or the analyst could spend more than a week quickly scanning the Time-Sequence plots of TCP packets collected in just one hour.) The manager of a network would not be able routinely to screen a large number of connections for performance problems with time-sequence plots even if the manager was an expert network analyst.

Time-sequence plots are not very useful for large-scale screening because they present all of the data to the user in an unreduced form. Yet, without an understanding of what problems or syndromes to look for it is hard to say what reduction to perform on the data. This motivates exploring how a poorly performing connection might be mechanically identified and how one might screen for such connections.

Towards this goal of finding a screening tool some effort was made at finding a way to draw a picture or plot which characterized the behavior of a TCP connection in a way which would allow faster recognition of problem behavior. The behavior of a long-lived connection is certainly shown in its Time-Sequence plot, but because the Time-Sequence plot of the whole connection does not't show sufficient detail, the analyst needs to zoom in and walk along the trace in time. Figure 3.11 and Figure 3.12

are a good example of this.

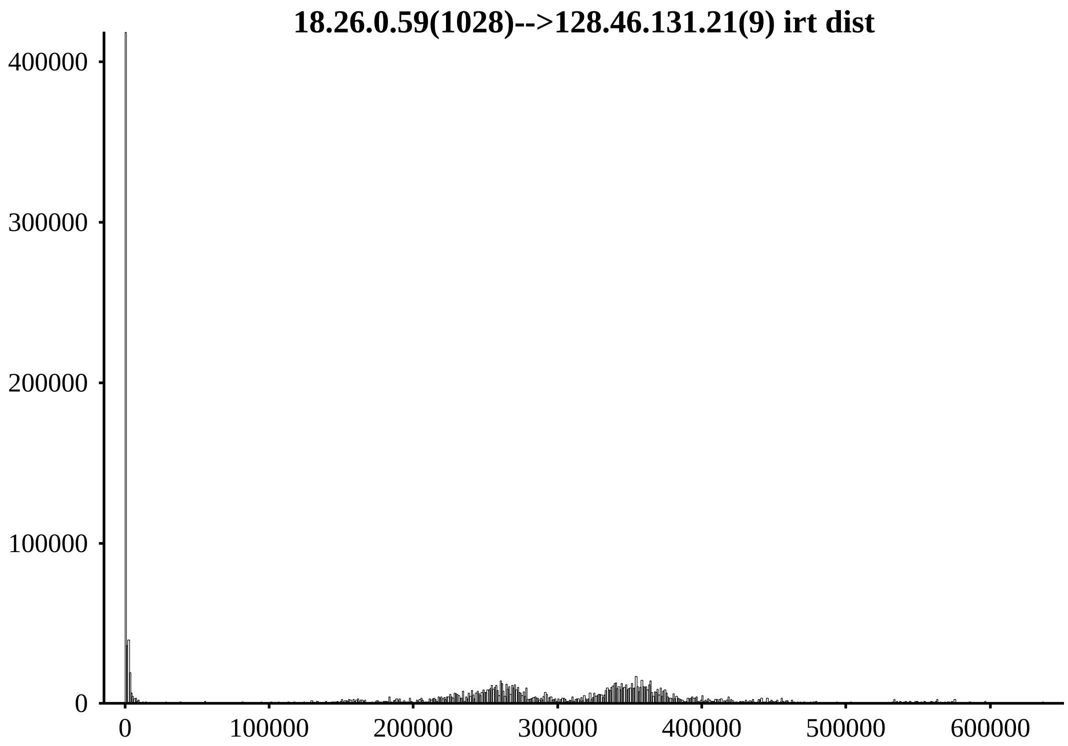
No tableau of a TCP packet trace has been found yet which conveys enough information about the behavior of the connection sufficient to upstage the use of time-sequence plots for performance problem debugging. However, experience looking at time-sequence plots can hopefully lead to some insight towards finding a good method for screening.

One simple observation is that a hypothetical plot or picture for rapidly characterizing the behavior of a TCP connection will not be able to have real time on one of its axes. Having real time on one axis would require the analyst to carry out the examination serially. This has motivated the exploration of some other graphical methods such as histograms and scatter plots. Interestingly, when the use of other plots was explored, the time-sequence plot had to be examined to understand what was really happening. No other plot was found to be as satisfying as the time-sequence plot.

The rest of this chapter will present one graphical method for looking at the burstiness of the TCP transmitter, the most interesting of the graphical methods tried, and discuss how this burstiness might have affected the performance of the two TCP connections presented in chapter 3.

4.1 Burstiness of the TCP transmitter

A major part of the TCP protocol left unspecified by [16] is when a TCP should transmit or retransmit a segment carrying data. The choice of method is left to the implementor. TCP is specified so that interoperation is possible for almost any choice. This choice was left to the implementor to allow TCP to be used as the transport protocol for a wide range of network and computer performance. A simple method is to always send data as soon as it is ready, and retransmit all unacknowledged data when a retransmit timer goes off. This is roughly the approach used in 4.2 BSD TCP implementation. This choice unfortunately can lead to congestion problems

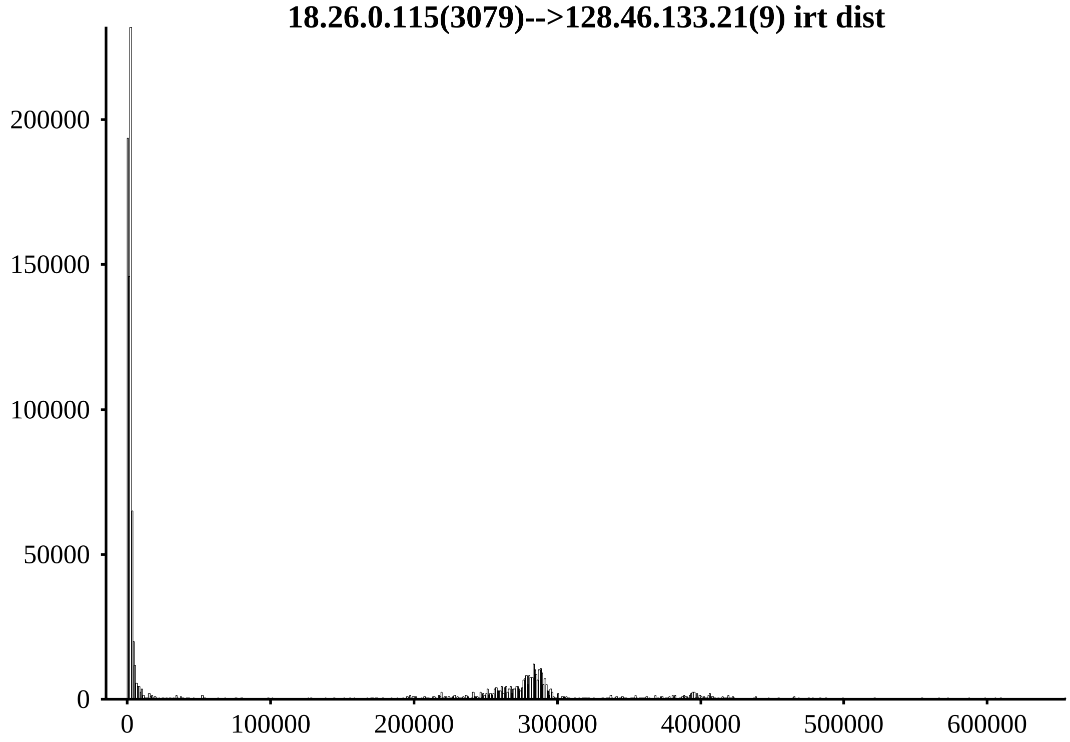


This is the distribution of the instantaneous rate of each byte sent in the packet trace plotted in Figure 3.7. The vertical axis is the number of bytes sent and the horizontal axis is bytes per second.

Figure 4.1:

in the network. Improvements made to this method in 4.3 BSD [11], and later by Jacobson [9], all attempted to reduce congestion by holding back transmissions and retransmissions, effectively reducing the rates at which packets were sent.

Examination of time-sequence plots of the two TCP connections shown in Figures 3.7 through 3.12 motivated a closer look at the behavior of the two TCP transmission algorithms involved. The most striking difference in the behavior of the two TCP transmitters was that the TCP which was performing better was not sending the packets in large bursts. Large bursts were sent by the unimproved TCP, and packets from these bursts were apparently dropped by the network. This motivated an attempt to quantify the burstiness of a TCP transmitter as seen in a packet trace.

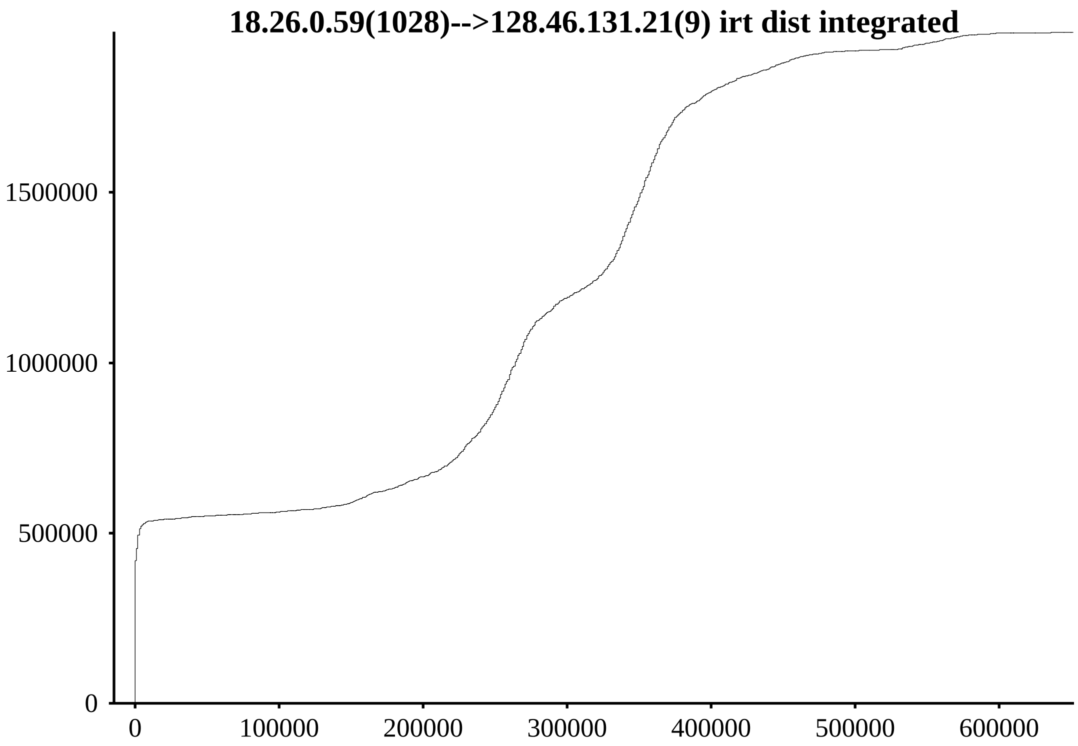


This is the distribution of the instantaneous rate of each byte sent in the packet trace plotted in Figure 3.8.

Figure 4.2:

To quantify the burstiness, determine the rate at which each byte was sent and then examine the distribution of these rates. Let the *instantaneous rate of a packet* be the number of bytes carried in the packet divided by the amount of time since the previous packet was sent. Let the *instantaneous rate of a transmitted data byte* be the instantaneous rate of the packet containing it. Once each packet and byte is labeled in this fashion, a distribution of the number of bytes sent over the instantaneous rate of those bytes can be produced.

Figure 4.1 and Figure 4.2 are histograms of the instantaneous rate of each byte sent on the two connections. The throughput actually achieved by the two connections was roughly 500 bytes per second and 2,000 bytes per second respectively. Both



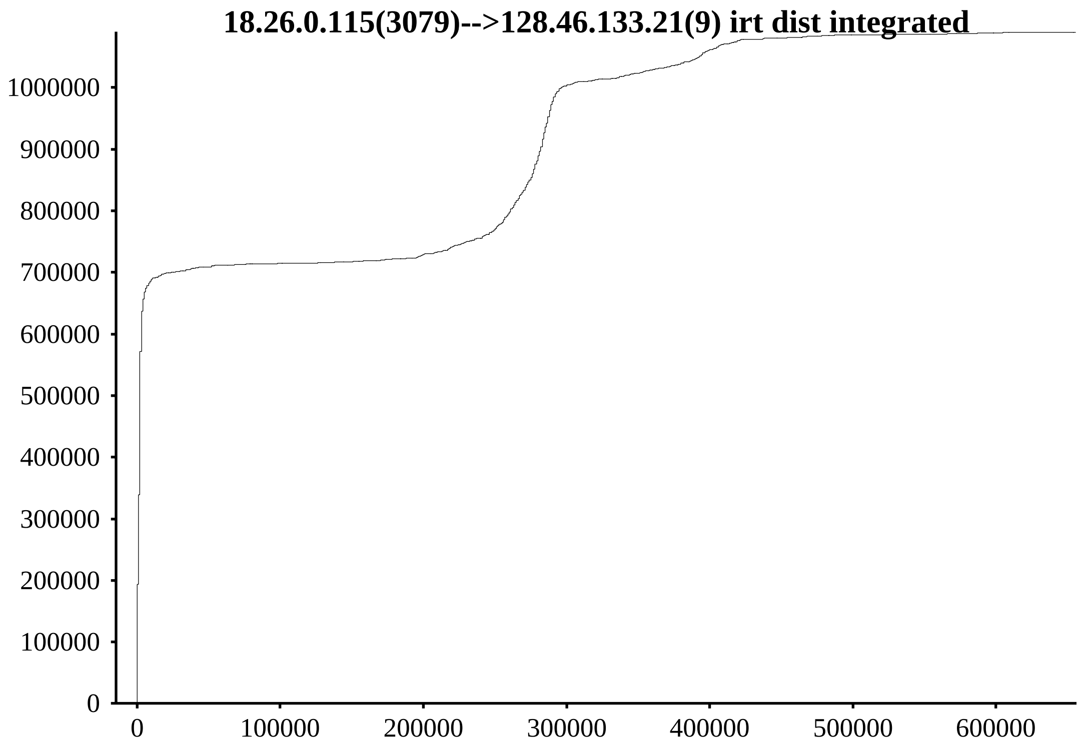
This is the integration of the histogram shown in Figure 4.1. The horizontal axis is in bytes per second, and the vertical axis is in total accumulated bytes.

Figure 4.3:

distributions have large clumps near zero, which represent the bytes in packets sent after a pause, and some bunches at rates a factor of 100 or more faster than the actual throughputs, which represent the bytes in packets sent very soon after the proceeding packet.

A more useful representation of this data is to plot the integration of the histogram. Figure 4.3 and Figure 4.4 are the integrations of the respective histograms shown in Figure 4.1 and Figure 4.2. Now it is very clear how many bytes were sent at each rate.

The ultimate height reached by the line in one of the integrated histograms is the total number of bytes transmitted, including retransmissions. The initial height



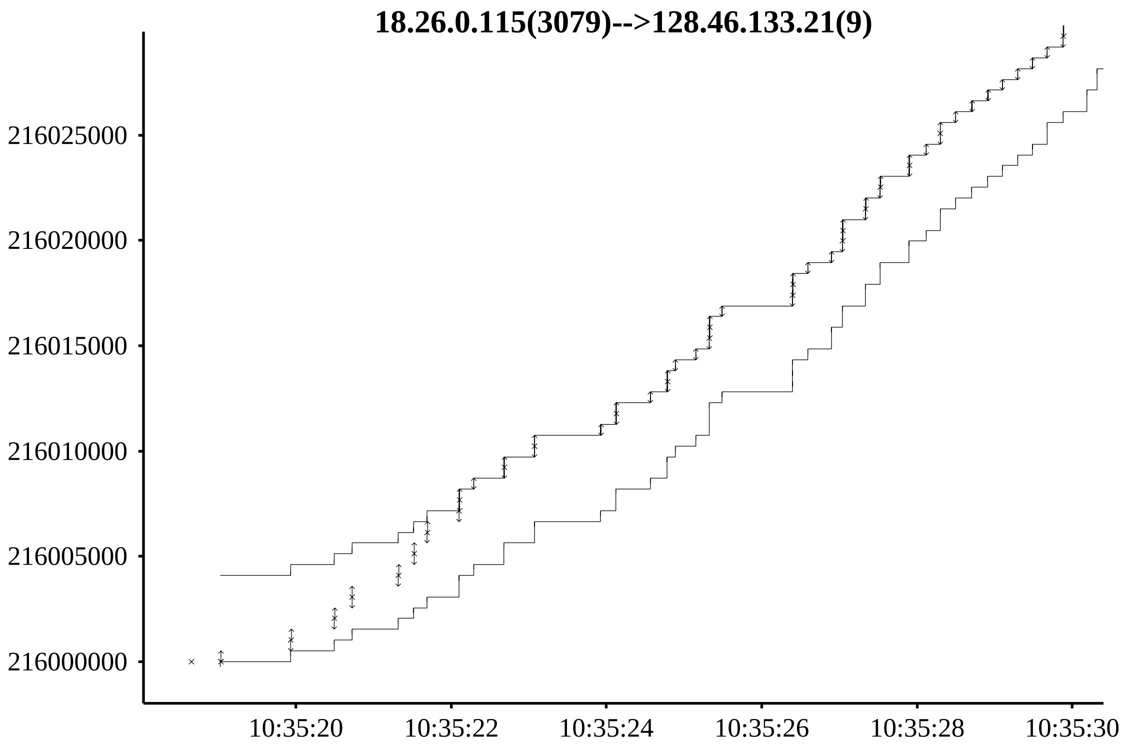
This is the integration of the histogram shown in Figure 4.2.

Figure 4.4:

reached by the line soon after zero indicates the number of bytes sent at a slow rate. Rises in this line occur at the rates at which many packets were sent.

Recall that each connection carried 1,024,000 bytes of user data. The improved TCP transmitted, including retransmissions, only slightly more data than was sent by the user while the unimproved TCP sent nearly twice as much. In Figure 4.4 it is clear that 700,000 bytes (which is about 70% of the total bytes to be transferred) were sent at a slow rate, while in figure 4.3, only about 500,000 bytes were sent at a slow rate.

The integrated histogram of burstiness shows that the improved TCP's behavior was less bursty than the unimproved TCP. This suggests that this measure of burstiness might be used as one measure of the quality or performance of a TCP



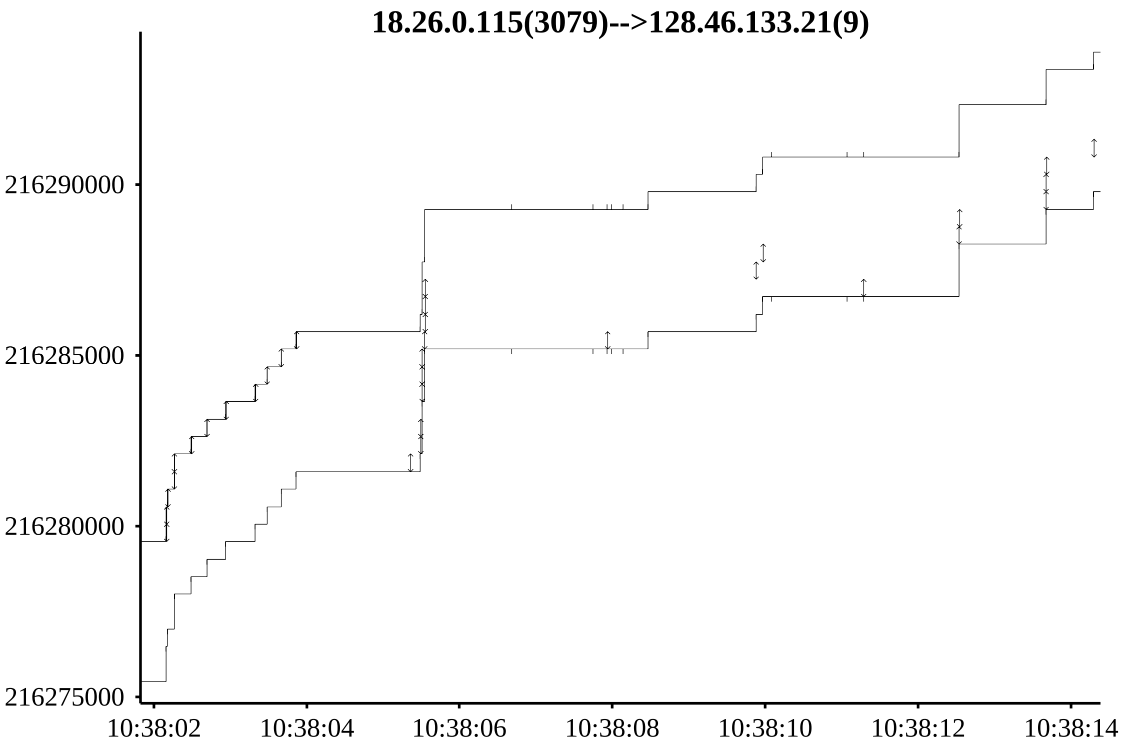
This is an enlarged view of the beginning of the trace shown in Figure 3.8. Examples of packets sent together as the congestion window opens and as acknowledgments are received which cover more than one previously unacknowledged segment can be seen here.

Figure 4.5:

transmitter.

It is not surprising to find that the unimproved TCP sent so many packets at high rates. Recall that in Figure 3.9 and Figure 3.11 that a large fraction of the packets were sent back to back. It is also not surprising that the improved TCP sent fewer packets at high bursty rates.

The improved TCP tries to use the incoming acknowledgments to determine when to send more data. The rationale for this is that the spacing in time of the returning acknowledgments should correspond to the spacing in time of the transmitted packets exiting the bottleneck [9]. So why is there any burstiness at all in the improved TCP?



This is an enlarged view of a portion of the trace shown in Figure 3.8. After 10:38:05 a retransmit timer expired and a single packet was retransmitted. Moments later, an acknowledgment was received which just covered the data just sent. This processed continued and let to a burst of 10 packets.

Figure 4.6:

Further examination of the time sequence plot shown in Figures 3.8, 3.10, and 3.12[1] led to three observed sources of burstiness in the improved TCP transmitter. The first source is when the TCP opens the congestion window allowing more than one packet to be sent in response to an ack for a single packet. The second source is when an acknowledgment is returned which acknowledges more than one unacknowledged packet. A few examples of both of these cases can be found in Figure 4.5.

The third source found in the trace is more surprising. In Figure 4.6 is an example of an incident which is similar to seven other incidents found in the trace. In these

[1]including detail not visible in these figures

incidents, quite a few packets were sent close together in response to a retransmit timer going off moments before some delayed acknowledgments returned bunched together. As each of the acknowledgments from the bunch were processed, more packets were transmitted. This led to a burst of ten packets. Further examination of what follows indicates that the last of the ten packets in the burst was dropped somewhere in the network.[2] The consequence of this large burst is similar to those of the large bursts seen in the unimproved TCP. A packet near the end of the burst is dropped and will need to be retransmitted again later.

A rough accounting of the packets sent in a bursty manner in this trace is 160 packets as part of the normal slow-start algorithm, 30 packets as part of a scenario described in the previous paragraph. The total number of packets sent was around 2000 and from Figure 4.4 30% of the packets were sent in a bursty manner, so 410 packets were sent in a bursty manner due to an acknowledgment which covers more than one unacknowledged packet.

Is burstiness harmful? In the improved TCP 30% of the packets were sent at a high rate, but only 1.5% were involved in scenarios where the burstiness was associated with any packet loss. In the incidents where packet loss did occur, a large burst of around 10 packets occurred. The bursts caused by bunched or batched acknowledgments are usually only 2 to 4 packets long. Burstiness does not appear to be harmful if the bursts are short. Long bursts of 10 or more packets did seem to have a high chance of being associated with a lost packet in the two traces examined.

[2] After the burst of 10 packets, some duplicate acknowledgments arrive. After the third duplicate ack, the a segment is resent. About one half second later an acknowledgment is received covering this segment. Later two more acknowledgments trickle in, each of which releases a packet. Then three duplicate acknowledgments are received, and a packet is sent. Only two of these three duplicate acknowledgments could be in response to the two packets just released. The first of the duplicate acknowledgments arrived too soon to be in response to either of them. Therefore the first duplicate acknowledgment must be in response to the previously retransmitted packet.

Chapter 5

Conclusions

The time-sequence plot has been shown to be a valuable tool for the network analyst. The large reduction of time needed to make a detailed examination of a packet trace should enable a more critical look at the performance of existing and future implementations of transport protocols such as TCP. The ability to display in a comprehensible form the detailed behavior of a TCP implementation has opened a new window into understanding of transport protocol operation.

The high level problem addressed by this thesis is the existence of bugs which are hidden by the robustness of TCP and are manifest only as performance problems. There are two steps in tackling this problem. The first is finding and identifying the bugs. The second is getting them fixed. Recent related work has shown that some definite improvements can be made to TCP transmitter, so the performance problems of TCP in existing networks can probably be ameliorated by incorporation of these improvements into the hosts.

5.1 State of the Art TCP

Currently Jacobson's slow-start and congestion avoidance algorithms [9] are being widely accepted as the correct methods of controlling when a TCP transmitter sends a packet. A look at the detailed behavior of a TCP using Jacobson's algorithms and a brief comparison with the behavior of an earlier TCP presented in this thesis does show that the slow-start and congestion avoidance algorithms seem to perform well.

Even though TCP implementors are given license to choose a wide variety of transmission strategies, they should be aware and make good use of the algorithms which have been developed and studied previously. Historically much of the wisdom needed to implement the TCP/IP protocol suite well was not available formally from a single source but was passed around among implementors as folklore. The Host-Requirements RFC [2] will help ensure that implementors are more aware of this wisdom.

What further improvements to TCP can be made? The problematic behavior of an up-to-date TCP shown in Figure 4.6 might be improved by adding selective acknowledgments, rate-controlled transmission, or both. Adding selective acknowledgments would require an extension to the TCP protocol and would only work when connected to another TCP which has implemented the extension. A rate controlled transmitter could be added independently.

One way to view Jacobson's algorithm is that it sets a rate to send at during one round trip. It is not really a rate, but rather the amount that can be sent in one round trip time, which has the same dimensionality as rate. Jacobson's TCP does not directly send at this rate but rather uses the incoming acknowledgments to clock out the packets. If the acknowledgments are arriving in bursts, then the transmitted packets will be sent in bursts. Since this TCP has already estimated the round trip time and determined how many packets to send in the time of one round trip, a simple improvement might be to pass these two parameters onto a rate-based transmitter to smooth out the bursts.

Use of a rate-based transmitter would have to be studied carefully to be sure that it would not lead to congestion. One possible way of assuring that a rate-based transmitter does not cause further trouble might be to only release packets to the rate based transmitter which would have been released by the current slow-start algorithm. Both algorithms could be run in parallel and each packet would need to be released by both algorithms before it could be sent.

5.2 Robust Systems Mask Faults

The trace of the 4.3 BSD TCP shows just how wrong things underneath the surface can be when protocols are designed to be robust. The 4.3 BSD TCP implementation which produced the troublesome packet trace shown in Figure 3.9 was a widely used TCP implementation in the Internet just a few years ago and is probably still in use on many hosts. It is probably common for this TCP to behave just as poorly as shown in the diagrams in Chapter 3, yet few users are aware that this behavior is so common. Though some may notice that the network seems to be slow and congested at times, few probably have any idea that one-half or more of the bandwidth is being lost to a poor retransmission strategy.

The continued existence of this behavior in a widely used system points out the need for effective monitoring of complex systems, particularly when the system compensates automatically for problems. Automatic error recovery is a good method for making systems robust, but when it hides underlying problems from those responsible for tending to them, either warning lights need to be added to the systems or the managers of the systems need effective monitoring tools and need to know how to use them effectively.

5.3 Automating TCP Packet Trace Analysis

The time-sequence plot enables detailed examination of the packet trace by a human, but does not automate the process itself. Automating the routine examination of TCP packet traces will require specifying much of our current knowledge about how this is done by human experts. The time-sequence plot will enable better understanding of the process of packet trace analysis and provides a good form for representing traces for the purposes of communicating ideas about examination of traces. As more experience is gained examining TCP time-sequence plots, the task of codifying the process of analysis should be made easier.

A screening tool capable of passing first judgment on the performance of a TCP connection as shown in a trace would be a good first step at aiding the analyst. By using a screening tool to identify particular traces for further analysis, it might be possible to carry out a large-scale audit of the performance of the TCP connections carried across a network. In order to be used in a large-scale study, it would not be possible to spend a large amount of time on each trace so the tool would have to use fairly simple algorithms to scan the traces. Exactly how this screening tool would work is not clear now, but experience looking at time-sequence plots of many connections suggests that it would not be too difficult.

One observation after looking time-sequence plots of many random connections from the Ethernet being monitored is that most of the connections are quite boring. Most TCP connections are between hosts on the same ethernet and are operating in one of two modes: keyboard typing and echo or bulk data transfer. By *boring* I mean that there are few surprises found when looking at these connections. Bulk data transfers across a single ethernet either run in lock step mode limited by the window offered by the receiver or do manage to stream somewhat.[1] Connections providing remote login service typically carry little data and exchange one packet per keystroke. Timeouts and retransmissions are rare on both of these types of connections. This observation that most all of the TCP connections are boring suggests that instead of trying to filter for connections with performance problems, it might be easier to filter out the boring connections and study what remains.

A initial definition for a *boring* TCP packet trace might be one in which all seg-

[1] Both of these behaviors can be seen in Figure 3.5.

ments are acknowledged before the next segment is sent, the transmitter is never limited by the window, and no segments are retransmitted. Another possible definition might be simply that no retransmission occurred. Whatever the definition, the filter should probably be configurable and somewhat extensible to allow the analyst to filter out common occurrences peculiar to the particular net being monitored. Learning and quantifying common behaviors on a particular net would probably become part of the analysis process. The filtering tool might count the events it has been configured to recognize as an aid to this quantification.

Appendix A

Tools

Before packet traces can be examined and analyzed, they must be captured, and if any of the analysis is not going to be performed in real time, the trace must be stored for later retrieval. This appendix will discuss the some of the reasoning behind the design of the packet capture and storage system and will describe briefly how it was constructed and how it is used to provide traces to the analysis programs.

A.1 Design Issues

Early in this project, FTP Software's LANWatch was used to capture packet traces off of the net. It is a program which runs on an IBM-PC with an ethernet interface and displays packets as they arrive and optionally logs packets to a file on the disk. LANWatch includes the ability to filter on a per-packet basis and allows separate filters for displaying and logging.

There were a few problems with using this system to capture packet traces for analysis. The first is that the speed of the PC was not fast enough to capture and log all packets (or even all TCP packets) on the net without missing an unacceptable number of packets. This required that the user know in advance what packets might be of interest. Another problem with this system was that while the PC was collecting packets, it could not do anything else. This means that packets could not be captured while the trace was either being examined or moved off of the PC onto another machine.

Experience with using the IBM-PC and LANWatch-based system for capturing packets led to the following design goals:

1. The system should be able to gather essentially all the packets on an ethernet.

2. The system should allow the trace of packets to be collected and accessed (for analysis) simultaneously.

3. The system should be designed so that it can be left gathering packets continuously at all times.

The first goal is essential to allow later analysis of the packet traces. In particular, if the system is to be usable for general purpose network debugging, it would be unreasonable to expect the user to know in advance which packets would be needed for later examination. If some packets are omitted from the trace because of the system's inability to gather packets at a high rate, then a burden will be placed on the subsequent analysis. The second goal is implied by the third goal unless the traces are never examined. The third goal was driven by the desire to not miss anything on the ethernet. By collecting all packets, the decision about which packets to look at can be deferred. The ability to go back and look at individual packets minutes to hours after something unexplained or strange is observed on the network is very useful.

A more general goal of this research was to not limit the analysis to experiments, but to also examine some packet traces of real day-to-day TCP connections. It would be difficult to collect traces of real TCP connections if the system did not't collect all of the packets, though one could use a system which watched for TCP packets, or TCP packets which indicate that they are the initial packet of a connection, and then filtered for the rest of the packets associated with the same connection. Without carefully designing the selection process, this might have undesirably skewed which connections were likely to be selected for gathering.

A.2 4.3BSD Unix Based Packet Trace Collection

A system for gathering and storing packet traces meeting the goals outlined above was integrated into a 4.3BSD Unix system running on a MicroVAX-III. This system has many advantages over the previous IBM-PC based system. An important advantage is that under Unix's multi-processing, a background task can be set running to gather and store the packet traces in the file system. This neatly meets the second and third goals outlined above. Most of the advantages are related to the richer environment the standard Unix utilities provide. For example, the standard Unix filters such as *grep* and *awk*, provide a good first cut at a tool box for tools to filter and manipulate the traces. By using these tools, the time spent on developing the infrastructure can be reduced. Another advantage is that sources for the entire Unix system were available for study and modification. This allowed the necessary changes for packet collection and buffering to be easily incorporated into the Unix kernel.

A.3 Packet Trace Collection System

The packet trace collection system consists of three parts. The first is support added to the kernel to collect the packets. The second is a program running in a user-mode process which reads the packets out of the kernel and saves them into files. The third is a library used by programs to access the traces from the files.

The first part, the part in the kernel, was modeled after the *nit* socket which is a standard part of Sun's Unix operating system. A new implementation of Sun's nit socket originally written at Project Athena was adapted to packet trace collection and integrated into the standard 4.3BSD Unix based workstation software. It includes hooks into the Unix device driver of the network interface being used. Upon receipt of each packet, the device driver's interrupt routine calls a routine `nit_input()` which copies onto a queue a small structure (containing a timestamp and length information) and the first 50 bytes of the packet and schedules a lower priority handler. The lower priority handler moves the packet off of the queue and onto a socket which has been opened by the user-mode program.

The user-mode program reads out of the socket in 10 kilobyte chunks and writes the data out to 1/2 megabyte files. When each file reaches 1/2 megabyte, it is closed and a new file is opened. The user-mode program also monitors the amount of free space left in the filesystem. Whenever the filesystem gets too full, it deletes the oldest packet-trace file. By always deleting the oldest data, the largest amount of recent data can always be made available for analysis.

The library for accessing the packet traces hides the fact that the traces are kept in multiple files. It provides primitives for finding the first packet after a given time and for moving to the next packet and to the previous packet. In its current implementation, it reads entire files into buffers. The interface to the library hides this so that on an operating system which allows mapping of files, the files could be directly mapped into the address space of the analysis program without changing the analysis part of the program.

The system runs on a MicroVAX-III computer running 4.3BSD Unix with about 370 megabytes of disk space available for packet traces. Depending on the network load, about one half of the CPU time is free while the packet trace collector is running and can be used by analysis programs or other programs. The packet trace collector is run at an extreme high priority so that other use of the machine does not interfere with the collection of packet traces. The remaining CPU time on the machine has even been used by other people doing work unrelated to packet trace collection and analysis with no noticeable effect on the packet trace collection.

A simple utility *print* is provided for printing out a portion of the trace. It takes two arguments on the command line. The first is a time in the form hh:mm:ss and indicates at what point to start scanning the trace. (The time is assumed to be from the proceeding 24 hours.) The second is the number of seconds of the trace to scan. It formats and outputs all the packet headers which occurred in the period specified. This utility alone has proven to be very useful. It provides an after-the-fact netwatch-like capability integrated into a Unix environment. By using this print program with the standard Unix tools such as *grep* and *awk* many useful tools can be rapidly prototyped.

Other more specialized programs were developed using the library to list TCP connections, count packets on each connection, filter for a given TCP connection, and to generate a time-sequence plot of a given TCP connection. Each of these programs are only a page or two long. Most of the functionality needed by these programs is available in libraries.

The time on the workstation was kept accurate (to within about ten milliseconds) by ntpd. Ntpd is a system program that uses the Network Time Protocol described in [13]. The kernel was modified to keep time using some external microsecond resolution clocks on the MicroVAX-III so that the packets could be timestamped to microsecond resolution when the interrupt is received from the Ethernet interface. Because this timestamp is taken after the packet has been relieved it is more closely related to the end of the packet on the Ethernet instead of the beginning (which differs by at most 1.2 milliseconds for a maximum length ethernet packet). There is also some unknown amount of jitter caused by the time taken to DMA the ethernet packet across the bus and for the interrupt to be serviced.

A.4 Use and Experience

The packet collection system has been in continuous operation for over nine months. At any time, the previous 8 to 16 hours of packet trace are available for analysis.

An important feature of this system is that a packet trace can be examined after an unanticipated failure or anomaly. If the system required that traces be anticipated, then only experiments are possible and we would never get a candid look at the network. Even in situations where the desire for a trace could be anticipated (e.g. before performing an experiment), not having to target the packet trace collection system in advance is a great convenience.

Keeping just the last disk full and always deleting the oldest traces makes good sense. If anything else were kept, then it would reduce the amount of recent data that could be kept.

The system for capturing packets was integrated into the kernel of the Unix system running on a workstation. The 4.3BSD Unix kernel provided an important substrate in which to integrate the packet trace collection system. The system for handling interrupts, scheduling network level processing, and handling the disk for logging did not have to be built from scratch. The socket code in 4.3BSD Unix provided the primitives for buffering and communicating with a user mode-process.

The system works surprisingly well given that no special purpose hardware was used. Special purpose hardware would have been useful to obtain more accurate timing information and for monitoring low-level phenomena on the Ethernet. For example, collisions on the Ethernet and packets with bad CRC's are never seen by the monitoring software.

Bibliography

[1] R. Aronoff, K. Mills, and M. Wheatley. Transport layer performance tools and measurement. *IEEE Network*, 1(3):21–31, July 1987.

[2] R. Braden. *Requirements for Internet Hosts — Communication Layers*. Request for Comments 1122, DDN Network Information Center, SRI International, October 1989.

[3] Robert T. Braden and Annette L. DeSchon. *NNStat: Internet Statistics Collection — Introduction and User Guide*. USC / Information Sciences Institute, release 2.4 edition, December 1989.

[4] D. D. Clark. *Window and Acknowledgement Strategy in TCP*. Request for Comments 813, DDN Network Information Center, SRI International, July 1982.

[5] David D. Clark. The design philosophy of the darpa internet protocols. In *Proceedings of SIGCOMM '88*, August 1988.

[6] David C. Feldmeier. *Empirical Analysis of a Token Ring Network*. Technical Report MIT/LCS/TM-254, Massachusetts Institute of Technology, January 1984.

[7] Eman Salaheddin Hashem. *Analysis of Random Drop for Gateway Congestion Control*. Master's thesis, Massachusetts Institute of Techonology, August 1989.

[8] Bruce Hitson. Knowledge-based monitoring and control: an approach to understanding the behavior of tcp/ip network protocols. In *Proceedings of SIGCOMM '88*, August 1988.

[9] Van Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM '88*, August 1988.

[10] Raj Jain and Shawn A. Routhier. Packet trains – measurements and a new model for computer network traffic. *IEEE Journal on Selected Areas in Communications*, SAC-4(6):986–995, September 1986.

[11] Michael J. Karels. *Changes to the Kernel in 4.3BSD*. University of California, Berkeley, 4.3 bsd edition, April 1986.

[12] Allison Mankin and Kevin Thompson. Limiting factors in the peformance of the slow-start tcp algorithms. In *Proceedings of USENIX — Winter '89*, January 1989.

[13] D. L. Mills. *Internet Time Synchronization: the Network Time Protocol.* Request for Comments 1129, DDN Network Information Center, SRI International, October 1989.

[14] Jon Postel. *Internet Protocol.* Request for Comments 791, DDN Network Information Center, SRI International, Sept 1981.

[15] Jon Postel. *SMTP.* Request for Comments 788, DDN Network Information Center, SRI International, Nov 1981.

[16] Jon Postel. *Transmission Control Protocol.* Request for Comments 793, DDN Network Information Center, SRI International, Sept 1981.

[17] Dheeraj Sanghi, M. C. V. Subramaniam, A. Udaya Shankar, Olafur Gudmundsson, and Pankaj Jalote. *Instrumenting a TCP Implementation.* Technical Report UMIACS-TR-88-50, CS-TR-2061, Institute for Advanced Computer Studies, Computer Science Department, University of Maryland, July 1988.