51

Analyzing the behavior of TCP Implementations

by

Pedro A. Zayas

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 21, 1998

Certified by ....
David D. Clark
Senior Research Scientist, Laboratory for Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Analyzing the behavior of TCP Implementations

by

Pedro A. Zayas

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1998, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The main objective of this thesis is to provide a framework for evaluating the correctness of TCP implementations. We use this framework in order to assess the completeness and correctness of four widely available implementations: DEC OSF 3.2, FreeBSD 2.2.2, Windows 95 and Windows NT 4.0. Through our evaluation, we will attempt to determine which TCP mechanisms each implementation provides and whether these mechanisms are implemented correctly according to TCP standards. We also show that our approach can be extended to evaluate the communication properties of applications that use TCP as their mean of communication. Our approach is to analyze real TCP traffic passively; that is, we obtain packet traces from the network and, later, we evaluate this traffic. In order to facilitate the evaluation process, we represent TCP traffic in graphical format. The development of a network traffic tracing tool was necessary to conduct the evaluation. The tool uses already existing packet tracing technology and xplot, a graphical tool to represent network traffic.

Thesis Supervisor: David D. Clark
Title: Senior Research Scientist, Laboratory for Computer Science

Acknowledgments

I would like to thank my thesis advisor, Dave Clark for his guidance, support, encouragement and patience all along this project. My experiences working on this project would not have been as enlightening and fulfilling if I had not been working with such a bright and enthusiastic person.

Dorothy Curtis, Lewis Girod, Garret Wollman and John Wroclawski illuminated me with their technical knowledge in every single step through my project.

Sally Floyd and Mark Handley provided target hosts to run my experiments across the Internet.

Rob Cheng, Wenjia Fang, Dina Katabi, Dan Lee, Atanu Mukherjee, Alex Snoeren and Rena Yang, my officemates from the ANA group, for making the process of thesis writing much more enjoyable with their conversations.

Finally, I would like to thank my parents, Carmen J. Ortiz and Pedro A. Zayas, and my brothers, Orlando and Arnaldo Zayas for their love and unconditional support. For tolerating and even welcoming my many late phone calls asking for advice or just someone to help me relieve my stress. Now, more than ever, I realize that I am infinitely fortunate to have such a wonderful family always there for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The network research community has invested considerable efforts into developing efficient communication protocols for today's network architectures. The Transmission Control Protocol (TCP), a reliable connection-oriented transport protocol, is one protocol that has been studied extensively. Issues such as reliability, effective throughput and congestion control have been investigated in TCP traffic, and efficient mechanisms to cope with them have been developed as a result. However, the question of whether different TCP implementations actually deploy these mechanisms, and whether they work correctly has not been addressed adequately. The goal of this thesis is to provide a framework for evaluating TCP implementations, and to use this framework in order to assess the completeness and correctness of four widely available TCP implementations: DEC OSF 3.2, FreeBSD 2.2.2, Windows 95 and Windows NT 4.0. We also show that our approach can be extended to evaluate the communication properties of applications that use TCP as their mean of communication.

1.1 Identifying incorrect TCP behavior

Over the years, TCP has become the most widely used transport protocol in the Internet Protocol (IP) suite. One of the features that has contributed to the popularity of this protocol is its ability to provide reliable data communication. TCP was designed to operate over intrinsically unreliable networks that can duplicate, reorder, damage or even lose data packets. Networks can also suffer from other unfavorable pathologies such as traffic congestion, point-to-point delivery latencies that vary over time and changes in the path taken by traffic, among others. To cope with these pathologies, TCP provides robust mechanisms that can handle practically any network irregularity short of a total network failure in order to achieve reliable communications.

However, reliability is not the only feature that TCP offers to users. Several mechanisms were developed in order to optimize the performance of transactions in terms of data throughput, efficient use of bandwidth and timely recovery from packet losses. Nevertheless, reliability is more readily noticeable than performance from the point of view of users and applications. Failure to deliver data reliably will very likely result in unexpected and incorrect behavior in transactions, thereby revealing the existence of flaws in TCP mechanisms. More importantly, reliability is trivial to verify by simply transmitting data, and then verifying its reception at the end host.

Performance flaws, on the other hand, are much harder to detect. While some knowledgeable users might be able to notice performance problems in TCP transactions, most users do not know what kind of services to expect from networks. It is extremely difficult to differentiate if poor transaction performance is due to deplorable network conditions or to errors in TCP mechanisms. Furthermore, TCP does not provide any feedback to inform users about the kind of network conditions it is encountering, whether it is continually losing packets in the network or any other irregularities. Users can only tell whether a transaction completed successfully. In this manner, the reliability mechanisms in TCP can effectively help to conceal performance related errors during transactions. In order to evaluate the performance mechanisms in TCP implementations it is necessary to explicitly study the behavior of their transactions, not from the perspective of the user, but by actually looking at the behavior of TCP.

Evaluating the performance of a TCP implementation requires different methods than evaluating its reliability mechanisms. This thesis concentrates in providing a framework for evaluating the performance of TCP implementations that are known to "work" (i.e. they reliably complete transactions under reasonable network conditions), but for which performance has not been certified.

1.2 TCP Performance in Modern Networks

The overwhelming pace of technological developments over the last few decades has triggered a massive demand for computational resources and techniques. The area of data communications has been no exception. As microprocessors become faster, parallel and distributed computation become more popular, and new applications demand more computational power, networks play a key role in allowing these technologies to be deployed fully and efficiently. Similarly, other widely available technologies such as the World Wide Web have become very popular in the last few years, thereby placing even more stringent performance requirements on today's network architectures and consequently, on protocols such as TCP. To this end, extensive research efforts have been invested in devising mechanisms and algorithms that improve the performance of TCP transactions.

TCP's performance mechanisms are designed to improve the behavior of individual transactions as well as to achieve the efficient use of network resources. When congestion is detected, hosts decrease the rate of outgoing data in order to avoid increasing network congestion even further. Because of TCP's popularity we expect the aggregate congestion avoidance effort of all hosts to have a noticeable impact on the condition of internetworks. Hence, avoiding network congestion is ultimately favorable to individual users as well, since in the long run their traffic will be able to arrive to its destination with less delays and with a lower probability of being lost. However, the benefits of TCP performance mechanisms will only be enjoyed if TCP implementations actually implement these mechanisms correctly. Failure to implement the performance mechanisms correctly will quite possibly result in poor transaction behavior and high levels of network congestion. Undoubtedly, these outcomes are undesirable for individual users as well as for the Internet as a whole. Therefore, the verification of TCP mechanisms is of great interest to researchers, users, and consequently to vendors of TCP implementations.

But not only is TCP performance an issue in today's network architectures, it is also a concern in the design of future network technologies. New technologies are being developed in order to provide the base for the services that networks will offer in the future. Areas such as traffic multicasting and Quality of Service – among many others – are currently being investigated because they are considered to be essential technologies in the process of building a useful framework for tomorrow's network architectures. The designers of these schemes often need to make assumptions about traffic characteristics in today's networks so they can provide appropriate solutions to existing problems. Many researchers assume that TCP traffic in today's networks actually conforms to TCP standards. Some researchers even propose to explicitly penalize non-conforming TCP implementations in order to ensure fairness in the utilization of network resources [6]. Hence, the verification of the correctness of TCP implementations also serves to justify and validate the assumptions of network researchers and to guarantee its users that their traffic will not be at a disadvantage with the development of future networks.

This thesis provides a framework for evaluating the correctness of the mechanisms available in TCP implementations. We provide a tool that facilitates the analysis of TCP traffic in real networks by representing this traffic in a graphical format that greatly simplifies the task of visualizing traffic behavior. We will use this graphical representation in order to assess the correctness of performance-critical mechanisms implemented in four widely available TCP implementations.

1.3 Organization of this Thesis

In Chapter 2, we present similar attempts to evaluate the performance of TCP implementations, and we also look at currently available tools that simplify the process of analyzing TCP traffic. In Chapter 3 we present the methodology that we will use to do our analysis, and we discuss its advantages and limitations. We also discuss how to minimize external interactions that can affect TCP traffic when experimenting in real networks. Chapter 4 provides a brief description of TCP and the mechanisms that we will evaluate in TCP implementations. We also present a description of our graphic representation of TCP traffic which will facilitate our evaluation process. This representation is the same used by Timothy Shepard in [15]. In Chapter 5, we analyze four TCP implementations and we provide a complete assessment of the pathologies we encountered in each. We demonstrate the incorrect behavior of the implementations by presenting graphic instances of their traffic in which they violate TCP specifications. In Chapter 6 we discuss how the performance of TCP can be affected as a result of its interaction with the network layer. We also discuss briefly how our tool can be used to evaluate the performance of higher level protocols and applications that use TCP as their mean of communication. Finally, Chapter 7 provides some concluding remarks.

Chapter 2

Related Work

There have been a number of attempts to evaluate TCP implementations in the past, but most of them employ evaluation techniques that are different than ours. The main distinction is that our approach is considered to be *passive* since we do not directly control the flow of TCP traffic in order to observe an implementation's behavior[1]. Similarly, a number of tools that facilitate the analysis of TCP traffic have been developed in the past. This chapter summarizes the previous work done in both of these areas.

2.1 Other evaluations of TCP implementations

This section summarizes other attempts to evaluate TCP implementations. We briefly discuss the experimental approach followed in each case in order to differentiate it from our *passive* approach. We also include a list of the TCP implementations they evaluated.

Comer and Lin used a technique called active probing in which the behavior of TCP implementations is evaluated by observing their reaction to controlled events such as loss of connectivity [4]. In their study, they evaluated the following TCP implementations: IRIX 5.1.1, HP-UX 9.0, SunOS 4.0.3, SunOS 4.1.4 and Solaris 2.1.

Brakmo and Peterson evaluated the performance of the TCP implementation

[1]We describe our experimental approach in detail in Chapter 3.

found in BSD 4.4 (a.k.a. TCP-Lite, Net/3 TCP) [2]. Their approach was to evaluate the behavior of this implementation using a simulator that executes the source code directly.

Stevens analyzed the TCP traffic incoming to a World Wide Web server running Net/3 TCP [17]. While he found a number of errors in the connections he observed, he could not determine the implementation of the TCP initiating the traffic. Much like our technique, his approach was passive.

Dawson, Jahanian and Mitton developed Orchestra, a "software fault injection" tool, and used it to evaluate the following TCP implementations: SunOS 4.1.3, AIX 3.2.3, NeXT (Mach 2.5), OS/2, Windows 95 and Solaris 2.3 [14]. Their software tool acts as a filter between the network and the TCP implementation, controlling the packets that the TCP sends and receives.

Paxson evaluated the behavior of TCP implementations automatically using a tool that analyzes packet traces off-line (i.e. also passive approach) [11]. His tool first attempts to determine which TCP version each implementation is based on, and then according to this version the tool evaluates the mechanisms that the corresponding version is supposed to provide. He studied the following TCP implementations: BSDI 1.1, 2.0, 2.1a; DEC OSF/1 1.3a, 2.0, 3.0, 3.2; HP-UX 9.05, 10.00; IRIX 4.0, 5.1, 5.2, 5.3, 6.2a; Linux 1.0, 2.0.30, 2.1.36; NetBSD 1.0; Solaris 2.3, 2.4; SunOS 4.1; Trumpet/Winsock 2.0b, 3.0c; Windows 95 and Windows NT.

2.2 Available tools for evaluating TCP behavior

One of the contributions our work is the development of a tool that facilitates the process of evaluating the behavior of TCP implementations. Our tool obtains packet traces directly from the network, storing them into disk and later displaying this traffic in a graphical format when the user requests it[2]. This section summarizes other existing tools for analyzing TCP traffic behavior.

Tcpdump is a very popular packet tracing package that is available for a variety

[2]We describe our tool in Chapter 3.

of platforms [19]. It allows users to conveniently filter traffic so as to avoid storing unwanted or unnecessary traffic. Besides its filtering capabilities, tcpdump also allows the user to print different types of information about individual packets (e.g. ethernet header, IP header, TCP header, etc.). In order to provide this flexibility, however, the software is inevitably bulky which is unacceptable and unnecessary for the kind of tool we wanted to create.

We also know of two existing packages that allow TCP traffic to be represented in graphic format. In order to develop our tool, we used xplot which was specifically designed to represent TCP traffic [15]. Tracelook is a similar tool written in Tcl/Tk, that converts the output of tcpdump traces into a graphical representation. We chose to use xplot in our tool because of our familiarity with the tool.

Finally, we know of two packages that allow for overall analysis of TCP implementations: Tcpanaly and Orchestra. Given a traffic sample from a particular TCP implementation, tcpanaly attempts to determine which TCP version the implementation is based on, and then according to this version the tool evaluates the mechanisms that the corresponding version is supposed to provide [11]. In other words, it allows for automatic *passive* traffic analysis of TCP implementations.

Orchestra, on the other hand, supports the *active* analysis approach by allowing the user to specify how and when packets are dropped, delayed, reordered, etc. Thus, orchestra facilitates the task of observing the behavior of TCP implementation in reaction to specific events. Orchestra was used in [14] to evaluate several TCP implementations.

Chapter 3

Analyzing TCP Traffic in Real Networks

In the process of evaluating our TCP implementations we collected traffic from real networks and then evaluated the behavior of transactions by analyzing this traffic. In order to validate our analysis and evaluation approach when experimenting in such real environments, we considered all the *external* factors that can potentially affect TCP traffic in practice. Then, we attempted to control these factors so that our conclusions about TCP implementations are not biased. In this chapter, we discuss the interactions between TCP and other external entities that can affect traffic patterns, and we present our methodology to control them during our experimentation process. We also discuss the advantages and limitations of our approach to analyzing TCP traffic.

3.1 A passive approach to TCP traffic analysis

In order to evaluate the behavior of TCP implementations we analyzed traffic in real networks by obtaining packet traces and, later, presenting the traffic in a graphical format. Representing traffic graphically simplifies and accelerates the process of evaluating network events[1]. This approach of obtaining packet traces and then evaluating them off-line is known as passive analysis [11]. The techniques generally used in other studies are more active in the sense that they control the traffic flow of TCP connections and observe the response to specific stimuli. Instead, we let connections flow freely and study their "natural" behavior.

Analyzing connections passively allows us to visualize the behavioral problems that normally arise in TCP connections. We also avoid the deterministic conditions that controlling TCP traffic artificially can create, and we concentrate on observing phenomena originated in real networks. As we discuss later in this chapter, controlling the traffic flow of TCP transactions can be very helpful in testing mechanisms that are designed to handle *unusual* network conditions. However, if we intend to evaluate the performance of transactions under normal conditions it would be far too complex to synthesize all the possible situations that TCP can encounter under average network conditions. Therefore, our passive approach is more effective and complete if we want to evaluate the performance of transactions under common situations.

One disadvantage of our passive approach, however, is that it depends on the occurrence of network events in order to encounter "interesting" TCP behavior. Consequently, we need to design our experiments in a way that allows us to control certain parameters even in real networks. For instance, consider the congestion avoidance mechanisms in TCP. These mechanisms are only activated when packets are dropped somewhere in the network. Hence, we need to wait for the network to "naturally" drop packets in order to evaluate their behavior. Another example is the round trip time (RTT) of a TCP connection which directly affects retransmit timers. It is of great interest to determine whether significant differences in RTT affect the correct behavior of the timers.

While we have no control over the traffic conditions that prevail in real networks at some point in time, we do know that there is a correlation between the time of the day and the network congestion we observe. For example, traffic between MIT (Massachusetts) and Lawrence Berkeley Labs (California) encounters more conges-

[1]We describe our graphic representation of TCP traffic later in this chapter.

tion during weekdays from 11AM to 5PM. Hence, we can schedule our experiments according to this time-to-traffic correlation so that we are likely to encounter the level of congestion we desire. Similarly, we can control the round trip time of a connection by varying the location of the target host in our experiments. During our experiments, we had hosts within the MIT LCS lab, in Berkeley, California and in London, England allowing us to vary the round trip times of connections. Therefore, while our passive approach has the disadvantage of not giving us full control over the prevailing network conditions, we can work around this problem by carefully designing experiments in real networks, under real usage constraints.

Another flaw of our passive analysis is that we lack the ability to test the behavior of transactions under *unusually* deplorable network conditions. Since it is impossible to predict when these conditions will arise in networks, we can not expect to encounter situations that test the mechanisms in TCP that are specifically designed to cope with poor network conditions. Such events might actually never occur while we monitor the network. Instead, for these rather extreme cases we need to explicitly disconnect the target host, or drop the packets sent by the target host so that the source host does not receive the packets.

However, we expect these situations to be rare in real networks so we do not need to consider them closely from a performance perspective. Improving the performance of transactions under common network conditions is more beneficial than improving the performance under these less probable cases since we expect the common case to be the dominant factor. This does not mean that fault tolerant mechanisms need not be tested. If these uncommon situations were to arise in real transactions, TCP implementations should be able to handle them appropriately. However, we do not evaluate these mechanisms in our tests because we want to concentrate in the analysis of TCP traffic under common network conditions, for which passive analysis is sufficient. In order to analyze fault-tolerant mechanisms we need to rely on active analysis such as that used in [4], or in [14].

3.2 Generating TCP traffic

The TCP traffic patterns we observe in real networks are affected by interactions between the host generating the traffic and many other entities that are not necessarily related to the network. In this section we consider how these interactions might affect our analysis, and present our solution to the problem of generating TCP traffic that interacts with relatively few external entities.

Our experiments concentrated in bulk data transfers between hosts; the data transferred was generally 1MB long. This type of transaction models the traffic pattern for applications that send large amounts of data all at once, such as the File Transfer Protocol (FTP). This approach fails to consider the behavior of TCP traffic when the application constantly sends small packets, or when the traffic pattern is bursty. A more complete analysis of TCP performance would also evaluate TCP under these kinds of traffic behavior, but due to the complexity of developing a traffic generator with these characteristics we leave the matter open for future work.

During our experiments, we avoided using actual FTP transactions and other similar applications because they can introduce undesirable interactions that affect traffic behavior. The hosts that generate the traffic during our evaluations can potentially be running other processes while we conduct our tests. Most modern operating systems use timesharing to give the impression that multiple processes are being executed simultaneously. Therefore, applications that have a lot of overhead besides that of sending and receiving data from the network might not handle traffic efficiently because they might not be given sufficient running time by the operating system. Since we only wish to test the performance of TCP implementations, this kind of external interactions are undesirable because they may lead to incorrect conclusions about TCP performance.

In order to generate our traffic we used **ttcp**, an application that does little work besides that of sending or receiving data. Ttcp has the capability of sending random data without having to look for it in the disk, and discarding all the data received which eliminates the overhead of writing data to disk. But even with the use of a "thin" application such as ttcp, we still can not claim that the traffic we generate is not affected by external factors. The processes that control the behavior of TCP are still affected by the operating system and the end hosts' hardware among other factors. The use of ttcp in our analysis at least allows us to identify when suboptimal timing related behavior exist, and guarantee that they are introduced by entities that lie below the TCP layer. Our analysis can not differentiate between problems that are the result of incorrect or inefficient TCP implementations from those caused by other factors within the host. Attempting to identify exactly where timing related problems exists within a host would require tools and techniques that are particular to each platform, and are beyond the scope of this work.

3.3 Packet traces and measurement issues

When using this approach of obtaining packet traces in real networks, we need to address the fundamental issue of the accuracy of the packet traces themselves. The packet filtering modules we used to obtain our traces can potentially drop packets for reasons such as hardware errors or overflows in the pre-allocated buffers. While the second situation can be solved by designing our tool to read packets from the buffers in a timely manner, there is always the possibility of losing packets during our traces. Therefore, when analyzing the obtained packet trace we need to keep this issue in mind so we do not reach erroneous conclusions.

Packet filters also have the task of time stamping the packets as they arrive to the interface. Time stamps are critical when observing the reaction of hosts to certain events, and when plotting our traffic in graphical format. Potentially, there exists a variation in the time it takes for a packet to be stamped after it has been received. Also, there exist the question of how accurate these timestamps are. Fortunately, BPF, the packet tracing package we used to implement our tool, timestamps packets as soon as they enter the interface card. Therefore, the relative timing between events that appears during our analysis is as accurate as the clock provided by the host where our tool operates. Generally, the accuracy of these clocks is reasonable to a granularity of tens of microseconds.

Finally, we also need to consider where the packet filtering occurs with respect to the sender and the receiver of the TCP transaction. If we take our measurements on the receiver side, we can not evaluate the behavior of the sender without considering the fact that the sender is working on a different time scale than what the trace reflects. In particular, the sender sees outgoing packets before what the trace suggests, and incoming packets much later than the trace. Similarly, it is hard to evaluate the behavior of the receiver when tracing the traffic from the sender side. We simplify our evaluation process by only analyzing the behavior of the side that is in the same network as our the tracing tool.

3.4 A tool for analyzing TCP traffic

In order to conduct our proposed evaluation of TCP traffic we needed a tool that can reliably obtain network traffic traces and present the traffic in the desired graphical format[2]. We considered the possibility of using an existing tracing tool that obtains traffic and displays it in text format such as tcpdump [19], and then converting the textual representation into our graphical format. This approach, however, is slow and tedious since we need to go through this process after every experimental TCP connection. We also considered modifying tcpdump to produce its output in graphical format. This approach would accelerate 0the process of obtaining the graphical representation. However, because tcpdump provides a myriad of other features and functionalities, the size of the software would have been unnecessarily large for our purposes since we would only be using the graphical representation module. Additionally, we wanted our tool to have a better user interface than the one provided by tcpdump.

Instead, we decided to create our own tool for TCP traffic analysis. Our tool uses the same packet tracing techniques used by tcpdump, and it facilitates the process of going through connections and displaying the traffic in graphical format. Our tool

[2]We discuss our graphic representation of TCP traffic in Chapter 4.

only implements the feature of converting raw network traffic into graphical representation, which makes the software considerably smaller. There are two major modules in our network traffic tool: the packet tracing module and the traffic analyzing module. This chapter discusses these modules in more detail and provides a comparison between our tool and other available tools.

3.4.1 Obtaining packet traces

Our packet tracing techniques are very similar to those employed by tcpdump. We use available packet filtering technology in order to record TCP traffic. We intended our tool to work on machines running the FreeBSD operating system: a free distribution of BSD-like OS for Personal Computers. Since FreeBSD supports the BSD Packet Filter (BPF) tool, we decided to use it in order to obtain our packet traces[3]. The packet tracing module implemented in our tool, "programs" BPF to accept every TCP packet seen in the local network and then it stores it in files so that we can analyze this traffic later. In order to save space we only store 100 bytes of each packet. This length is enough to capture the header information of each packet which is the only necessary information for TCP behavior analysis. If the disk where we are storing our traces ever gets full, we erase the earliest recorded file so that we can keep only the most recent traffic always stored. The packet tracing module is designed to operate permanently; even when we are not analyzing traffic.

As it can be readily seen from this description, our packet tracing module is rather simple. The packet tracing capabilities offered by tcpdump are far more complex, and unnecessary for our purposes. For instance, tcpdump allows the user to select traffic involving specific hosts, protocols and many other characteristics. While we realize that these features could prove very useful in certain circumstances, they are not necessary for the purposes of our analysis. In addition, tcpdump does not provide the flexibility of recording traffic in separate files that could be erased as the disk space is consumed. While we could have modified tcpdump to provide this feature, we

[3]See [10] for more information on BPF

deemed this solution inefficient since our tool can provide this feature in a much more concise manner by eliminating the unnecessary functionality included in tcpdump.

3.4.2 Analyzing the traffic

The traffic analyzing module in our tool provides a simple user interface and allows the user to search for "interesting" TCP behavior. The interface provided to the user is in graphical format and was implemented using the TCL/TK toolkit software. Using simple mouse-driven commands, the user can obtain a display of all the TCP connections that were seen during a certain period of time. By simply double-clicking on the entry corresponding to a connection, the user obtains a graphical xplot display of that connection's traffic. If the connection is still active, the display is updated after new traffic is received.

Additionally, along with the information for each connection, the module prints the ratio of data actually sent and the data that was intended for communication. As we will see in Chapter 4, packets can sometimes be retransmitted, so the number of bytes actually transmitted can be greater than the number of bytes that were originally intended for transmission. We believe that the ratio of data sent and the data intended for transaction can help identify interesting TCP behavior. The rationale for our hypothesis is that TCP traffic does not need to be retransmitted if the traffic is flowing under normal circumstances. In these cases, the sending TCP just sends data as fast as it can and never runs into any problems, thereby minimizing the possibilities of producing "interesting" behavior. On the other hand, if traffic is constantly being retransmitted, there is a higher possibility that we will encounter more difficult network conditions that can trigger less normal TCP behavior. The purpose of this ratio is to serve as a guideline when selecting which connections to study more closely, and in no way implies that TCP implementations that do not retransmit packets are correct.

Our tool also allows the user to select connections involving specific hosts or applications just as tcpdump. The advantage of our packet tracing tool is that it obtains traces for every TCP connection so we can always go back and study other connections if we decide to. The disadvantage, of course, is that our tool unnecessarily uses disk space storing packet traces, even in the cases when we are certain that there is some traffic that we do not care about. This, however, was a trade off that we were happy to undergo given the simplicity with which we were able to implement the packet tracing module.

Chapter 4

Overview of TCP

TCP is currently the most widely used transport layer protocol in the IP suite. Probably, the main reason for its popularity is that TCP provides a connection-oriented, reliable data communication service. We say that this service is *connection-oriented* because whenever two hosts want to communicate with each other using TCP they must establish a *connection* between them before they can exchange data. We also say that this service is *reliable* because whenever a transaction completes, the user is guaranteed that the data transmitted arrived successfully to its destination. In order to guarantee reliability, TCP provides special mechanisms that can handle traffic that travels over intrinsically unreliable networks. For instance, TCP has mechanisms that can handle the duplication, reordering, corruption or even the loss of data packets. Reliability, however, is not the only feature that TCP offers to users. TCP also contains mechanisms that are designed to optimize the performance of transactions. When we refer to the *performance* of transactions in this context, we refer to data throughput, the efficient use of bandwidth and the timely recovery from packet losses.

In this chapter we present an overview of TCP and its mechanisms for reliability and performance. Our approach will be to present the basic properties of TCP, and then discuss the more complex mechanisms using our graphical representation of TCP traffic. This approach will serve two purposes: First, it will simplify the task of explaining these mechanisms, since the graphic representation allows for a better visualization of TCP events. Secondly, using the graphical representation in this chapter will familiarize the reader with the format we will use to present our evaluation of TCP traffic in later chapters. We start with an explanation of how TCP connections are identified in the internet, and with a discussion of the very basic characteristics of TCP. Then we introduce our graphical representation of traffic by showing how we represent these very basic traits of TCP in graphical format. Finally, we combine our basic knowledge of TCP and our graphical representation in order to describe some of the mechanisms that guarantee reliability and performance[1].

4.1 The basics of TCP connections

Every TCP connection can be uniquely identified by the IP addresses and the TCP port numbers of the two ends involved in each connection. An *IP address* is a 32 bit number assigned to every host that uses the IP suite so that each host is uniquely identified in the entire Internet. A TCP *port number* is a 16 bit number assigned to each connection on a particular host. Some port numbers serve to identify specific "services" that use TCP (e.g. HTTP traffic uses port number 80), but they are mostly useful in identifying different connections within a single host. Every TCP packet that travels through the Internet has this and other relevant information stored in uniform data structures called *headers.*

In order to guarantee the reliable delivery of TCP data, TCP also labels data segments within each connection using *sequence numbers.* Labeling data segments greatly simplifies the task of detecting when packets are reordered or lost. Each TCP packet carries in its header the sequence number of the first byte of the data segment it carries, and enough information to calculate the length of the segment. For instance, if we send 100 bytes of data in four segments of 25 bytes each, the sequence number of the four packets will be 0, 25, 50 and 75 respectively. Note that if any of the packets is received out of order or is not received at all, the receiving TCP will see a discontinuity in the packet sequence.

When packets successfully arrive to their destination, the receiver notifies the

[1]For a complete description of TCP see [13].

sender by replying with an acknowledgment packet (ACK) that contains the next sequence number it expects to receive. In our example, when the target host receives the packet with sequence number 0 and length 25, it returns an ACK with an acknowledgment sequence of 25 which is the next sequence number it expects. The sequence numbers of connections need not always start at 0 as our example shows. In fact, the initial sequence number can be any number between 0 and 2^{32}. If the sequence number of a data segment goes over the upper limit, it wraps around continuing at 0 again.

4.1.1 The sliding window scheme

When a TCP connection is opened, each host allocates memory in order to store the data received from the other end. There are two reasons why the receiver needs to allocate this memory: First, when TCP receives data from the network it does not immediately pass this data to the application for which it is intended. Instead, TCP waits until it has a "buffer length" worth of data for the application and then it gives the entire buffer to the application[2]. In most implementations, applications have the ability to set the size of this buffer. The default buffer length is usually 8192 bytes. TCP implementations, however, have the flexibility of maintaining bigger buffers in order to speed up the network transaction as long as it gives the data to the application in fragments equal to the buffer length requested.

The second reason why hosts pre-allocate buffers is because TCP needs to present the data to the application in the same order with which the data was sent. Therefore, if a packet is lost or is received out of order, the recipient should store the rest of the data that has arrived, until the missing packet is retransmitted or it finally arrives. When the packet is received, TCP arranges the buffer so that the sequence of data segments preserves the order with which it was sent, and then gives the buffer to the application. If TCP did not store the data that arrives out of order, then the source host would need to retransmit all the data segments starting with the one that was

[2]Unless the PSH bit in the TCP header is set, in which case the TCP "pushes" all the data that it has in its buffer to the application.

lost or reordered, thereby increasing traffic unnecessarily.

Intuitively, the amount of data sent by a host should never exceed the buffer space that the receiver has allocated. In order to avoid overflowing the receiver's buffer the sending host needs to limit the amount of outgoing data. However, since the size of the allocated buffer can be set by each application, the sender can not know the size of the receiver's buffer a priori. TCP solves this problem by having each host advertise a window size in each packet sent. The *window size* represents the maximum amount of unacknowledged data that a host can receive at a particular moment. *Unacknowledged data* is data that has already been sent by the source, but has not yet been acknowledged by the receiver. As the amount of data being stored in the buffers increases, the receiver *closes* the window thereby indicating that its available space for new data is decreasing. When the receiver hands the data to the application, its buffers are free once again so it can reopen the window and the sender can continue transmitting new data.

This strategy is called the *sliding window* scheme, and it is explained fully in [3]. Intuitively, we can imagine one end of the window being always maintained at the last unacknowledged sequence, and the other end being maintained at the upper limit of available buffer space. The window can close from below (as the buffers start filling up) or it can slide in order to allow the source host to send new data (whenever TCP passes the data buffers to the application). Each connection has two windows – one in each direction – and each is controlled by the side that receives data in that particular direction.

4.2 Graphic representation of TCP traffic

As we can readily appreciate, the mechanisms included in the TCP standard are often complicated and involve very specific behavior. The original efforts to evaluate TCP implementations relied on the use of packet tracing tools that transform network traffic into text format. This analysis, however, proved to be immensely tedious and it promoted the development of more efficient ways to represent traffic. One of the first efforts in this area was the work done by Timothy Shepard in his development of **xplot**, a graphical tool to represent TCP traffic [15]. We decided to use xplot in our evaluation of TCP implementations because of our familiarity with this tool and because of its availability. This section serves as a tutorial to the representation of traffic used by xplot which we will use later in order to describe more complex TCP mechanisms, and to complement our evaluation of TCP implementations.

TCP traffic easily lends itself for graphical representation because it uses sequence numbers to maintain the order of the data transmitted in a TCP connection. The representation employed by xplot presents time in the x-axis and sequence numbers in the y-axis. Packets are presented with a single x-coordinate (the time when the packet is traced from the network) and a line that extends in the y-direction. The y-coordinates of this line correspond to the sequence numbers that the packet is carrying. The acknowledged sequence and the advertised window are lines that run parallel to the x-axis until they are incremented; the increments are shown as vertical steps in the line. As an example, figure 4-1 presents a normal data flow of a TCP connection. The arrows are the packets flowing in time, and increasing sequence number. The bottom line with steps is the acknowledgment line, and the top line is the advertised window line. The reception of acknowledgment packets can be identified by the vertical steps in the acknowledged sequence line. Similarly, increments in the window can be identified by vertical steps in the advertised window line. As the transaction makes progress, the sender controls the flow of packets (or arrows) and the receiver controls the growth of the acknowledgement line and the advertised window line.

Notice that a correct TCP implementation should never send packets that are not within the "envelope" formed by the acknowledgment sequence and the advertised window. If a packet has already been acknowledged by the receiving host, there is no need to retransmit the packet so there should not be any packets below the acknowledgment line. As we discussed in section 4.1.1, the sending TCP should never send more data than advertised by the receiving host, so there should not be any packets above the advertised window line as this would violate the sliding window

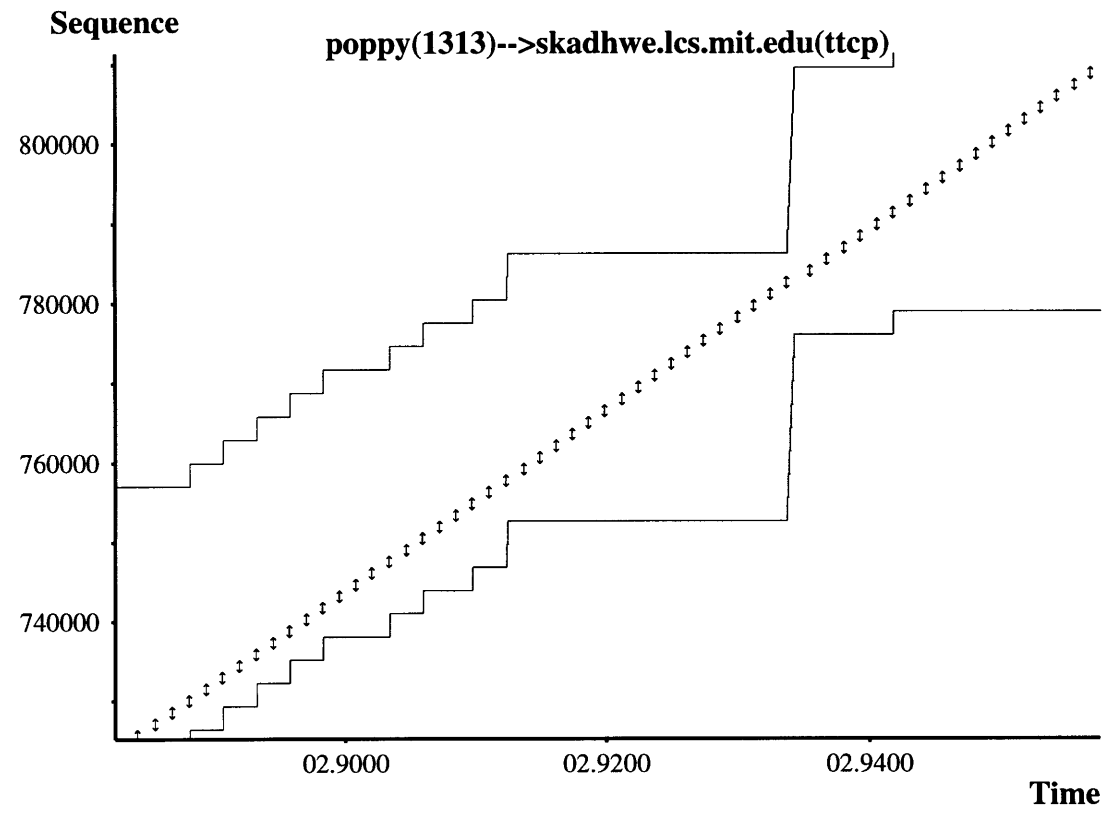

Figure 4-1: Normal packet flow of a TCP connection

scheme. As we can see from this example, representing traffic in this graphical format greatly simplifies the task of analyzing the behavior of TCP traffic.

4.3 Performance mechanisms in TCP

Now that we have discussed the basic properties of TCP and that we have a useful format for representing TCP traffic, we explain more complex mechanisms that enhance the performance of TCP transactions.

4.3.1 Slow start and congestion avoidance

Because TCP data travels over networks that are subject to congestion, hosts can not assume that the network has enough resources to handle all of its traffic. Therefore, even if the receiving end advertises a certain window size, the sender needs to be cautious in terms of the amount of data it sends at once. In order to minimize the number of packet lost during a transaction, TCP has specialized mechanisms that avoid sending too much data when the network is congested. The idea behind this strategy is to optimize the efficiency of a transaction by limiting the number of outgoing packets when there is a higher probability that packets will be dropped. Additionally, this approach avoids incrementing network traffic during times of congestion which would result in even more congestion.

In order to implement this behavior TCP needs be able to detect network congestion when it occurs. TCP standards specify that the sender should interpret packet losses as an indication of network congestion. The rationale behind this interpretation is that the network drops packets only when it lacks enough resources to handle the existing level of traffic. When the sending host detects a packet drop, it assumes that the network is not able to handle its current sending rate, so it slows down in order to avoid loosing even more packets. But how can TCP tell what the right sending rate is for the existing level of congestion?

Jacobson's slow start algorithm specifies that the source host should send a small number of packets first, and then depending on whether the network is able to handle this traffic volume, it should increase the sending rate gradually [8]. TCP keeps an internal variable **cwnd** (for congestion window) that determines the amount of unacknowledged data that it can have outstanding given the existing level of congestion. In the beginning of the connection, TCP does not know what the existing level of congestion is so it starts with a very small value for cwnd[3]. Since cwnd limits the amount of data that a host can send, the sender increases cwnd gradually as long as the network is able to successfully deliver its traffic, and decreases its value if the network looses any of its packets.

During the slow start phase, when an ACK arrives from the receiver, cwnd is incremented by one packet. The rationale is that the reception of the ACK indicates that the network was able to handle the packet that triggered the ACK, and that the sender should probe the capacity of network to see if it can successfully handle another

[3]Usually one packet worth of data although this is a source of current debate.

packet worth of data without facing too much congestion. Notice that increasing cwnd in this fashion results in an exponential growth in its size. Figure 4-2 shows a host entering the slow start episode right after the connection establishment phase. The sender starts by only transmitting a single packet, and then for each ACK received cwnd is incremented by one packet. Hence, for each ACK that arrives, the sender transmits one new packet to "substitute" the packet that was just handled by the network, and another that is the result of cwnd increasing by one packet. Note that the number of outstanding packets increases exponentially as a function of round trip time (RTT)[4]. This scheme ensures that hosts will not begin a connection sending data too aggressively, and ultimately end up flooding the network with traffic. At the same time, it also ensures that the sending rate increases fast enough so as to takes advantage of the network resources that are available if the network is not congested.

When a packet is lost, cwnd is reduced to one half of its value. The idea is that since the host roughly increases cwnd by a factor of two for each RTT, whenever a packet is lost TCP concludes that the value of cwnd for the previous RTT yielded an approximately optimal sending rate for the network to handle. From now on, if the cwnd is below this optimal level TCP will increase the its value aggressively and if the sending rate is above the optimal level it will increase its value more conservatively. To achieve this behavior, TCP keeps a second internal variable **ssthresh** that determines when to stop doing slow start (increasing cwnd exponentially) and when to start doing congestion avoidance (more conservative cwnd growth). Usually, ssthresh is initialized to 65535 bytes; this number is intended to be very large so that the sender initially attempts to increase its sending rate exponentially until the network drops at least one packet.

Whenever a packet loss is detected, ssthresh is set to one-half the value of cwnd. After this, cwnd is set to its initial value once again and the transaction continues with the retransmission of the lost packet. From now on, if the value of cwnd is smaller

[4]A round trip time is the amount of time that it takes a packet to travel to its destination and then back to the source. We generally approximate it to the time elapsed from the moment a packet is sent until the moment the ACK for the packet arrives to the source.

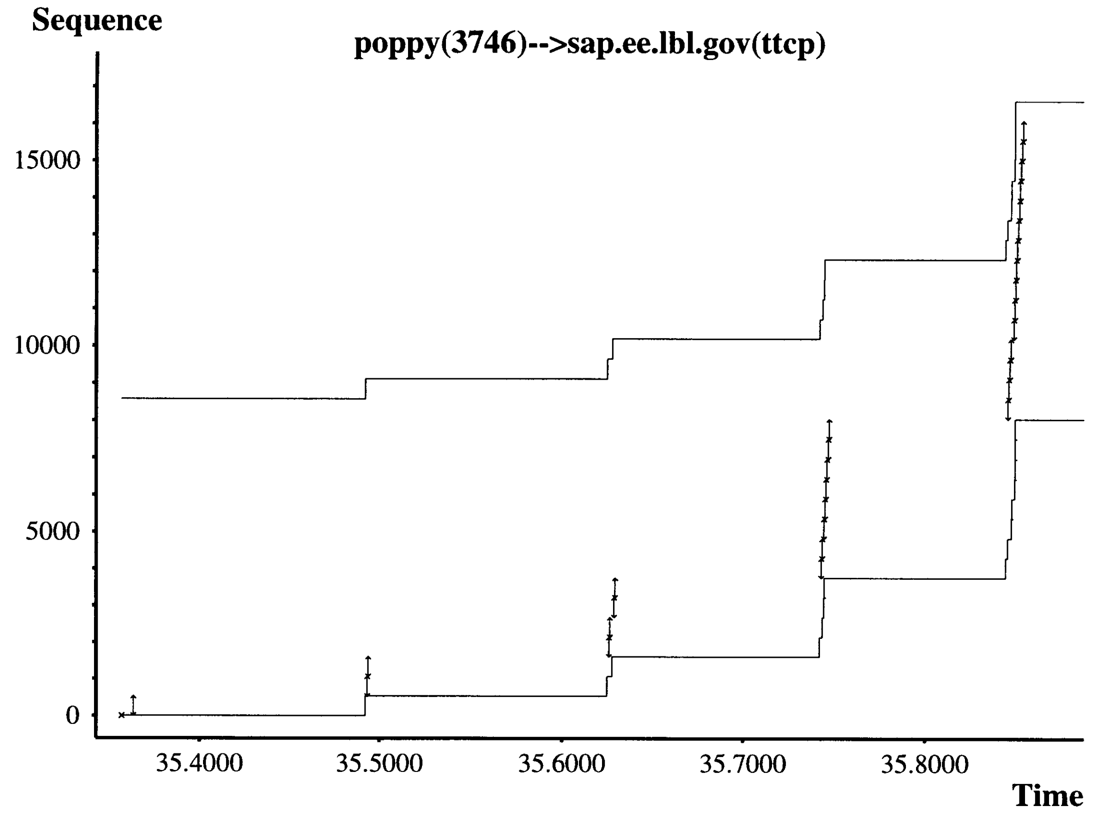

Figure 4-2: Connection establishment and slow start episode

than ssthresh, cwnd will still be increased by one packet after the reception of each ACK. However, if cwnd is greater than ssthresh, cwnd is increased by the reciprocal of its current size (one divided by cwnd). The idea is that cwnd will now increase by one packet after every RTT so that the growth is linear rather than exponential. We should point out that that the sender should never send more data than the receiver can handle even if the network has enough resources to deliver more traffic. Hence, whenever TCP decides whether it will send another packet it needs to consider both the window advertised by the receiver and the congestion window, and then select the smallest of the two; if there is enough space to send new data then a new packet can be sent.

The mechanism for recovering from packet losses explained in this section is inefficient because it makes the sender go into the slow start phase after a packet loss, when in reality we know that the network can handle at least half of our current sending rate. In section 4.3.3 we discuss a much more efficient mechanism to recover from packet losses.

4.3.2 Delayed Acknowledgments

Another factor that affects the performance of TCP transactions is the transmission of small packets. Sending small data packets is detrimental because a large amount of bandwidth is wasted in transmitting header information rather than actual data. For illustration purposes, let us assume that our host is connected to an Ethernet link, so all of our TCP packets and IP datagrams are in turn sent in Ethernet frames. Each of these layers require its own encapsulation header carrying necessary information to deliver the data to its destination. A TCP header occupies at least 20 bytes, an IP header takes up at least another 20 bytes and an Ethernet header is 14 bytes long plus 4 checksum bytes at the end. So for each TCP packet, we need at least 58 bytes worth of header and checksum information. Hence, if we constantly send small data segments (i.e. less than 60 bytes) we waste approximately half of the network resources transmitting information that is not relevant to applications which are the ones that initiate the communication[5].

ACK packets that do not carry any data are one manifestation of the small TCP packet situation. Empty packets are evidently a waste of network resources. However, ACKs are necessary to inform the sender when the data has been successfully delivered, and they are also essential to the growth of cwnd which helps to maintain an efficient sending rate. Therefore, there is a tradeoff between sending empty ACKs and the indirect effect they have on performance. The *delayed acknowledgment* strategy proposes that, whenever possible, the receiver should only send one ACK for every two data segments received [1]. In other words, whenever the receiver gets a new packet, it should not acknowledge the packet immediately. Instead, it should wait until it receives another data packet and then send one ACK for both packets. This way, the number of empty ACKs is effectively reduced by a factor of two. However,

[5]This is not always the case. There are cases in which sending smaller packets yields better performance. See [16] for a complete analysis

waiting too long to ACK the packets might cause the source host to confuse this delay with a packet loss, causing an unnecessary retransmission.

To solve this problem, the delayed acknowledgment scheme stipulates that the receiver should not wait longer than 500ms before ACKing only one packet even if no other packets are received in this period. According to [16], most implementations wait for 200ms. While the delayed acknowledgment algorithm reduces the number of empty ACKs, some authors have argued this strategy might actually harm the performance of TCP connections. These authors argue that the reduced number of ACKs returned to the sender will limit the growth of the congestion window, thereby delaying the time it takes to achieve the ideal sending rate [7]. Other authors, on the other hand claim that the delayed acknowledgment strategy actually enhances the performance of transactions under common network congestion because the startup epoch of TCP transactions is only a transient state.

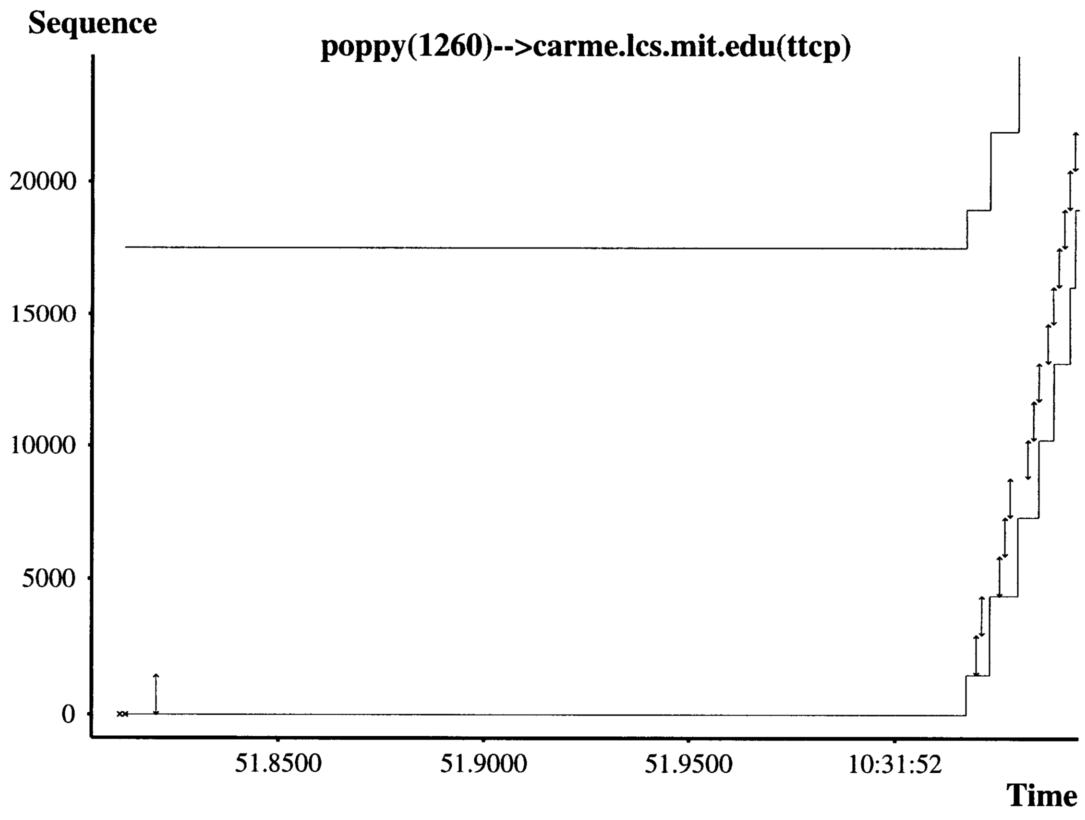

Figure 4-3: Connection establishment and slow start episode with Delayed ACKs.

Figure 4-3 shows the delayed acknowledgment scheme in practice. Note that all the steps in the acknowledgment line correspond to two packets. The only exception is the very first data packet sent in the connection. In the beginning, cwnd is set to one packet so the sender will only send one packet and will not not send any new data until this packet has been ACKed. Hence, the receiver will not get a second packet, so it just sends an ACK for a single paket after 200ms. Afterwards, however, the receiver is able to delay the ACKs and it sends one single ACK for two data packets.

4.3.3 Responding to packet losses efficiently

In section 4.3.1 we discussed the behavior of TCP when it detects a packet loss, but we did not discuss how packet losses are detected. The simplest way to detect losses is by setting a timer that goes off after allowing enough time for a packet to reach its destination and for its ACK to return to the sender (i.e. the timer should be longer than one RTT). If the timer expires and no ACK has been received, TCP concludes that the packet must have been lost. We call this timer the retransmission timeout (RTO). While this scheme provides a reasonable way of identifying packet losses under most realistic situations, it is not optimal from a performance perspective. In this section we present a more efficient mechanism to detect packet losses. Furthermore, we also present a more efficient way to handle packet losses than the one presented in section 4.3.1.

Whenever the receiver gets a packet with a sequence number other than the one it expects to receive, it immediately sends a new ACK containing the sequence number it expects. This ACK is known as a *duplicate acknowledgment* since the receiver must have already sent another ACK with the same sequence number. For example, if a receiver gets all the data up to and including sequence number 24, it is obviously expecting to receive a data packet containing sequence number 25. If instead it receives a packet containing sequence number 50, the receiver sends a new ACK with a sequence of 25. Notice that sequence 25 must have previously been ACKed whenever the packet containing sequence 24 first arrived. Therefore, unless one of the ACKs is lost in transit, the sender will (eventually) see two ACKs with a sequence of

25; the second is called a duplicate ACK.

This strategy serves to notify the sender that the order of packet arrival was not as expected by the receiver. However, the reason for this slightly awkward arrival sequence could be the result of packets being reordered and not necessarily due to a packet loss. In the example above, the packet containing sequence 25 through 49 may arrive right after the duplicate ACK is sent, so retransmitting the packet after receiving only one duplicate ACK is not an optimal approach. However, the reception of many duplicate ACKs would strongly suggest that the packet was actually lost. According to the TCP specification, receiving three duplicate ACKs is indicative that the packet was lost and that it should be retransmitted. Since TCP assumes that packet losses are the result of congestion, ssthresh is set to one-half the value of cwnd as discussed in section 4.3.1. This mechanism is known as the *fast retransmit* algorithm because it does not wait for the RTO to expire.

Notice that the reception of duplicate ACKs also indicates that other packets are successfully reaching the receiver and triggering these duplicate messages. Hence, we know that the network is not critically congested since it is still able to process a good portion of the current sending rate. Going into slow start mode at this point is not necessary and would considerably harm the performance of the transaction. Instead, we use a mechanism known as the *fast recovery algorithm* that allows the sender to continue transmitting new data at a slower rate even if the missing packet has not been acknowledged. The algorithm is as follows: After receiving three consecutive duplicate ACKs, the sender retransmits the packet, sets the value of ssthresh to one-half the value of cwnd and sets cwnd to ssthresh plus three packets. For each new duplicate ACK received, it increases cwnd by one packet and sends a new packet as long cwnd and the window size allows it. Finally, whenever the ACK for the retransmitted packet arrives, we set cwnd to the value of ssthresh and continue operating normally.

Figure 4-4 shows a fast retransmission event followed by fast recovery. Note that in our graphical representation, duplicate ACKs are denoted by downward ticks that "hang" from the acknowledgment sequence line. After receiving 3 duplicate ACKs right after time 11:34:28, the sender retransmits the packet and goes into fast recovery. From the picture, we can tell that the sender has entered the fast recovery phase because new packets are only released after receiving duplicate ACKs. In this particular example, the retransmitted packet was itself lost, so it is retransmitted once again after the RTO expires right at time 29.5000. Note that after all the outstanding data is ACKed (i.e. after the large step in the acknowledgment line) the connection goes back to normal operation mode. Since the last retransmission was due to an RTO timeout, the sender goes into the slow start phase.

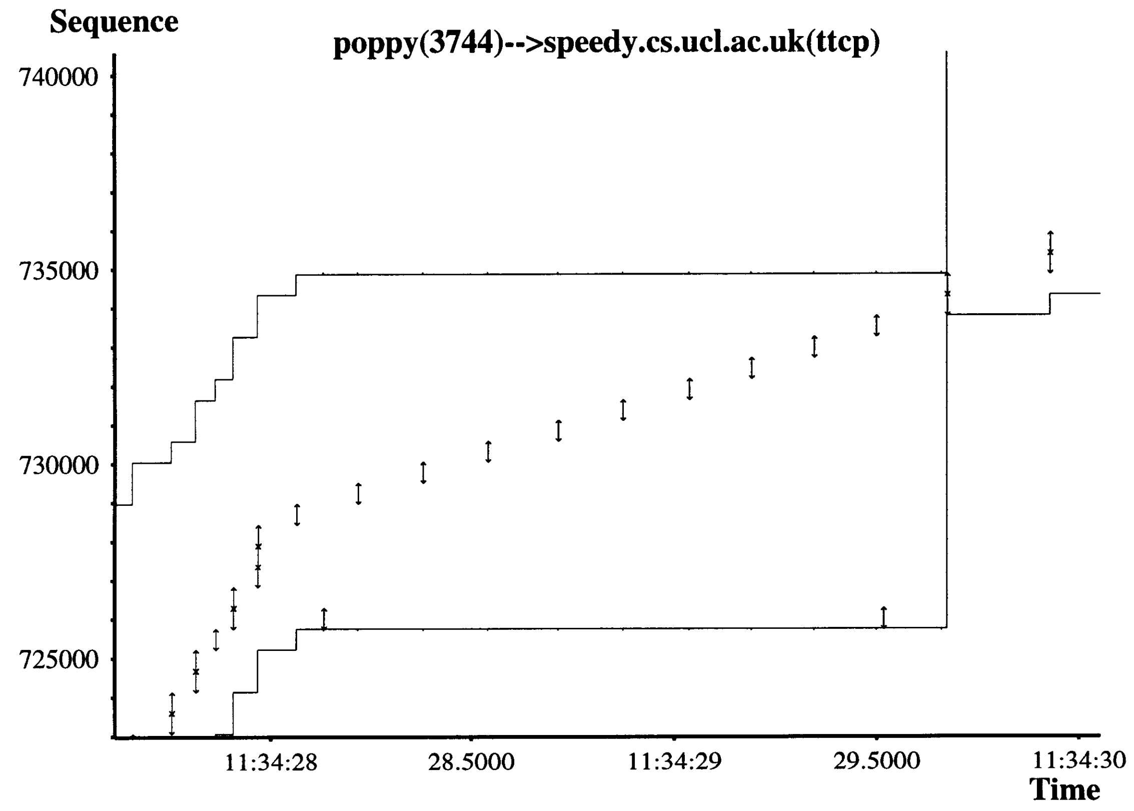

Figure 4-4: Fast retransmission and fast recovery algorithms

Both the fast retransmit and fast recovery algorithms allow the sender to recover from single packet losses more efficiently than the more conservative retransmission timer approach. However, it should be pointed that this scheme only works if there are enough outstanding packets that successfully reach the destination and trigger the transmission of duplicate ACKs. If the network drops too many of the outstanding packets, or if the number of outstanding packets is too small, these algorithms will not be activated in which case the sender will need to wait for the RTO timer to expire. For an exhaustive mathematical analysis of the dynamics of these mechanisms see [18]. For proposed solutions to the duplicate acknowledgment starvation situation see [7].

Chapter 5

Evaluation of TCP implementations

Now that we have described our approach to analyzing TCP implementations and discussed our graphic representation of network traffic, we can finally engage in the task of documenting the behavior of all the implementations we observed. The format of this chapter is as follows: Each section introduces a situation in which at least one of the implementations studied exhibits a peculiar behavior that affects the performance of TCP. We first present those cases that violate TCP specifications, and then we present cases which do not necessarily violate any specifications but still affect the performance of transactions. Each case includes a graphic instance of the problem as it occurs in a real TCP connection. Besides serving to demonstrate that the situation actually arises in practice, the graphic instance also serves to simplify the discussion of how and when the situation arises.

Note that not all of the cases we document in this chapter are "bugs" or violations of the TCP specification. Rather than only presenting incorrect behavior in TCP, we chose to present all implementation specific behavior that might affect the performance of TCP transactions even if the behavior is not intrinsically erroneous. The discussions in this chapter assume that the reader has a good understanding of TCP, especially its congestion avoidance mechanisms.

5.1 Incorrect TCP behavior

The following sections present situations in which the TCP implementations we evaluated fail to comply with TCP standards.

5.1.1 Violating the receiver's window during fast recovery

This pathology was observed only in the TCP implementation of Windows NT 4.0. Windows NT attempts to implements the fast retransmission and fast recovery algorithms. However, this implementation allows the sender to transmit data that is clearly above the advertised window limit of the receiver. This error can be easily appreciated in our graphic representation because it clearly delimits the advertised window at every point of the transaction. In figure 5-1 we can observe an instance of this erroneous behavior. After the lost packet is retransmitted, note that the two packets sent right after time 17.9000 clearly lie above the receiver's window limit.

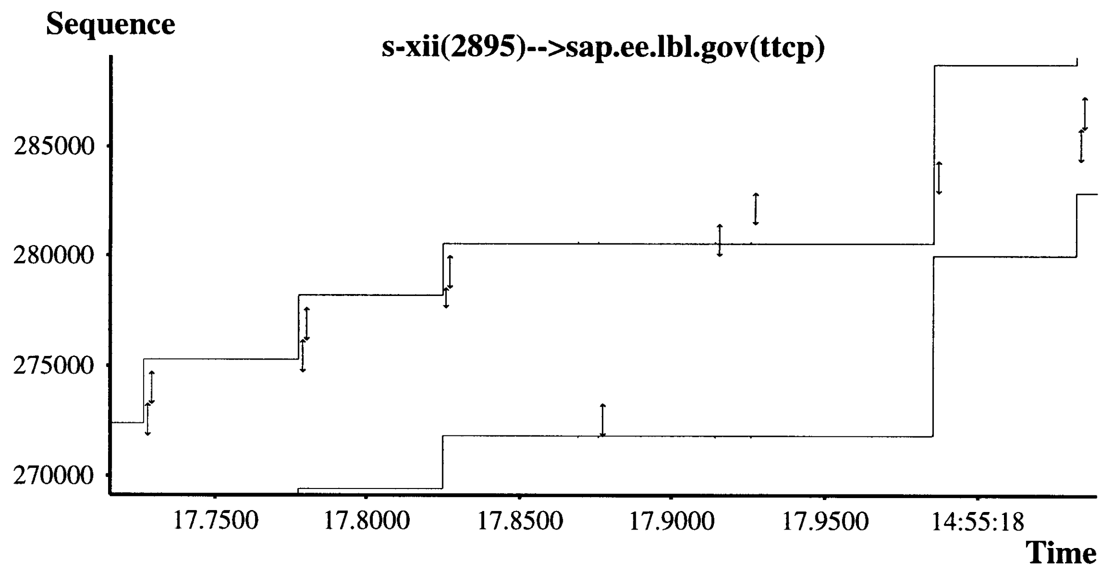

Figure 5-1: Windows NT sending over the window limit during fast recovery.

The difficulty in observing this pathology is that it is more likely to appear if the window advertised by the receiver is small. When the advertised window is large, the receiver is more likely to ACK the retransmitted packet before the sender has a chance to get fill the window. Also, this behavior is not always exhibited by Windows NT. We observed at least one example where the sender had the opportunity of going over the receiver's window limit during fast recovery and correctly refrained from doing so. This example is shown in figure 5-2. Notice all the vertical ticks representing duplicate ACKs; at each of these times an extra packet could have been sent if we ignored the window limit, but the Windows NT implementations does not send any new packets.

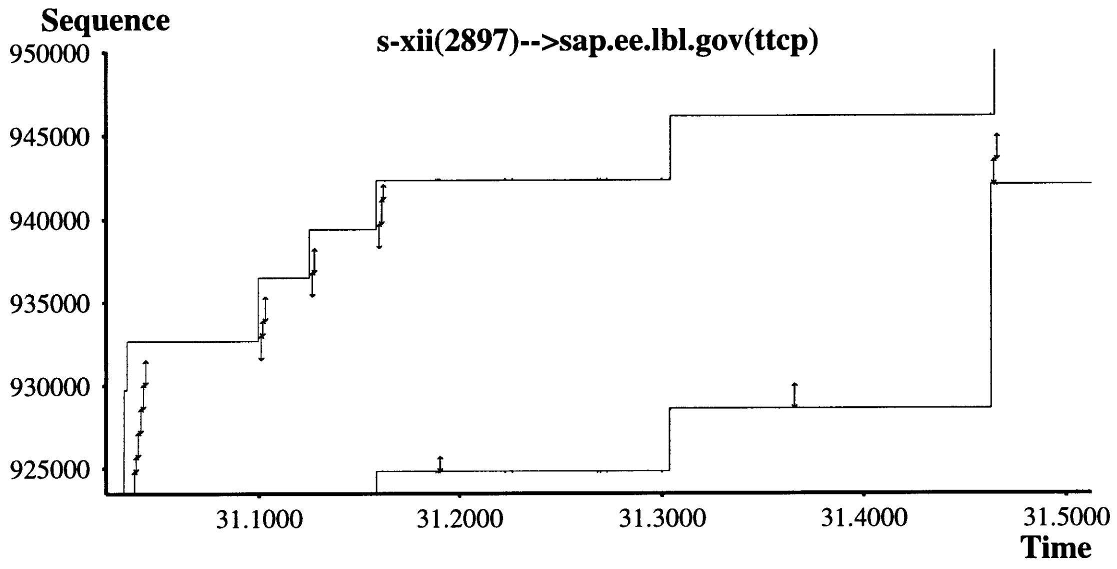

Figure 5-2: Windows NT not going over window limit when it had the chance.

While we did not have access to the source code of this implementation, we imagine that this behavior is the result of either not properly calculating the window limit during the fast recovery phase, or the result of bypassing the tcp_output function. The tcp_output function, as described in [20], checks for both the value of cwnd and the value of the advertised window to see if another packet can be sent. If the implementation bypasses this function during fast recovery (in order to speed up the transmission in this phase) it might be possible to miss the check for the advertised window. We believe that the case when the sender actually behaved correctly by not sending data over the receiver's window limit has something to do with the fact that the window was completely full before the fast retransmission occurred.

Violating the receiver's window limit can be detrimental to the performance of transactions because the sender is transmitting packets to the network that the receiver potentially might not be able to handle. If the receiver does not have space for these packets, the implementation will need to retransmit all the packets that were sent over the limit, maybe even causing the sender to go into slow start if the RTO timer expires. In almost all the cases we observed, there were at most three packets sent over the window limit and the receiving TCP was actually able to store the additional data. However, the receiver is not required to store data that is sent over its window limit. This behavior is erroneous and it should be considered critical.

5.1.2 Sending new data upon receiving duplicate ACKs

As discussed in section 4.3.3, the sender should never respond to duplicate ACKs by sending new data unless it is in the fast recovery phase and both, the congestion window and the receiver's advertised window, have space for the new data. The Windows NT 4.0 implementation violates this invariant by allowing the sender to respond to duplicate ACKs with new data even before going through fast retransmission. Figure 5-3 shows how NT sends a new packet upon receiving a duplicate ACK, before retransmitting the packet, right before time 59.6000.

The problem with this behavior is that it might be increasing traffic unnecessarily exactly when the network is potentially congested. Recall from our discussion in section 4.3.3 that duplicate ACKs potentially signal the loss of a packet, and packet losses should be interpreted as a sign of congestion. Also, recall that right after packet retransmission, cwnd is set to one-half of its value plus three packets, which reduces the rate of outgoing data. Therefore, increasing cwnd before performing fast retransmission effectively allows the sending rate to grow at a time when the network could very likely be congested. Not only can this behavior seriously impact the performance of transactions, but it can also affect the level of network congestion, and it should be considered critical.

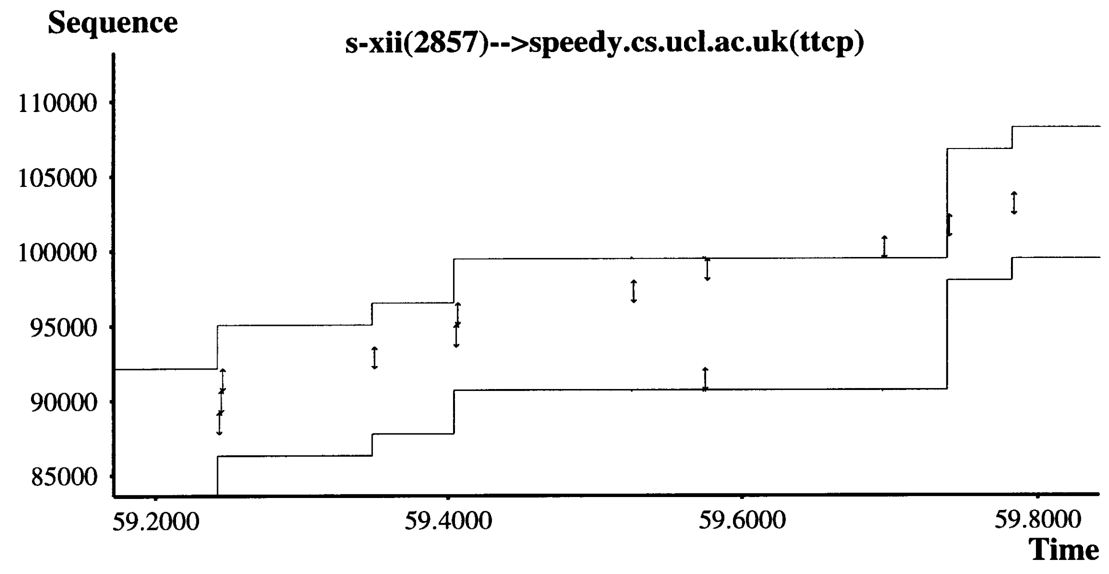

Figure 5-3: Windows NT increasing cwnd after the reception of a duplicate ACK.

5.1.3 Failure to consider window limit when halving cwnd

In section 4.3.3 we discussed that after receiving three duplicate ACKs, the sender should assume that the packet corresponding to the missing sequence was actually lost and consequently, it should retransmit the packet, set ssthresh to one-half the size of cwnd and finally, enter the fast recovery phase. Notice, however, that when setting the new value for ssthresh TCP needs to consider that the value used to compute the "effective" congestion window is actually the smallest value between cwnd and the receiver's advertised window. Recall from section 4.1.1 that the sender should never send more data than allowed by the advertised window of the receiver, and it is therefore impossible to have a congestion window that is larger than the window limit.

Therefore, after a packet loss the sender should set the value of ssthresh to one-half the smallest value between cwnd and the advertised window. During our analysis, we that Windows NT fails to consider cwnd when setting the value of ssthresh after a detecting a packet loss. Using our graphical representation, we can see whether the value of ssthresh was wrongly calculated by noticing how cwnd is updated after the

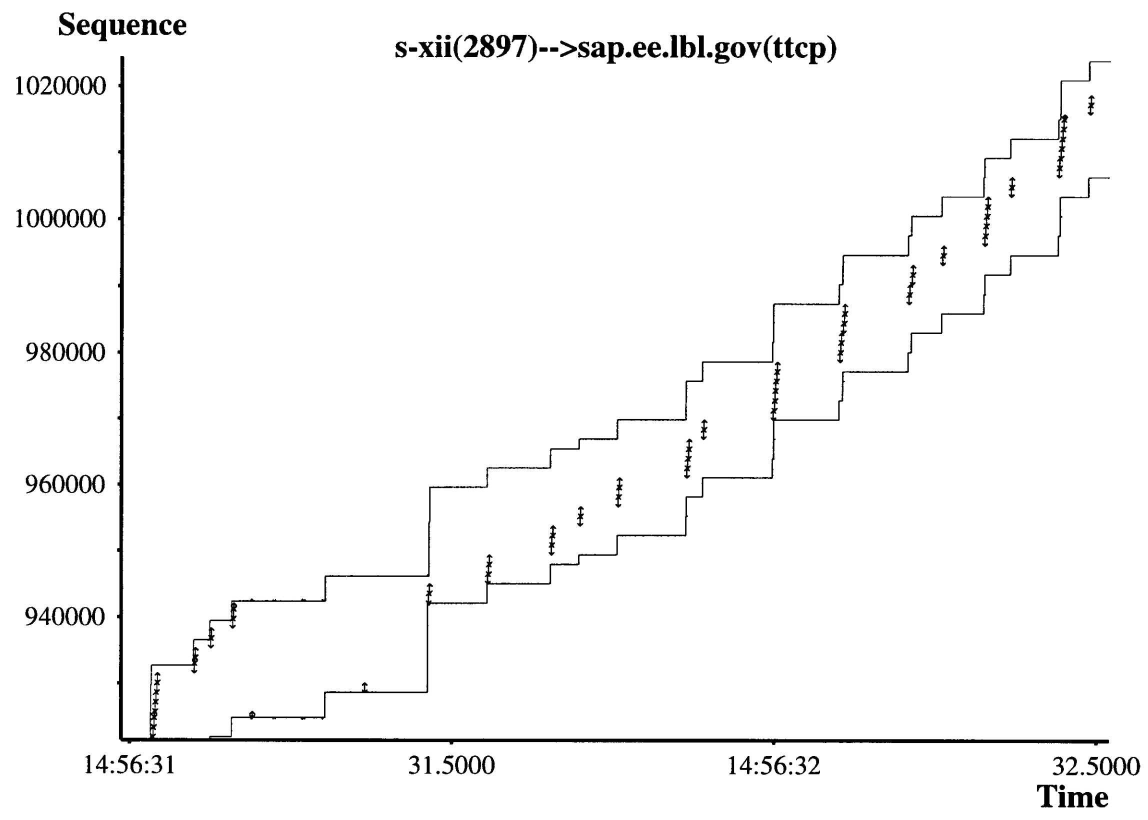

Figure 5-4: Windows NT wrongly halving the sending rate during a fast retransmission.

retransmitted data has been received and ACKed, and the connection is attempting to return to normal operation. Figure 5-4 shows Windows NT exhibiting the pathology presented here. Notice that the number of outstanding packets right before the first retransmission is approximately eleven, but cwnd is potentially larger than that. Since the sender should use the smaller of the two, after the retransmission occurs, ssthresh should be 5 or 6 packets. There is, however a second retransmission after which ssthresh should be 2 or 3 packets. Now, after the retransmitted data is ACKed, denoted by the large step right before time 31.5000, the sender goes into slow start. Notice that when performing the slow start algorithm cwnd goes all the way up to 6 packets before going into linear mode. This suggest that the value of ssthresh is 6 rather than 2 or 3 as expected. It seems that Windows NT, instead of including the advertised window in its calculations for the new value of ssthresh, is using the value of cwnd, which is very likely larger, and ssthresh ends up being too large for the congestion conditions.

The problem with this behavior is that the new value of ssthresh might not decrease the sending rate effectively after a retransmission. Hence, the sender could have a sending rate that is potentially too fast for the congestion levels that exist in the network. This behavior could be detrimental to the TCP transaction itself because it fails to refrain from sending packets during times of congestion which can potentially result in many of its packets being dropped. Moreover, by not correctly avoiding congestion, the host can contribute to maintain the network in a congested state thereby affecting the performance of other connections.

5.1.4 Fast retransmission after two duplicate ACKs

According to the TCP standard, the fast retransmission mechanism should only be entered after the sender has received three duplicate ACKs. The Windows NT implementation, however, performs a fast retransmission after receiving only two. The problem with this approach is that two packets are not enough to tell with certainty that a packet has been actually dropped. The packet could have potentially been reordered and its ACK could be about to arrive. Thus, this approach is too aggressive given that TCP is expected to work under intrinsically unreliable networks.

Clearly, retransmitting the packet before the third duplicate ACK might be unnecessary, and therefore detrimental to performance. Moreover, since after fast retransmissions the sender reduces cwnd by a factor of two, the data rate of the sender will also be unnecessarily reduced. Therefore, misinterpreting the signal of a dropped packet when the packet has only been reordered is very costly. Figure 5-5 shows Windows NT retransmitting a packet after receiving only two duplicate ACKs.

5.1.5 Performing fast retransmissions after a timeout

This is a subtle situation that could potentially arise in any TCP implementation, but it is more likely to be observed in Windows NT because of other bugs in this implementation. Under normal operation, assuming that our estimate of the RTT is

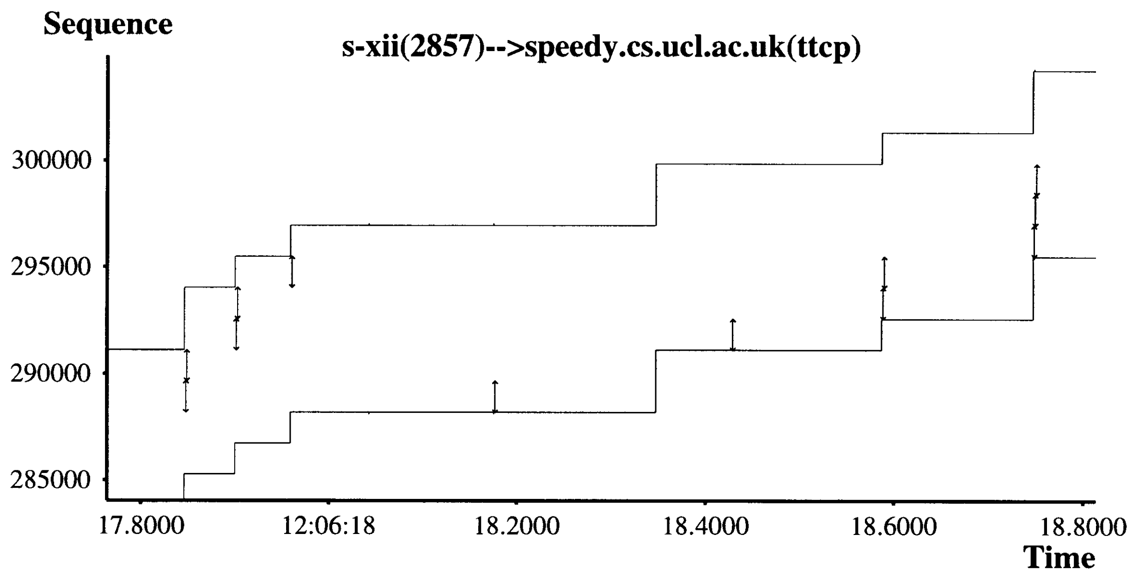

Figure 5-5: Windows NT retransmitting a packet after receiving only two duplicate ACKs.

reasonably accurate, we expect the fast retransmission algorithm to be activated, long before the retransmission timeout (RTO) expires. If there are enough outstanding packets when a packet is dropped, these outstanding packets will trigger the generation of duplicate ACKs which will cause fast retransmission to occur and the RTO will never expire. If there are not enough outstanding packets, then the packet is retransmitted after the RTO expires, and the fast retransmission algorithm is never activated.

There are two "bugs" in the Windows NT implementation that allow this situation to arise. As we'll discuss shortly, entering the fast retransmission phase after receiving only two duplicate ACKs instead of three makes the situation more likely to occur. Secondly, it seems that Windows NT does not reset the duplicate ACK counter after the RTO expires which also propitiates the situation. The last "bug" is not necessarily incorrect as the specification for TCP does not specify that the duplicate ACK counter should be reset after the RTO expires. However, BSD derived implementation actually reset this counter making the situation less likely to occur in these implementations.

Figure 5-6 shows an example of a fast retransmission occurring right after a retransmission timeout. We can see that the first retransmission is due to the expiration of the RTO because there are no duplicate ACKs right before the packet is retransmitted. Note that before the first retransmission, the sender received a duplicate ACK. After the RTO expires it correctly retransmits the packet, but fails to reset the duplicate ACK count. When a new duplicate ACK arrives shortly after the first retransmission, the sender retransmits the packet *again* since it will handle the duplicate ACK as if it were the second it received.

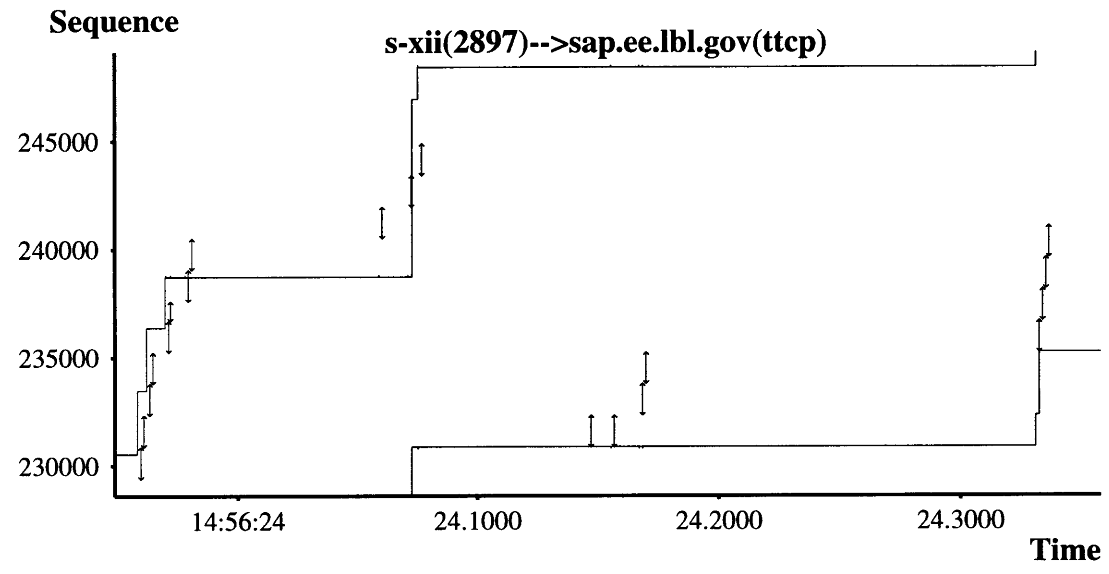

Figure 5-6: Windows NT performing a fast retransmission after a retransmission timeout.

In theory, this situation could occur under any TCP implementation. However, the fact that other implementations do not have the pathologies exhibited by Windows NT makes this situation very unlikely, if not impossible. We can think of a situation in which the first packet in a burst of four is lost, and the other three take an unusually long path to the destination. If the path is so unusually long that the RTT estimator causes a retransmission timeout before the duplicate ACKs for these packets are received by the sender, this situation will arise under any TCP implementation. But all three duplicate ACKs need to arrive *after* the retransmission timeout occurs, which can be avoided with good RTT estimators. This is why the situation is not very likely to arise in other TCP implementations.

This pathology is obviously detrimental to the performance of transactions. The sender retransmits a packet that has been recently retransmitted. Given that packet losses are more likely to occur during times of network congestion, the retransmitted packet will be adding more traffic to the network exactly when it is more dangerous. Since the two retransmissions are likely to occur very close to each other, they are also likely to encounter similar network conditions so the fate of both packets will probably be the same. Therefore, this double retransmission is unnecessary and harmful, and should be avoided by TCP implementations.

It should be noted that the similarly harmful case in which a retransmission timeout occurs shortly after a fast retransmission is not possible according to the TCP specifications. After the packet is retransmitted by the fast retransmission algorithm, a new RTO estimate is associated with the retransmitted packet, so the timeout should only expire if this retransmitted packet is itself lost.

5.1.6 Header prediction error

The header prediction mechanism in TCP is an optimization that speeds the processing of certain kinds of packets that are believed to be very common during transactions. For instance, during one-sided data transfers, will receive many empty ACKs from the receiver, so it would be beneficial for the sender to handle these situations efficiently. If a non-empty ACK is received instead, then the sender can process the packet more carefully, but we expect these situations to be rare during this type of transactions. Similarly, the receiver can optimize its processing speed by efficiently handling the situations where the incoming packet contains the sequence number it expects.

The problem with some TCP implementations is that they fail to consider all the necessary conditions that make a packet "common". In particular, the possibly empty ACK that acknowledges new data after the fast recovery period should be handled differently since cwnd needs to be set to ssthresh right after its reception. Some implementations fail to make this distinction resulting in cwnd having an incorrect value after fast retransmission. Since the values of cwnd would be unusually large, this pathology causes very large bursts of packets to be sent right after fast recovery. Hence, the sender fails to effectively reduce the sending rate after a packet loss episode. This problem is fully documented in [2], where there are also suggestions on how to solve the situation.

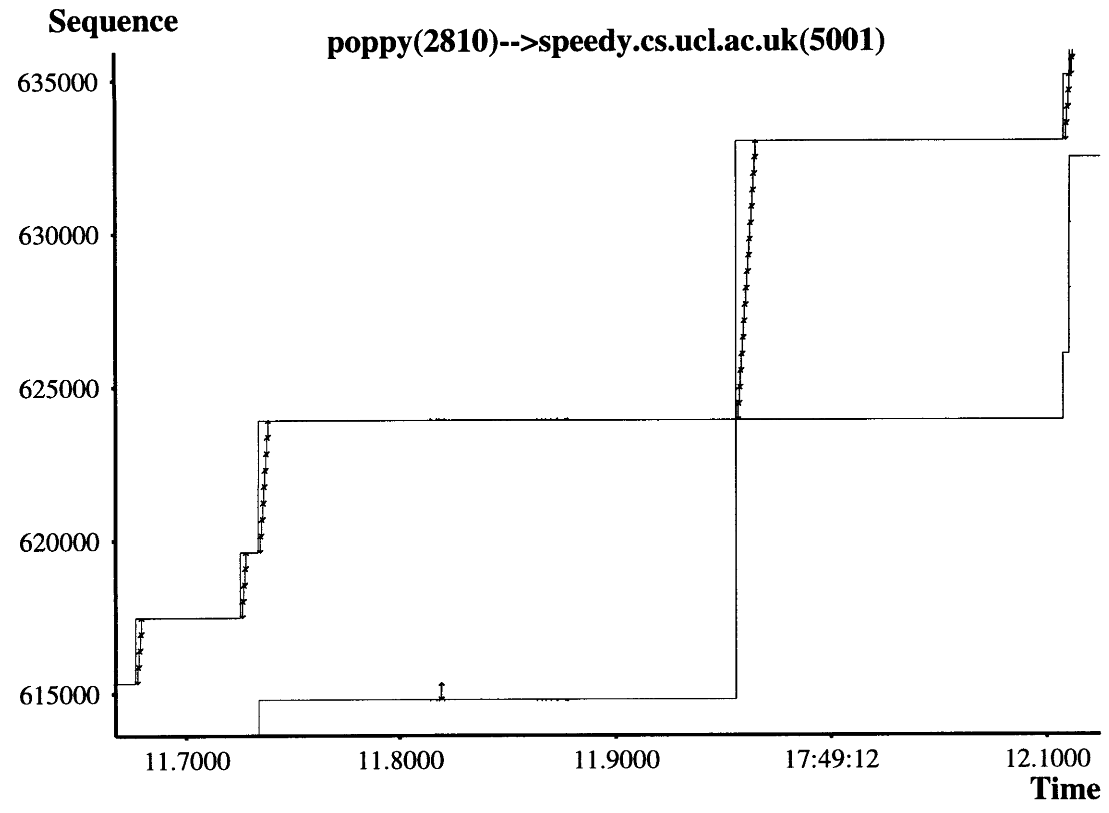

Figure 5-7: DEC OSF 3.2 not decreasing the value of cwnd after fast recovery.

During our evaluation, we found this pathology in two of our implementations: DEC OSF 3.2 and on Windows NT. The error can be consistently observed in the DEC OSF 3.2 implementation, but we only found one instance of its occurrence in Windows NT[1]. Figures 5-7 and 5-8 present instances of the problem occurring in OSF and Windows NT respectively. After the lost packet is retransmitted, the value of ssthresh is set to one-half the value of the advertised window in both cases. Now, when

[1]Curiously, this was also the only instance we found of Windows NT respecting the window limit during fast recovery.

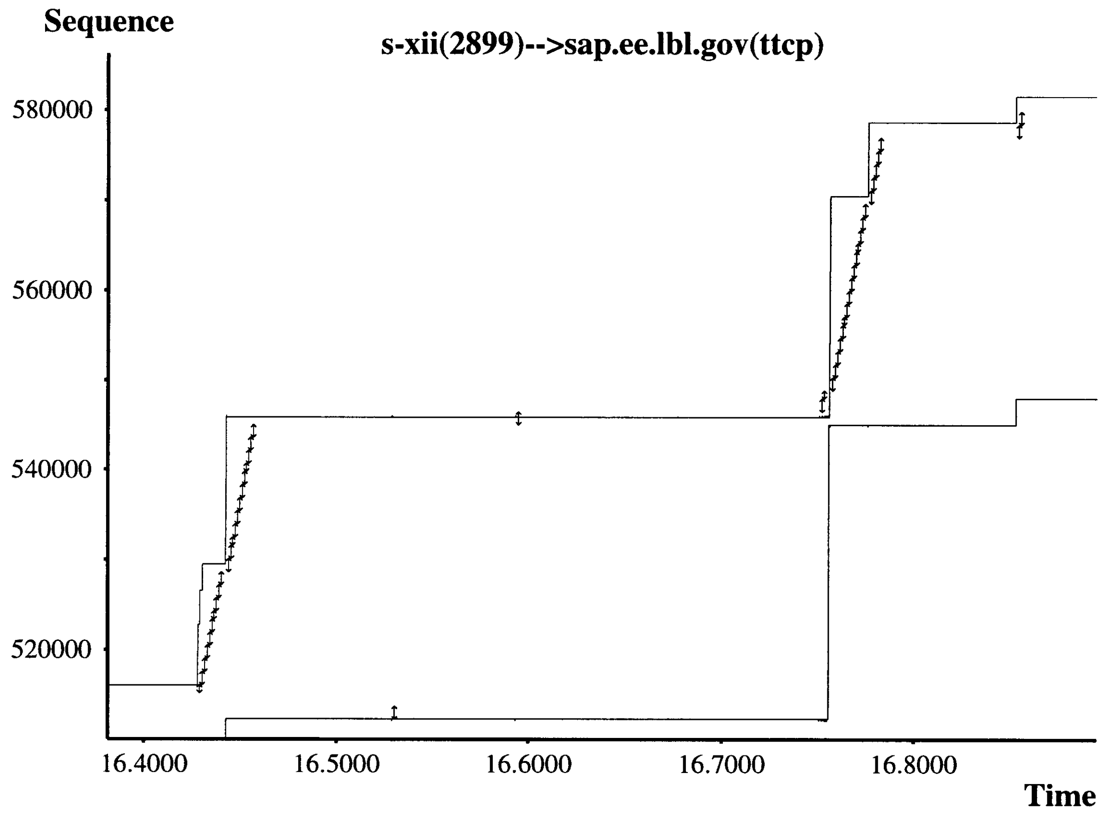

Figure 5-8: Windows NT not decreasing cwnd after fast recovery.

the outstanding data is ACKed (denoted by the large step in the acknowledgment line) cwnd should be set to ssthresh, so we should only see a burst with a size of exactly half the advertised window. However, because the ACK is processed differently, cwnd does not get updated, so the size of the burst is actually twice as large. Note that the problem is only exhibited whenever the value of cwnd is larger than the value of the advertised window. Otherwise, the header prediction code is avoided because TCP infers that the connection could potentially be in the middle of slow start or congestion avoidance, which would require the ACK to be handled differently.

5.1.7 Failure to perform slow start in LANs

When two hosts are in the same network, performing the slow start algorithm might seem highly inefficient since, after all, there are not many routers or switches in between that can become congested, and since the traffic levels are relatively low in comparison to the capacity of local networks. The probability of a packet being dropped in a Local Area Network (LAN) transaction is very small, so waiting for ACKs before sending new data seems unnecessary and harmful to performance.

The TCP implementation of FreeBSD attempts to avoid this situation by not performing slow start if the receiver is in the same network as the sender. Figure 5-9 shows an example of this situation. From the diagram we see that the traffic is handled properly by the receiver and that the optimal sending rate is reached much faster than when performing the slow start algorithm.

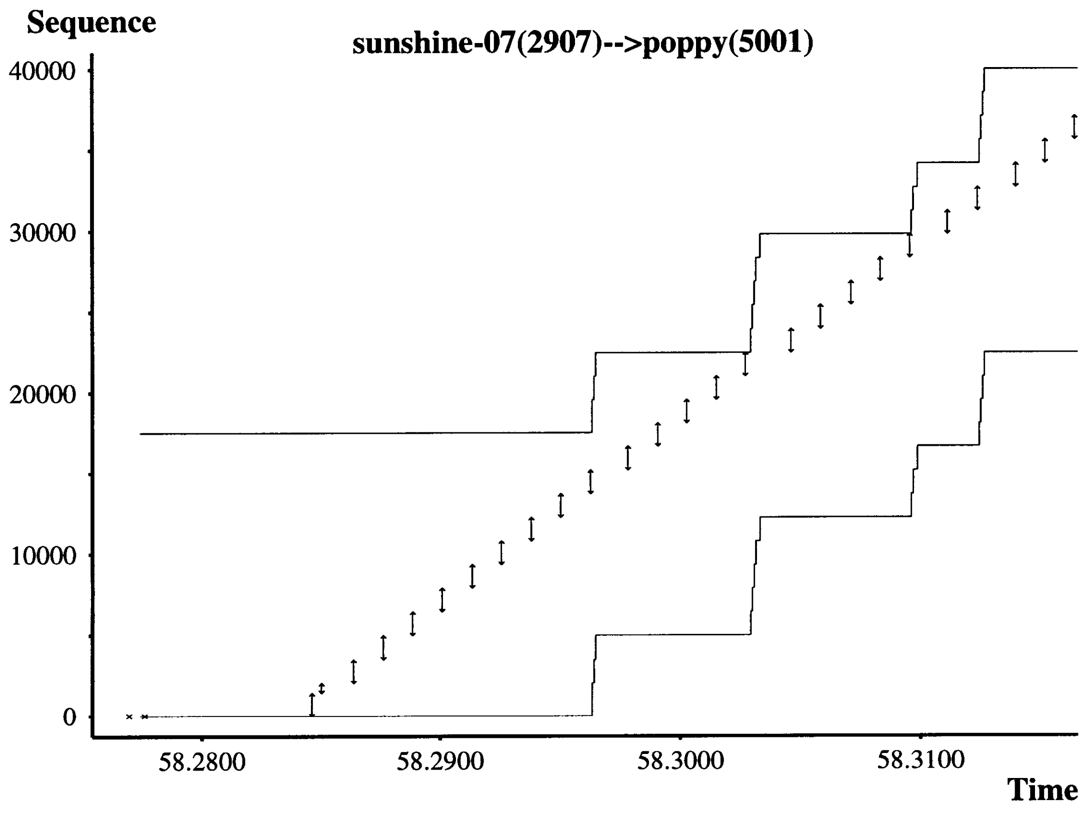

Figure 5-9: FreeBSD failing to perform slow start in a local connection.

While the argument presented above might seem convincing and reasonably acceptable for most LAN architectures, it is not clear that the assumptions made about traffic in local networks are true in all possible cases. We can imagine a LAN that has a very slow link connecting two ends (e.g. a wireless link connecting two remote sites) in which case avoiding slow start might cause disastrous network failures. The argument of whether this scheme can be implemented with confidence and included in the TCP standard would not be easy two resolve. Nevertheless, this behavior is not included in the TCP standard, and since it might result in critical performance hazards, we include it here as a bug in the FreeBSD implementation.

5.2 Performance issues in conforming TCP behavior

In addition to finding the violations of the TCP standards presented in the previous section, we also found a number of cases that are not explicit violations, but that affect the performance of TCP transactions. We discuss these cases in this section.

5.2.1 Retransmission of packets with different sequences

According to TCP standards, when a host decides to retransmit a packet that has been lost, it does not necessarily need to include the same sequence numbers that were included in the original packet that was lost. Presumably, this flexibility was allowed in the TCP standard because it gives the opportunity for implementations to improve the performance of transactions in the cases that many small packets are lost. In these cases, the sender can just retransmit one large packet containing the data originally distributed among the smaller packets, thereby reducing the time for all the packets to be ACKed. On the other hand, this flexibility also allows TCP implementations to harm the performance of transactions by choosing to retransmit a packet that contains less data than it was originally sent. During our evaluation, we observed a number of instances where Windows NT and Windows 95 retransmitted packets in this fashion. These instances were observed during both fast retransmissions and during retransmission timeouts.

Figure 5-10 shows an example of this situation occurring in a Windows NT transaction. Notice, that in the first retransmission, the packet is much smaller than the packet originally lost. Besides wasting resources in sending a small packet, when the

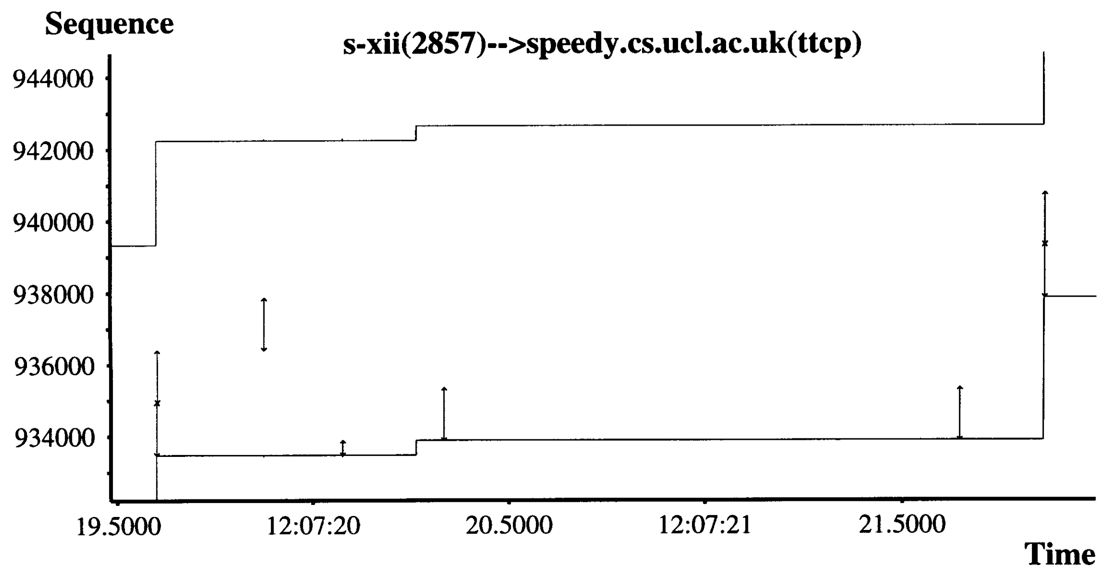

Figure 5-10: Windows NT retransmitting a packet with a different size than the original.

ACK for this small packet arrives, the sender transmits a full-size packet that contains data from the originally lost packet, and also from the next packet. Therefore, the sender ends up retransmitting data that was not lost to begin with. To make matters worse, the second retransmitted packet was lost, so the unnecessary data was sent twice. Clearly, this behavior is not optimal because it wastes network resources in transmitting small packets, and data that does not need to be retransmitted.

5.2.2 Failure to perform fast retransmission

Early versions of TCP did not include the fast retransmission and fast recovery algorithms. Therefore, TCP implementations that were developed based on these early versions might not implement congestion avoidance. Even if these algorithms have been included in recent versions of TCP, older implementations can not be considered incorrect for not including them. During our studies, we found that the Windows 95 implementation does not provide the fast retransmission and fast recovery algorithms.

Figure 5-11 shows Windows 95 failing to perform fast retransmission after receiving more than three duplicate ACKs. While this is not incorrect, we can see from

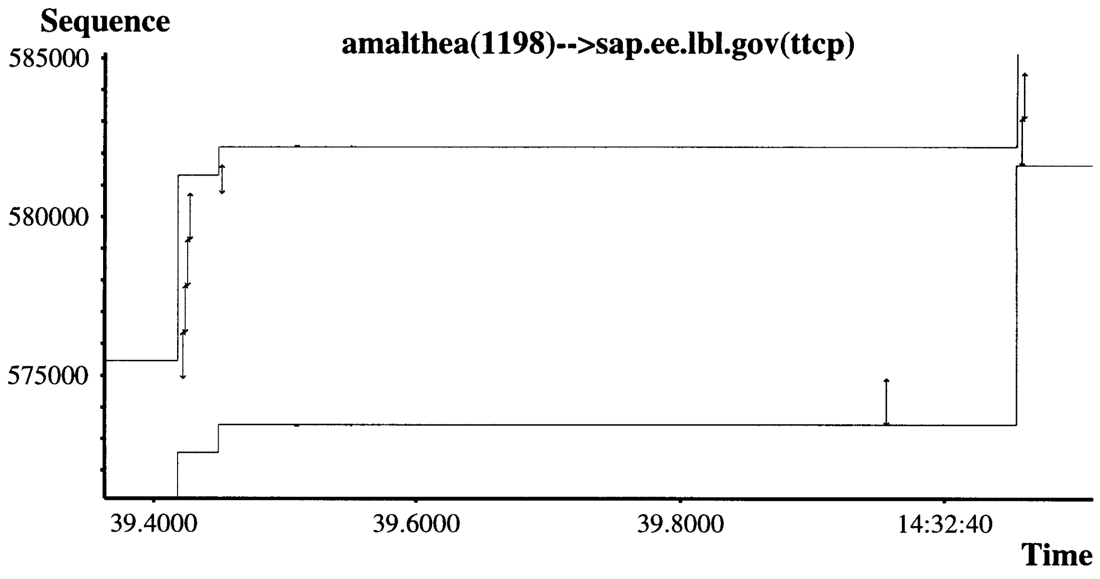

Figure 5-11: Windows 95 failing to perform fast retransmission.

the figure that not implementing these algorithm can considerably harm the performance of transactions. Notice the long period of "silence" between the moment when the packet is first transmitted until the RTO timer expires and the packet is finally retransmitted. Furthermore, after the RTO timer expires, TCP goes into slow start mode which harms the performance of the transaction even further.

5.2.3 Stretch acknowledgments

In section 4.3.2 we saw that upon receiving data packets, a TCP host can wait for some period of time before acknowledging the data, as long as it does not wait longer than 500ms. The delayed acknowledgment strategy presented there proposed sending a single ACK for every two data packets received. In general, however, TCP hosts can choose to ACK more than two packets worth of data with a single ACK as long as the ACK is not delayed by more than 500ms. The problem with sending these "large" or *stretch acknowledgments* is that it can limit the growth of cwnd and it can cause the traffic to be extremely bursty.

In section 4.3.1 we saw that cwnd increases by one packet for every ACK the sender receives. If the receiver reduces the number of ACKs by sending stretch acknowledgments, cwnd will grow much slower which will result in TCP transactions taking longer to find the ideal sending rate. Additionally, the arrival of large ACKs will cause the sender to respond by transmitting a burst containing at least as many packets that were acknowledged by the ACK. Figure 5-12 shows an example of this situation. For every packet sent, the receiver responds with packets that acknowledge more than two packets. Notice that the total number of unacknowledged data increases by one packet per RTT, which is linear rather than exponential. Also, notice the burstiness of the outgoing traffic which directly results from the reception of large ACKs. Clearly, stretch ACKs can harm the performance of TCP transactions.

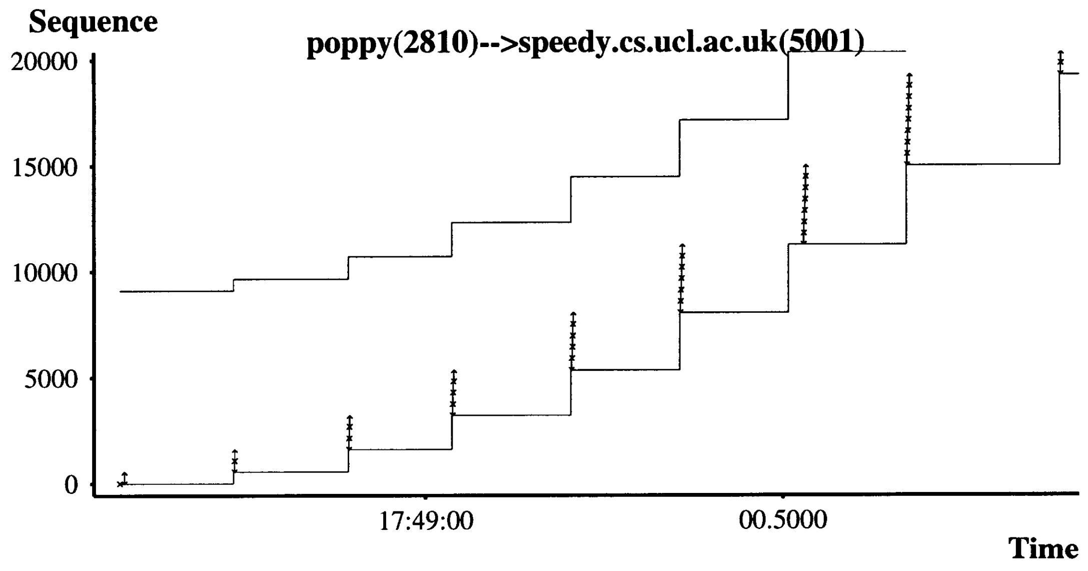

Figure 5-12: Receiver sending large or stretch ACKs.

Chapter 6

TCP Traffic in Real Networks

TCP is a transport layer protocol. In principle, TCP sees the network as an abstraction that provides the means to communicate with other hosts, and ignores the details of how the network operates in order to provide its services. Nonetheless, the operation strategies followed by the network can dramatically affect the performance of TCP traffic. The fundamental problem with this dependency is that the performance of the network layer is measured from a different perspective than the performance of TCP transactions, so finding an optimizing both simultaneously is not trivial. For instance, optimizing network performance might imply minimizing the number of nodes visited when delivering data between two hosts, or avoiding congested links by distributing the traffic among multiple links. These strategies, however, might have negative effects on the performance of TCP transactions. In this chapter we show a few examples of particular TCP behavior that arises as a result of specific traffic patterns or characteristics exhibited by the network layer. We should point out that the situations presented here are not examples of misbehavior by TCP implementations or by the network layer, but are only instances of interesting interactions between the two that result from conflicting performance metrics used in both layers [1].

[1]For a better assessment of Internet performance metrics and their effect on TCP traffic, see [12].

6.1 ACK Compression

During TCP transactions, the sender only transmits new data after receiving ACKs from the target host. This behavior creates a natural "timing" between the sender and the receiver that depends on the capacity of the smallest link in the path between the two hosts. To see this, observe that when a flow has reached a "steady state", the sending rate of the source will depend on how fast ACKs are arriving from the receiver, which in turn depends on how fast data packets make it from the sender to the receiver. Intuitively, the time it takes data packets of a given size to reach their destination will depend on the capacity of the smallest link in the path taken. Empty ACKs, on the other hand, are so small that we do not expect the bottleneck link to significantly affect the time it takes them to travel back to the sender. Therefore, the time elapsed between the arrival of two consecutive ACKs represents the time it takes the network to deliver one data packet from the sender to the receiver. This synchronization between the two ends of a connection is known as the *self-clocking mechanism* of TCP[2]. Maintaining this timing is beneficial for the performance of transactions because data packets are sent at exactly the rate that the network can handle.

There are, however, a number of methods and gadgets used by networks that can disrupt the timing created by this self-clocking mechanism. Particularly, the existence of packet queues in network devices has the effect of increasing the time it takes the network to deliver packets because of the additional delay introduced by these queues. Ultimately, this additional delay will affect the timing between the two ends. If the delayed packets contain any data (i.e. they are not empty ACKs), this behavior will cause the receiver to send its ACKs as fast as the network is handling the data packets, so there will only be a delay effect in the traffic, but the self clocking mechanism will still work as expected. On the other hand, if the delayed packets are empty ACKs, queuing will alter the time intervals between ACKs which are essential to maintain a healthy sending rate as explained above. As a result, empty ACKs will

[2]For a better explanation of the self-clocking mechanism see [8].

arrive almost simultaneously to the sender, which will cause the sender to respond with new data packets that are very close to each other, effectively causing bursts of packets instead of traffic that is carefully timed according to network capacity. This phenomenon is called ACK compression and it is described by in [5].

Traffic burstiness is generally undesirable because the demand for network resources does not remain constant over time. A more efficient approach would be to distribute the demand of resources evenly so as to maintain the utilization level of the network as constant as possible. Figure 6-1 shows an example of this situation. Note how all the ACKs arrive approximately simultaneously even when the packets were sent with time spacing in between. Note that the new data packets that are sent as a result of the ACKs are also very close to each other resulting in bursty traffic.

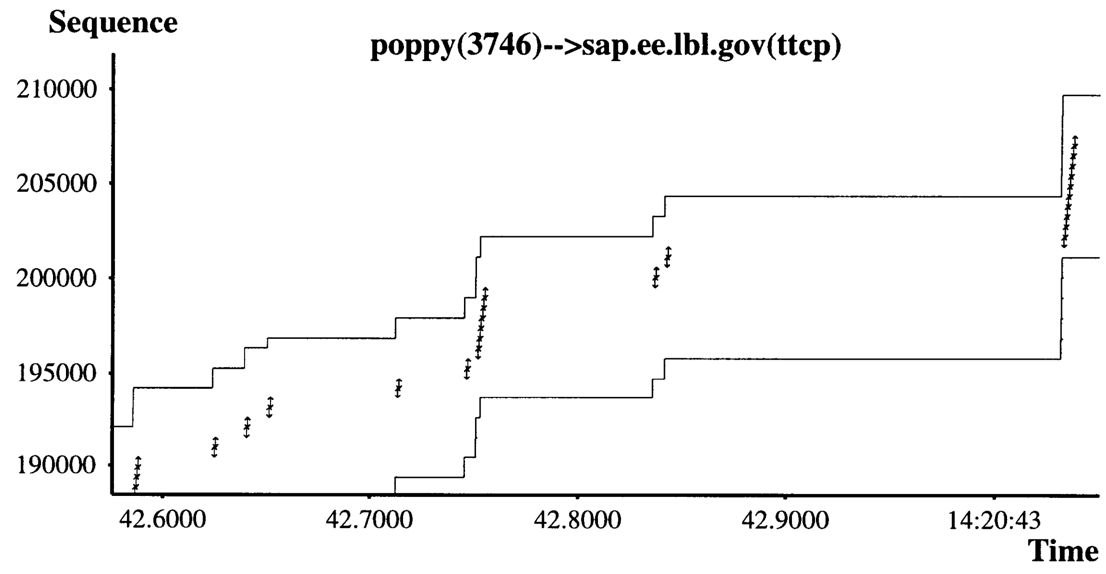

Figure 6-1: ACKs "compressed" in time due to queuing effects.

It should be noticed that the existence of queues is not the only factor that alters the self clocking behavior of TCP. If the receiving host fails to send ACKs right after data packets arrive, the timing between empty ACKs will not represent the interarrival time of data packets, thereby affecting the timing in a similar manner. Figure 6-2 shows an example of this situation. Our traffic tracing tool is located on the same network as the receiver, so we specifically observe the behavior of the receiving host. Notice that instead of ACKing the data immediately, the receiver waits until an entire burst arrives before it starts sending the corresponding ACKs. As a result, the ACKs are sent in a large burst which will eventually cause the incoming traffic to be bursty. Hence, we see the network is not the only entity that can break the self-clocking mechanism of TCP.

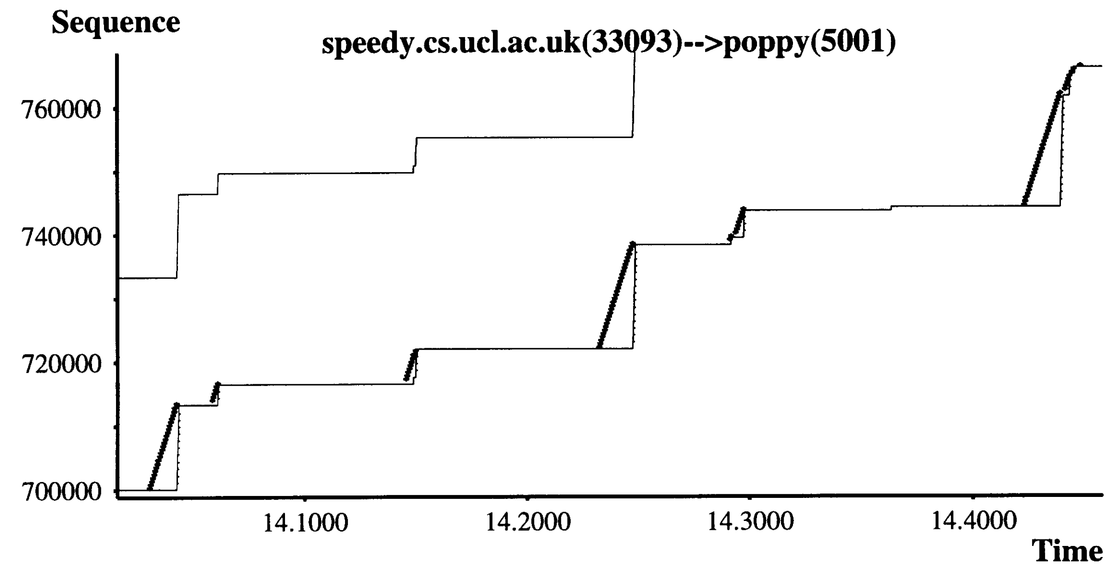

Figure 6-2: Receiver sending ACKs almost simultaneously resulting in compression.

But even if the performance of the TCP transaction is affected in these situations, it is incorrect to conclude that the network is providing a suboptimal service, or that the receiver is misbehaving according to TCP standards. The network only agrees to provide a best-effort traffic delivery service that makes no guarantees in terms of timing. In the case of the TCP receiver, we need to consider that the TCP processes that run on hosts are subject to scheduling by the operating system and other similar factors. Therefore, it is not feasible to impose requirements on the time it takes a host to respond to TCP events because these parameters can not be controlled by TCP processes.

6.2 Unnecessary retransmissions

While TCP ignores the details of how networks operate, it does make some assumptions about the operation of the network layer in order to improve the efficiency of transactions. For example, the fast retransmission mechanism discussed in section 4.3.3 knows that the network can potentially reorder packets, but at the same time it assumes that the level of reordering that can occur in practice is not large. In particular, this mechanism assumes that the arrival of three duplicate ACKs signals that a packet was not merely reordered, but actually lost. Similarly, the round trip time (RTT) estimate is assumed to be sufficiently larger than the actual RTT, so that if the RTO expires the sender can reasonable conclude that the packet has been lost. The network, however, could potentially fail to comply with either of these assumptions at any time, and this might harm the performance of TCP transactions.

In order to improve the efficiency of the packet delivery service provided by the network layer, the traffic might sometimes be distributed among several links in an attempt to balance the workload. This strategy, known as *load balancing*, may cause packets to be reordered if the links used to distribute the traffic exhibit different characteristics. For instance, if one of the links is more congested than the rest, the packets that go through this link will arrive at the destination much later than those sent through less congested links. While load balancing can improve the performance of the network layer, the side effect of packet reordering can considerably harm the performance of TCP transactions. The reason is that if the level of packet reordering is large, many duplicate ACKs can be generated and TCP might confuse them with signals of a packet drop, which will result in the unnecessary retransmission of data packets.

Figure 6-3 shows an instance of exactly this type of behavior triggered by a link doing load balancing among its links. Note that there is relatively large number of duplicate ACKs that arrive at the sender causing the retransmission of the packet after time 53.3500. Interestingly, the ACK for the packet that was thought to be lost arrives shortly after the retransmission occur. We know that the ACK does not correspond to the retransmitted packet because the time between the retransmission and the arrival of the packet is too small. Hence, the duplicate ACKs seen right before the retransmission must have been due to the reception of packets with higher sequence numbers, and we can conclude that the packet was not really lost but only reordered. Using **pathchar**, a tool that analyzes the path between in the Internet, we discovered that indeed there is a link that was using load balancing at the time of the transaction[3]. Therefore, while load balancing can help to improve the efficiency of network traffic in general, we see that it can actually have negative effect on TCP traffic.

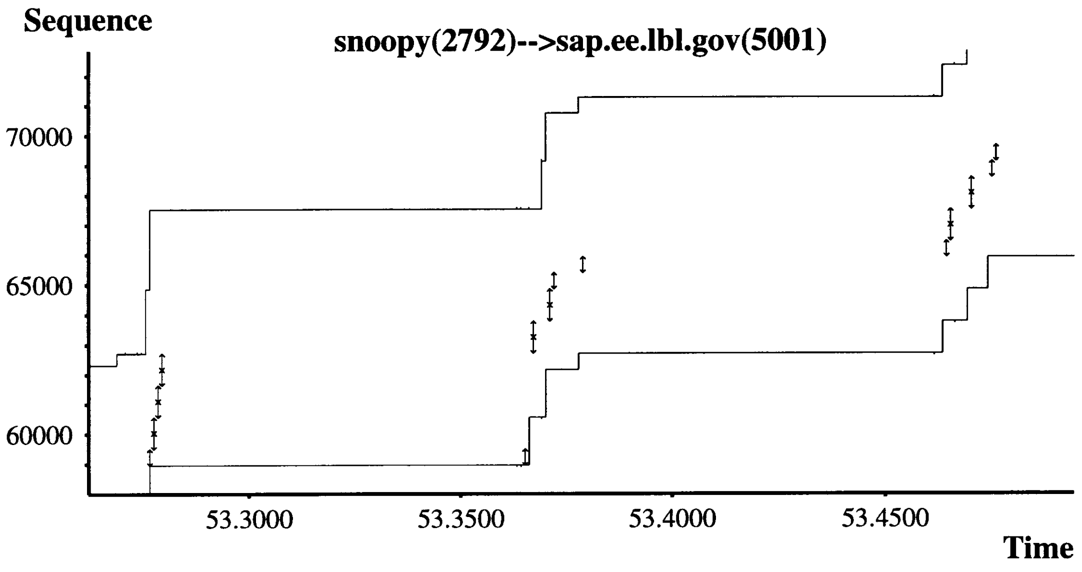

Figure 6-3: Unnecessary retransmission of packet due to load balancing.

Another situation in which we observed data packets being unnecessarily retransmitted, was due to a TCP mechanism. In section 4.3.2 we saw how the receiving TCP does not ACK packets as soon as they arrive, but it actually waits for another data packet to arrive in an attempt to minimize the number of empty ACKs sent. However, if the receiver waits for a long time before giving up and sending a single empty ACK, the sender might interpret the delay as a packet loss and retransmit the

[3]For further information on patchar, see [9].

packet unnecessarily. Figure 6-4 shows an example of this situation. Note that we are monitoring traffic at the receiver side. The receiver takes a relatively long amount of time to ACK the packet sent right before time 29.400, which causes the sender to send a packet retransmission which arrives after time 10:53:30. Clearly, the RTO timer of the sender must have expired while it was waiting for the ACK. Note that this behavior could be the result of the receiver waiting for too long, or the sender's timer being too short (i.e. a bad estimation of RTT).

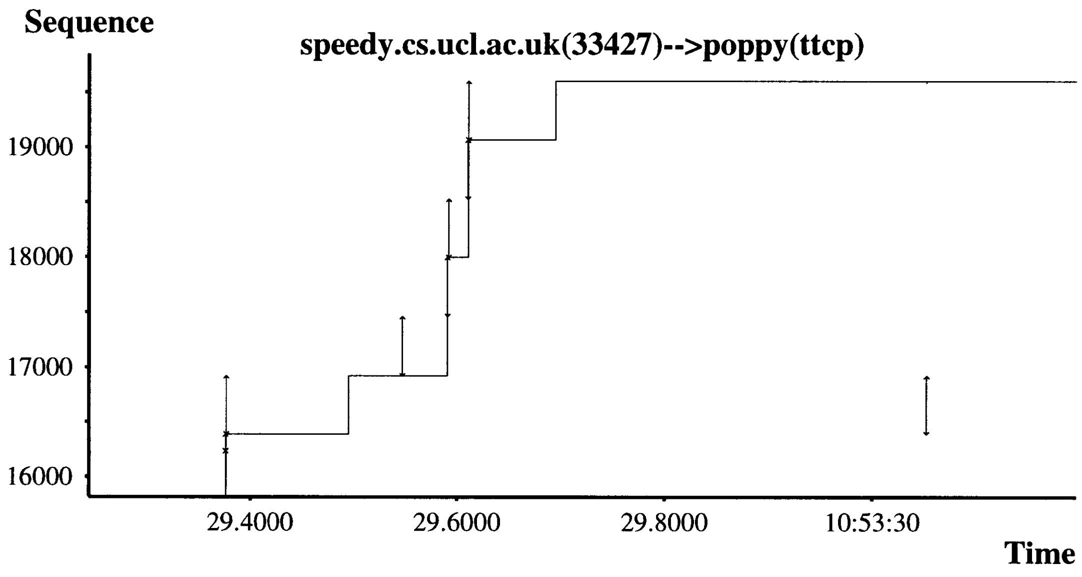

Figure 6-4: Unnecessary retransmission caused by a delayed ACKs.

6.3 Variability of round trip times

Figure 6-5 shows a TCP connection for which the RTT clearly changes over a short period of time. This type of changes can be caused by two possible situations: First, the network might have suddenly become more congested so it takes longer for the network to deliver packets, thereby increasing the RTT. The source of this congestion could be aggregate traffic from a myriad of transactions, or it could be the result of the connection itself increasing its sending rate and filling up buffers in network devices. The second possibility is that the network might have changed the path through which it used deliver packets for this connection to a new path with a longer RTT. A change in paths could be due to a link failure that makes it necessary to switch to a different path, or because a new path is expected to provide a better packet delivery service. In the case shown in figure 6-5, the congestion explanation seems more reasonable, since changing paths would imply either a temporary loss of connectivity (packets dropped) or hopefully an improvement in the delivery service rather than a decrease in RTT. The interesting point, however, is to notice that the network clearly makes no guarantees regarding the efficiency with which packets are delivered, and TCP has to cope with these changes.

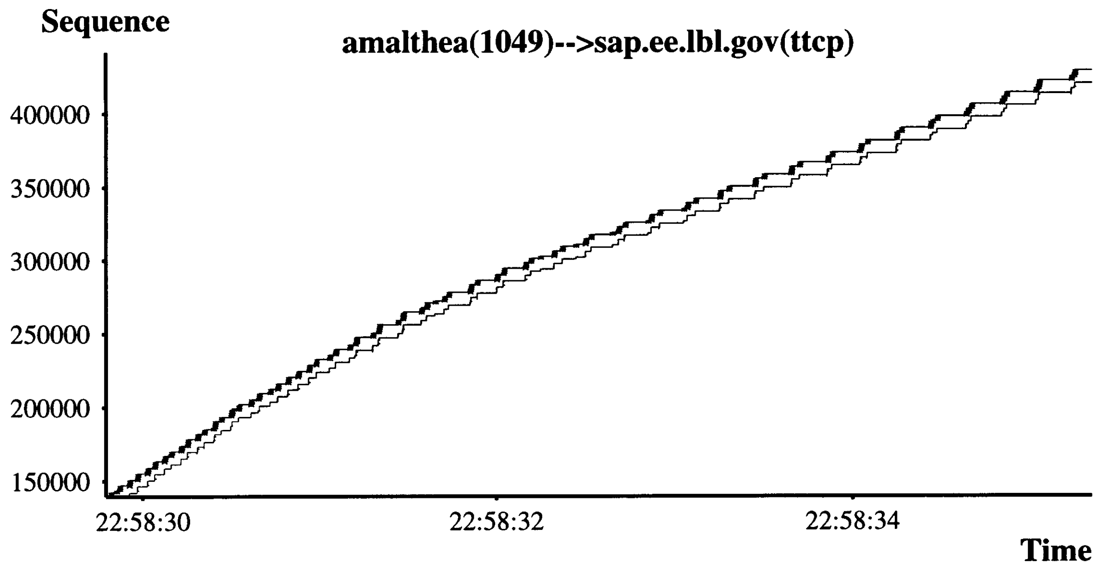

Figure 6-5: RTT between sender and receiver changing over time.

But how can these changes affect the performance of TCP transactions (besides the trivial factor or slower packet delivery)? Recall from our discussion in chapter4 that TCP uses a timer estimate of the RTT in order to detect when a packet has been lost. If the change in RTT is too sudden, the RTT estimate might not have enough time to readjust to the new network conditions, and may be set according to the RTT seen previously which is too short for the current congestion. Hence a timer can expire erroneously, and a packet could be unnecessarily retransmitted due to the longer RTT. While this situation is highly improbable in practice, it is an issue that

TCP needs to consider.

6.4 Analyzing the Behavior of Higher Level Protocols and Applications

Through our evaluation of TCP mechanisms we have seen the usefulness of our tool in providing a graphic representation that facilitates the analysis of network events. With our graphic representation, we were able to identify a number of situations in which the performance of TCP transactions was not completely optimal or was somehow affected by incorrect implementations of TCP mechanisms, or even external conditions. But the usefulness of our tool and our graphical representation of traffic extends in one more dimension: Since we can easily see the behavior of any TCP transaction, our tool could also be very useful in analyzing the performance of higher level protocols and applications that make use of TCP as their mean of communication.

As an example, we show two transactions of higher level protocols that use TCP. First, in figure 6-6 we show an example of a transaction using the **smtp** mail protocol. Note that there are two periods in which there is no communication flowing between the two ends: right after connection establishment and right after the actual message has been sent. Presumably, the initial idle time is due to the fact that smtp is waiting to verify the existence of the user for whom the mail is intended. The second idle period is probably due to the sender waiting for handlers in the receiving end to acknowledge the reception of the mail message and its successful result after storing the mail for the correct user.

Figure 6-7 shows an HTTP transaction between two hosts. Here we see that there is a long silence after the data has been transmitted. While connection establishment happens flawlessly, the process of closing the connection seems to be inefficient. If the sender of HTTP data is a very busy server, this inefficiency can be very costly because it can unnecessarily hold its resources for an extended period of time. It

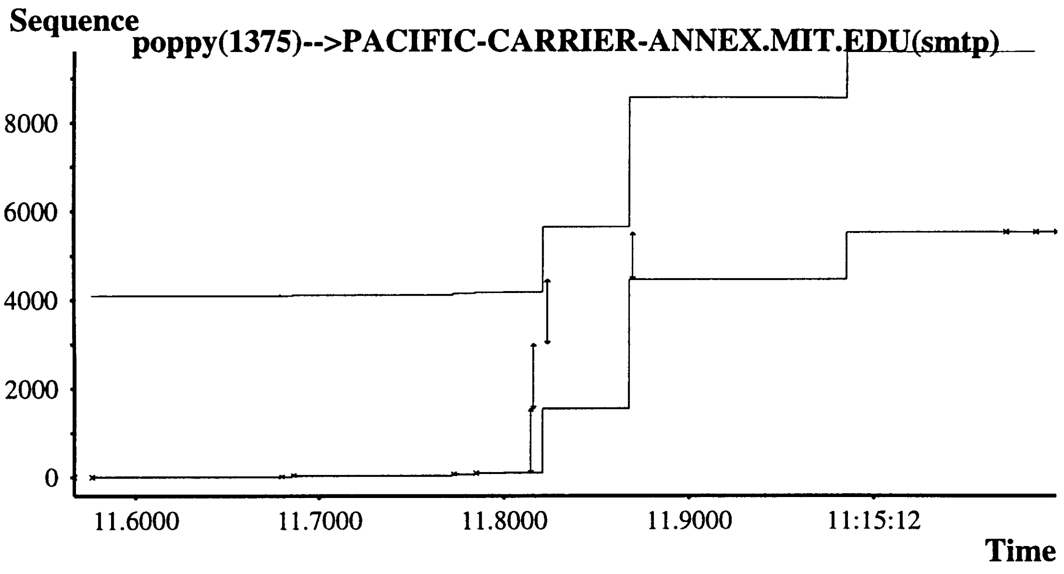

Figure 6-6: Electronic mail transaction (SMTP protocol).

has been suggested that this inefficiency when closing HTTP connections is due to one of the ends closing the transaction while the other end remains with the active transaction. We do not analyze the situation any further here. The purpose of showing this instance is to demonstrate that HTTP traffic can be readily analyzed using our representation, but we leave any further analysis as the subject of future work.

Based on these two simple examples, we can conclude that our tool facilitates the analysis of any type of TCP traffic, and could potentially serve to debug higher level protocols and applications, as well as TCP implementations like we showed in Chapter 5.

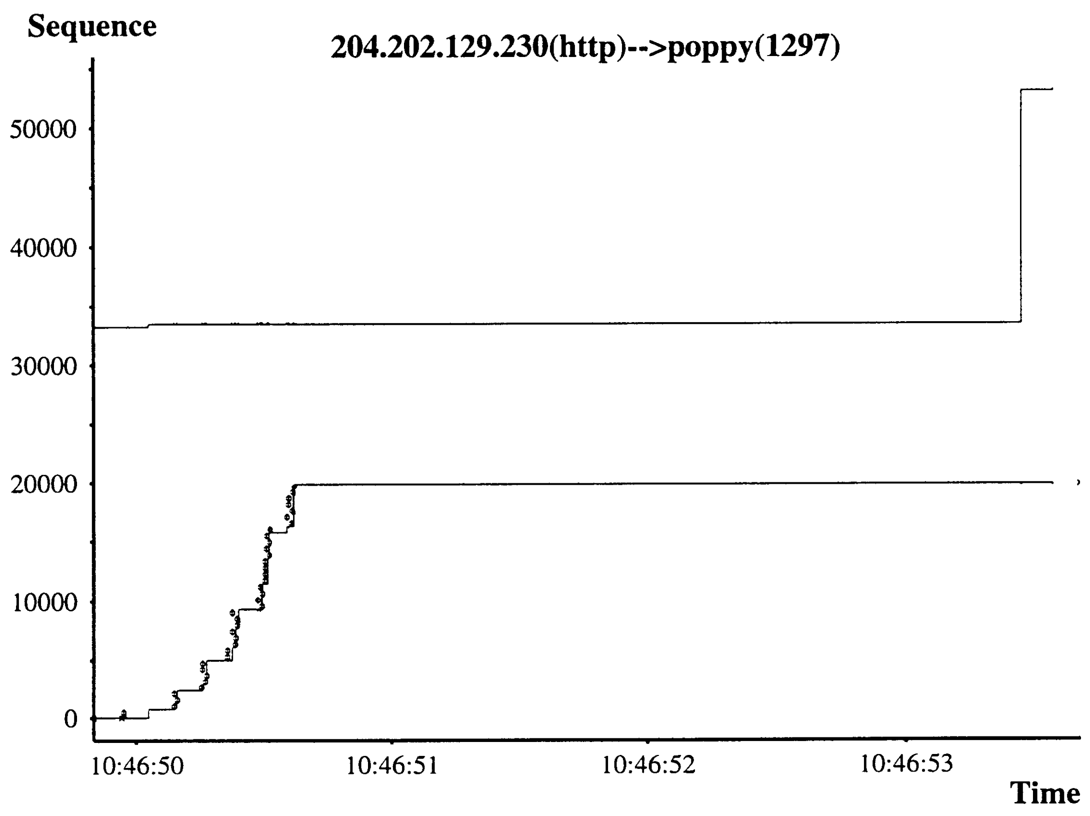

Figure 6-7: HTTP transaction.

Chapter 7

Conclusions

We have provided a framework that facilitates the evaluation of TCP transactions. Through our evaluations, we attempted to determine which performance mechanisms are included in certain TCP implementations and whether these mechanisms are implemented correctly. As a result, we were able to identify several examples of TCP implementations violating TCP standards. Many of the violations we observed are not only detrimental to the performance of their own transactions, but they can also to harm the performance of other transactions. Consequently, such violations to the TCP standards should be considered critical.

In addition to these pathologies in TCP implementations, we also found examples of TCP behavior that, while not being incorrect according to TCP standards, they are clearly inefficient from the perspective of transaction performance. In many of these cases we could argue that modifying the behavior of TCP could yield considerable benefits in terms of performance. However, the TCP standard purposely leaves many parameters unspecified so that developers can optimize their TCP implementations for particular network characteristics or usage constraints. Therefore, it would be incorrect to conclude that the implementations we tested should be modified to accommodate the optimizations that follow from our analysis because they might be optimizing different traffic patterns that we did not test here. Section 7.1 proposes possible extensions to our work that would allow us to verify whether TCP implementations have been tuned for particular network environments. Through such analysis, we might actually determine that the TCP implementations that performed poorly in our analysis perform better under different conditions.

The tools designed for this work can be readily used to extend the evaluation of TCP performance presented in this thesis. In order to study the extensions proposed in section 7.1, it would just be a matter of finding the correct network environment to evaluate our TCP implementations. Furthermore, the framework for TCP evaluation presented in this thesis also serves to analyze the communication characteristics of applications that use TCP as their mean of communication. As we saw in Chapter 6, it is trivial to visualize the traffic patterns generated by applications with our graphic representation of TCP traffic. In short, this thesis has shown that our framework for TCP analysis not only facilitates the process of validating the correctness of TCP implementations, but also serves for comparing TCP traffic under different network environments and for characterizing the communication patterns of applications that use TCP.

7.1 Future Work

The performance flaws exhibited by some of the TCP implementations we tested in this thesis might be due to the fact that they have been tuned to operate in network environments that differ from the ones used for the tests conducted here. For instance, the developers of the Windows 95 and Windows NT implementations might have assumed that the most popular communication medium for their implementations is not Ethernet, but modem links. It could be the case that the behavior we classified as inefficient during our evaluation, could actually yield better performance when communicating through other environments such as modems. Besides modem links, it would also be interesting to evaluate the performance of TCP implementations in wireless links, Fast Ethernet links, and ATM networks.

Another parameter in our evaluation process that should be considered in future evaluations of TCP performance is the characteristics of the traffic patterns between TCP hosts. As discussed in Chapter 3, we evaluated the performance of transactions that send large amounts of data, just like FTP transactions. It would be interesting to observe how bursty communication patterns (like that of a telnet connection) affect the performance of TCP implementation. Due to the difficulty of developing a traffic generator with these characteristics, we leave this analysis for future work.

Bibliography

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

[2] L. Brakmo and L. Peterson. Performance problems in BSD4.4 TCP. *Computer Communication Review*, 25(5):69–84, October 1995.

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

[4] D. Comer and J. Lin. Probing TCP Implementations. In *Proceedings of the 1994 Summer USENIX Conference*, pages 245–255, Boston, MA, June 1994.

[5] S. Dawson, F. Jahanian, and T. Mitton. Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool. *Software Practice & Experience*, 27(12):1385–1410, December 1997.

[6] S. Floyd and K. Fall. Router Mechanisms to Support End-to-End Congestion Control. Unpublished manuscript. URL: http://www-nrg.ee.lbl.gov/floyd/papers.html, February 1997.

[7] J. Hoe. Start-up dynamics of tcp's congestion control and avoidance schemes. Master's thesis, Massachusetts Institute of Technology, Cambridge, June 1995.

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

[9] V. Jacobson. Pathchar: A tool to infer characteristics of internet paths. Slides from talk at MSRI. Available through anonymous FTP to ftp.ee.lbl.gov, April 1997.

[10] V. Jacobson, C. Leres, and S. McCanne. Tcpdump. Available through anonymous FTP to ftp.ee.lbl.gov, June 1989.

[11] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, CA, January 1993.

[12] V. Paxson. Automated Packet Trace Analysis of TCP Implementations. In *Proceedings of SIGCOMM '97*, September 1997.

[13] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics.* Ph.D. dissertation, University of California, Berkeley, April 1997.

[14] J. Postel. *Transmission Control Protocol.* Request for Comments 793, DDN Network Information Center, SRI International, September 1981.

[15] T. Shepard. Tcp packet trace analysis. Technical Report 494, MIT Laboratory for Computer Science, February 1991.

[16] W. R. Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, 1994.

[17] W. R. Stevens. *TCP/IP Illustrated*, volume 3. Addison-Wesley, 1996.

[18] J. Kemperman T. Ott and M. Mathis. The stationary behavior of ideal TCP congestion avoidance. Available through anonymous FTP to ftp.bellcore.com, August 1996.

[19] G.R. Wright and W. R. Stevens. *TCP/IP Illustrated*, volume 2. Addison-Wesley, 1995.

[20] L. Zhang and D. Clark. Observations on the dynamics of a congestion control alrgorithm: The effects of two-way traffic. In *Proceedings of SIGCOMM '91*, September 1991.