END-TO-END CONGESTION DETECTION AND AVOIDANCE IN WIDE AREA NETWORKS

by

Lawrence Sivert Brakmo

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 9 6

END-TO-END CONGESTION DETECTION AND AVOIDANCE IN WIDE AREA NETWORKS

Lawrence Sivert Brakmo, Ph.D.
The University of Arizona, 1996

Director: Larry L. Peterson

As human dependence on wide area networks like the Internet increases, so does contention for the network's resources. This contention has noticeably affected the performance of these networks, reducing their usability. This dissertation addresses this problem in two ways.

First, it describes TCP Vegas, a new implementation of TCP that is distinguished from current TCP implementations by containing a new congestion detection and avoidance mechanism. This mechanism was designed to work in currently available wide area networks and achieves between 37% and 71% better throughput on the Internet, with one-fifth to one-half the losses, as compared to the current implementation of TCP.

Second, it describes *x*-Sim, a network simulator based on the *x*-kernel, that is able to simulate the topologies and traffic patterns of large scale networks. The usefulness of the simulator to analyze and debug network components is illustrated throughout this dissertation.

STATEMENT BY AUTHOR

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Larry Peterson. He has been the ideal advisor, one whose experience and knowledge provided me with the haven that allowed me the freedom to explore my own ideas. I have been very fortunate to have both his guidance and friendship during this journey.

I am also grateful to the other members of my committee, Pete Downey and John Hartman, for their comments and suggestions on my work. I am thankful to Udi Manber for our many enjoyable discussions. I would like to thank the minor members of my committee, Ted Laetsch, and specially William Velez, who has been a great counselor and friend for many years.

I also want to thank everyone in the Computer Science department at Arizona for making my stay here so pleasurable. Special thanks to Cara Wallace, Margaret Newman, Wendy Swartz, John Cropper, Cliff Hathaway and John Luiten.

Finally, I would like to thank my fellow graduate students, Vince Freeh, David Mosberger, Peter Drushel, Mark Abbot, David Lowenthal, Nina Bhatti, and Mudita Jain for their friendship.

This work was supported by the National Science Fundation under Grant IRI-9015407 and by ARPA under Contract DABT63-91-C-0030.

DEDICATION

To my parents, for always supporting my quest for knowledge, even when the journey strayed from the direct path. For raising me in an environment that encouraged the pursuit of knowledge. For always giving of themselves. And most importantly, for teaching me the meaning of love, caring, and selflessness through their example.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

As human dependence on wide area networks like the Internet increases, so does contention for the network's resources. This contention has noticeably affected the performance of these networks, reducing their usability. This dissertation addresses this problem in two ways.

First, it describes TCP Vegas, a new implementation of TCP that is distinguished from current TCP implementations by containing a new congestion detection and avoidance mechanism. This mechanism was designed to work in currently available wide area networks and achieves between 37% and 71% better throughput on the Internet, with one-fifth to one-half the losses, as compared to the current implementation of TCP.

Second, it describes x-Sim, a network simulator based on the x-kernel, that is able to simulate the topologies and traffic patterns of large scale networks. The usefulness of the simulator to analyze and debug network components is illustrated throughout this dissertation.

CHAPTER 1

Introduction

Our society's dependence on computer networks and computer communication has been increasing during the last two decades, a trend that is likely to continue in the foreseeable future. The importance of computer networks in our daily life can be attested by the exponential growth of the Internet, a global computer network consisting of thousands of connected heterogeneous networks. When traced back to its origins, the number of hosts connected to the Internet has been doubling in size every year since 1981.

This growth has been fueled by applications that have enabled new levels of communication and information sharing between people in different cities or countries. Graphical World Wide Web (WWW) browsers, like Mosaic and Netscape, have been responsible for most of the Internet's growth in the last two years by simplifying the chore of making information publicly available and easy to find on the Internet.

A downside to this growth has been an increase in contention for Internet resources. This contention has noticeably affected the performance of distributed applications like Mosaic, at the same time that our dependence on such applications is increasing. This dissertation addresses this problem in two ways. First, it proposes a new technique to manage this contention, and second, it describes a set of tools to analyze the performance and behavior of such mechanisms.

1.1 Computer Networks

Computer communication is made possible by the connectivity provided by computer networks [24]. At the lowest level, computers are directly connected with each other through a physical medium, such as a cable. We call such a physical medium a *link*, and refer to the computers it connects as *nodes*, to underscore the fact that

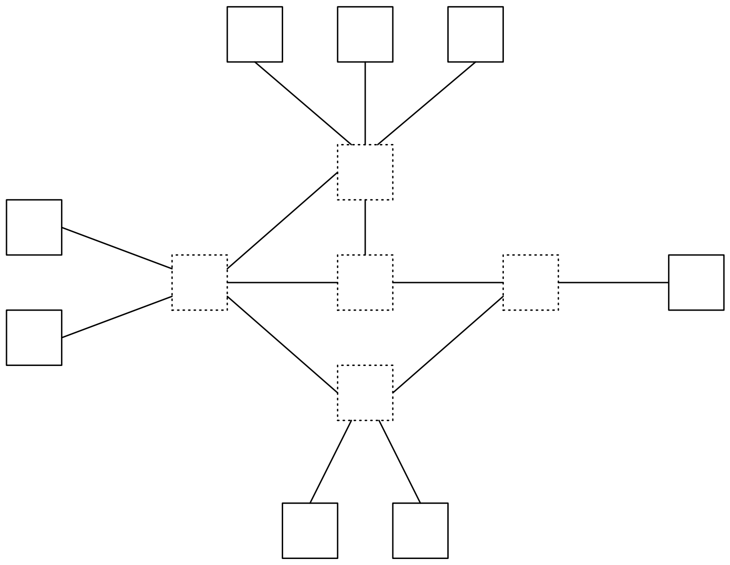


Figure 1.1: Switch network.

they may be implemented by specialized hardware rather than general purpose computers. Links can be *point-to-point*, connecting a pair of nodes, or *multiple access*, connecting more than two nodes. The *bandwidth* (throughput) of a link is given by the maximum number of bits than can be transfered over the link in a certain period of time. The *latency* (delay) of the link refers to the time it takes a particular bit to travel from one end of the link to the other.

Links provide direct connectivity between nodes and are one of the basic blocks upon which indirect connectivity is built. The need for indirect connectivity arises from the desire to have total connectivity: the ability of reach any node from any other node. It would be prohibitively expensive to achieve this with direct connectivity, since the number of links needed would be $\mathcal{O}(n^2)$, where n is the number of nodes.

Figure 1.1 shows a network with limited direct connectivity. Indirect connectivity is achieved by having any node connected to at least two links, shown in the figure by squares with dashed lines, forward (or route) data received on one link out on another. If this is done in a systematic way, data from any node can reach any other node. This type of network is known as a *switched network*.

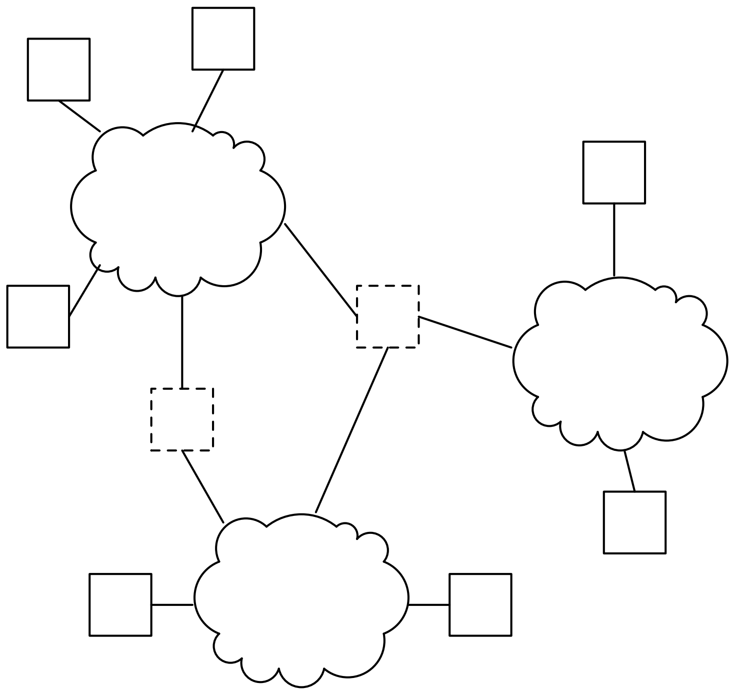


Figure 1.2: Interconnection of networks.

The most common type of computer network is a *packet-switched* network, where nodes send discrete blocks of data to each other. These blocks are called *packets* or *messages*. The most common strategy used to forward the packets in packet-switched networks is called *store-and-forward*. Each node waits until it has received a whole packet before forwarding it at a later time to the appropriate output link. The *end-to-end latency* of a packet is defined as the amount of time it takes the packet to go from the source host to the destination host.

The nodes inside the network, connected to two links or more, usually have the sole function of storing and forwarding packets. They are generally called *switches* or *routers*. The nodes at the periphery of the network, connected to only one link, are general purpose computers whose goal is to support users and run their applications. They are called *hosts*.

Packets must contain special information so routers know how to forward them. This information is usually held at the beginning of the packet in an area called the packet *header*.

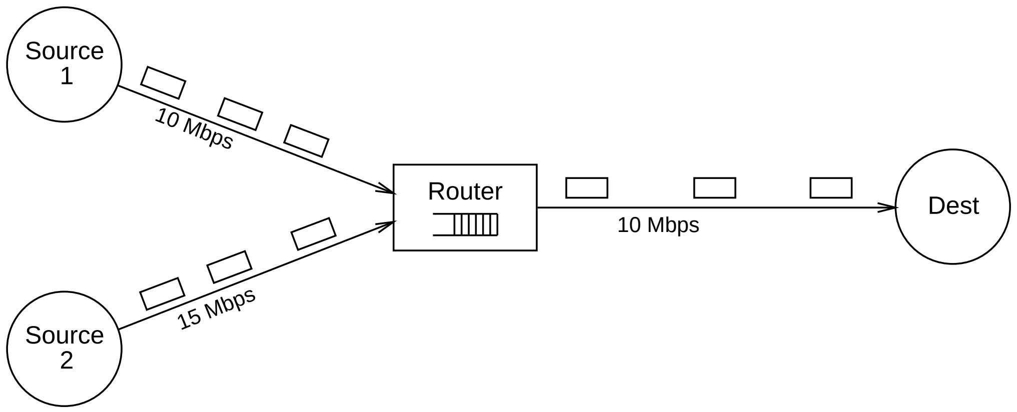


Figure 1.3: Small congested network.

Networks can be categorized according to their size. Two well known examples are LANs (local area networks), which typically extend less than one kilometer, and WANs (wide area networks) which can extend to any size. Larger networks can be built recursively from smaller, independent ones. An example is shown in Figure 1.2, where each cloud represents the switches and links of independent networks. The nodes connecting the independent networks are called *gateways* or *routers*. We call such a network an *internetwork*, or internet for short. The largest current operational internet is called the *Internet* (capitalized).

1.2 Congestion

Congestion occurs in a router when the aggregate bandwidth of incoming packets, destined for a particular output link, exceeds the link's bandwidth. Figure 1.3 shows a very simple network consisting of three hosts (two sources and one destination) and one router. The two incoming links have bandwidths of 10 and 15 Mb/s (millions of bits per second), the output link has a bandwidth of 10 Mb/s. Congestion can occur in this network—even with only one source active—as long as source 2 is able to send the data faster than 10Mb/s.

There are two types of congestion, *transient* and *persistent*. Transient congestion can be managed with a queue of buffers at the router. During the congestion period the queue will grow and contain the excess packets. When the congestion period ends, the packets in the queue are sent through the output link and the queue shrinks. All current routers contain some buffer space to handle transient congestion. It is worthwhile to note that a non-empty queue increases packet latency.

Persistent congestion occurs when the congestion period is long enough that the queue overflows (packets are removed from the network by the router). In this dissertation we interpret the term congestion, when used by itself, to mean persistent congestion.

Strategies to handle congestion can be divided into two categories. *Insider* strategies are those in which the routers are actively involved. For example, the routers can detect when their queues become non-empty and send a special message to the sources asking them to slow their sending rate. Such a strategy, whose goal is to prevent queue overflow, is called *congestion avoidance.*

End-to-end strategies are those in which the routers are not actively involved, and the hosts must use indirect methods to detect the congestion. For example, the destination can inform the source when a packet has been lost, at which time the source can slow its sending rate, under the assumption that all losses are caused by congestion. This type of strategy, whose goal is to control congestion once it occurs, is called *congestion control.*

Insider strategies have the advantage of being able to detect congestion in its earliest stages. Their main disadvantage is that they require a certain amount of processing be done at the routers, increasing their cost and complexity.

In principle, routers can use the *Internet Control Message Protocol* (ICMP) to inform sources that they should decrease their sending rate (source quench request). This is not done for two reasons. First, there is no accepted mechanism for deciding how to pick which sources should be informed and when they should be informed. Second, there is no accepted general mechanism for decreasing the sending rate of the sources when the source quench requests are received.

End-to-end strategies have the advantage of being implemented in general purpose computers, so they are easier to implement and modify. Routers, on the other hand, are sometimes implemented with specialized hardware, so modifying their programming is much harder. Another advantage of end-to-end strategies is that they make few (or no) assumptions about the routers in the network, so they can work in any network. Their disadvantage, as mentioned earlier, is that they must depend in less accurate indirect methods to detect congestion.

Routers have historically been designed to be as simple as possible. As a result, there is no widespread insider congestion strategy in use at this time on the Internet. The only congestion control mechanisms in use today are end-to-end.

As a final comment, when developing a new congestion strategy, an end-to-end strategy has the added advantage of being easily tested in real networks. All that is required is the right software at a few of the hosts. On the other hand, testing an insider strategy on a real network requires changes to the routers in the network. This implies that it would be nearly impossible to test an insider mechanism on large networks like the Internet.

1.3 Protocols

A *protocol* is a communication standard. It describes a set of rules that allow different machines to communicate with each other without ambiguity. In particular, it specifies how to pass messages, details the message format, and describes how to handle error conditions.

The task of writing protocols is simplified by following a layered approach, where lower level protocols provide some of the functionality upon which higher level protocols are successively build. For example, the lowest layer may provide host-to-host connectivity, the next highest layer may provide reliability, and so on.

Two of the most important protocols are the *Internet Protocol* (IP) and the *Transmission Control Protocol* (TCP). The main goal of IP is the delivery of packets in heterogeneous connected networks. It forms the infrastructure that allows the "seamless" integration of heterogeneous networks. IP follows a *best effort* approach to the delivery of packets; there are no guarantees that the packets will be successfully delivered to their final destination. IP is a *connection-less* protocol, meaning that it maintains no state information about successive packets. Each packet is handled independently from all other packets.

TCP is layered on top of IP and provides such services as reliability and end-to-end congestion control. It is responsible for most of the Internet's traffic. In particular, traffic traces collected at one of the main Internet access points (FIX-West) show that TCP is responsible for more than 80% of the bytes and 70% of the packets seen in the traces. The most widely used implementations of TCP—Tahoe and Reno—are named after the Berkeley Software Distributions (BSD) of Unix they were a part of.

1.4 Dissertation Outline

The goal of this dissertation is to addresses the problem of increased contention for the resources on the Internet. It does so in two ways. First, it proposes a new end-to-end congestion avoidance strategy. Second, it presents a new architecture for network simulation, together with a set of analysis tools.

The rest of this dissertation is organized as follows. Chapter 2 presents and describes the architecture of x-Sim, a realistic network simulator that is able to simulate the topologies and traffic patterns of large scale networks. This chapter also describes a set of tools whose purpose is to aid in the analysis of network behavior and performance.

The simulator and its tools are then used to do a detailed analysis of TCP in Chapter 3. This analysis shows the weaknesses in TCP's congestion control mechanism and provides the insight needed for developing a new congestion avoidance mechanism. This new mechanism is implemented in TCP by modifying TCP's existing congestion control mechanism. This new congestion avoidance mechanism is described in detail in Chapter 4 in the context of its TCP implementation.

Chapter 5 examines the performance of both the existing and new implementations of TCP through experiments, in both real and simulated networks. The performance of the new congestion avoidance mechanism is gauged by comparing it to the performance of the current TCP implementation.

Chapter 6 presents some of our experiences with x-Sim, including a study of a new implementation of TCP. This study uncovers a series of problems with this implementation, proposes solutions to these problems, implements the changes, and measures the performance of the fixed implementation. It also gives a set of guidelines for network simulation.

Finally, Chapter 7 summarizes the major contributions of this dissertation and discusses possible future directions.

CHAPTER 2
Simulator and Visualization Tools

Simulation is a critical tool for the development, analysis, testing and evaluation of network components. This is specially true in the case of wide area networks (WANs) where we are unlikely to have access to more than a few of the system's components. Access to the other components of a WAN is particularly important when comparing congestion detection, control and recovery mechanisms. It is not sufficient to show that one particular connection with the new mechanism performs better than one with the old mechanism; it is important to measure the effects that the new mechanism has on the other traffic in the network. For example, it is inadvisable to employ a mechanism that results in unfair sharing of network resources, or in low utilization of these resources.

This chapter describes x-Sim, a new network simulator we have implemented to aid with network research, as well as a set of tools developed to aid in the analysis of the simulations. It starts with a discussion of the x-kernel—a protocol implementation environment [11]—which is the framework on which the simulator is built. It also describes the minor modifications to the x-kernel that were necessary to support the simulator. Next, there is a description of the simulator itself, starting with the overall design and ending with the major individual components. This is followed by a comparison of x-Sim with existing network simulators, as well as with other alternative approaches such as network emulation. The chapter concludes with a description of the analysis tools.

2.1 The x-Kernel

The x-kernel specifies an architecture and provides mechanisms for the efficient implementation of network protocols. It is an object-oriented architecture whose main objects are protocols, sessions and messages. Protocol objects are static and correspond to conventional network protocols; e.g., IP, UDP, TCP. Session objects are dynamically created and can be thought of as instances of protocol objects which contain the data structures that represent the local state of some network connection. Messages are also dynamically created objects that represent the basic unit of communication among hosts or the entities. The data contained in a message object corresponds to one or more protocol headers and user data.

Both protocol and session objects are passive, unlike messages, which can be thought of as active objects that move through session and protocol objects, as well as through the network. The fact that messages are carried along by either execution threads or network packets should not prevent us from the useful abstraction of viewing messages as active objects.

To open a new TCP connection, the TCP protocol performs the *Open* operation on IP, its lower level protocol. Once the connection is established, the operation returns the handle of a session. This session is then used in all future operations dealing with this connection. To send a message, one only needs perform the *Push* operation, with a message as an argument, on the session.

The *x*-kernel encourages a modular approach to protocol implementation, where complex protocols can be implemented by many protocol objects. Each type of object supports a uniform set of operations which simplifies the composition of protocols.

Figure 2.1(a) illustrates a suite of protocols that might be configured into a given instance of the *x*-kernel. This figure is an example of a *protocol graph.* Figure 2.1(b) gives a schematic representation of the *x*-kernel objects corresponding to the suite of protocols in (a); protocol objects are depicted as rectangles, the session objects associated with each protocol object are depicted as circles, and a message is depicted as a "thread" that visits a sequence of protocol and session objects as it moves through the *protocol-session graph.*

The following subsections briefly describe each of the objects and some of the main operations supported by them; a full description can be found in [12].

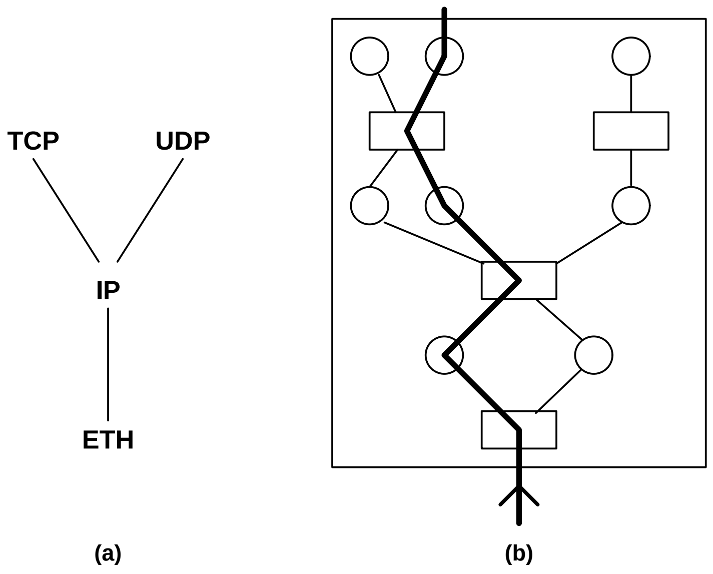


Figure 2.1: Example x-Kernel configuration.

2.1.1 Protocol Objects

Protocol objects serve two major functions: they create session objects and they demultiplex messages received from the network to one of their session objects. Sessions can be created actively (*active open*), with the *Open* operation, or passively (*passive open*) with *OpenEnable* and *OpenDone* operations.

Active opens are triggered by higher level protocols or user processes. Passive opens support session creation triggered by a message arriving from the network. For example, a server could invoke the *OpenEnable* operation on the object corresponding to the TCP protocol advising it that it will accept messages addressed to a given port number and originating from any host. Then, when the first message from a given client arrives to that port, the TCP protocol creates a new session and invokes the *OpenDone* operation of the higher level protocol which had originated the *OpenEnable* operation, passing the newly created session as an argument. The server can then use the session to send messages back to that client.

In addition to creating sessions, each protocol also "switches" messages from the network to one of its sessions with a *Demux* operation. *Demux* takes a message as its argument, and either passes the message to one of its sessions, or creates a new session—using the *OpenDone* operation—and then passes the message to it. In the case of a protocol like IP, *Demux* may also "route" the message to some other lower-level session.

Each protocol object's *Demux* operation makes the decision as to which session should receive the message by first extracting the appropriate external id(s) from the message's header. It then uses a map routine—part of the x-kernel's framework—to translate the external id(s) into either an internal id for one of its sessions (in which case *Demux* passes the message to that session) or into an internal id for some high-level protocol (in which case *Demux* invokes that protocol's *OpenDone* operation and passes the message to the resulting session).

2.1.2 Session Objects

A session is an instance of a protocol created at runtime as a result of an *Open* or *OpenDone* operation. Intuitively, a session corresponds to the end-point of a network connection; i.e., it interprets messages and maintains state information associated with a connection. For example, TCP session objects implement the sliding window algorithm (among other tasks), IP session objects fragment and reassemble datagrams, UDP sessions only add and strip UDP headers.

The meaning of a connection is, of course, protocol dependent. For example, an IP session object is associated with a unique { protocol_id, remote_ip_addr } id, whereas a TCP session object is associated with a unique { local_port, remote_port, remote_ip_addr } id. Hence, one IP session will handle all TCP connections to a given host, whereas each TCP connection will have its own TCP session object.

Sessions support two primary operations. The *Push* operation is invoked by a high level session to pass a message down to some low-level session. The *Pop* operation is invoked by the *Demux* operation of a protocol to pass a message up to one of its sessions.

For example, once a connection has been established, a session wishing to send a message out invokes the *Push* operation of the corresponding lower-level session.

Only session objects, and no protocol objects, are involved in the path of the message as it works its way down. Note however, that a message's journey through the session objects is not necessarily continuous, as it may temporarily block if, for example, a protocol enforces congestion or flow control.

On the other hand, as a message works its way up a protocol-session graph, a lower-level session invokes the *Demux* operation on the corresponding higher-level protocol, which in turn finds the session associated with that connection and passes it the message by invoking its *Pop* operation.

2.1.3 Message Objects

Conceptually, messages are active objects. They either arrive at the bottom of the *x*-kernel (i.e., a device) and flow upwards, or they originate from a user process or higher level protocol and flow downward to a device. While flowing downward, a message visits a series of sessions via their *Push* operations. While flowing upward, a message alternatively visits a protocol via its *Demux* operation and then a session in that protocol's class via its *Pop* operation.

As a message visits a session on its way down, headers are added, the message may fragment into multiple message objects, the message may suspend itself while waiting for a reply message, and so on. As a message visits a session on the way up, headers are stripped and the message may suspend itself while waiting to reassemble into a larger message.

The data portion of a message is manipulated—e.g., headers attached or stripped, fragments created or reassembled—using the buffer management routines described in the next subsection.

2.1.4 Support Routines

A *buffer manager* is used to allocate space, concatenate two buffers, break a buffer into two separate buffers and truncate the left or right end of a buffer. The buffer manager is implemented in a way that allows multiple references to arbitrary pieces of a given buffer without incurring any data copying. The buffer manager is used to implement and manipulate messages; i.e., add and strip headers, fragment and reassemble messages.

A *map manager* is used to manage a set of bindings of one identifier to another. The map manager supports the addition of new bindings to the set, removal of bindings from the set, and the mapping one identifier into another relative to a set of bindings. Protocol implementations use the map manager to translate identifiers extracted from message headers—e.g., addresses, port numbers—into capabilities for x-kernel objects.

An *event manager* is used to create and manage threads, as well as timer events. The event manager lets a protocol specify a timer event as a procedure that is to be called at some future time. By registering a procedure with the event manager, protocols are able to *timeout* and act on messages that have not been acknowledged.

2.1.5 x-Kernel Modifications to Support the Simulator

Only minimal modifications to the x-kernel were necessary in order to use it as the infrastructure for the simulator. No modifications were done to the x-kernel interfaces, and only some of the routines implementing the interfaces were changed. This means that protocols can be moved between the simulator and the x-kernel without modifications.

The event manager was modified to support a global virtual time. The simulator keeps track of a simulated global time, and when there are no more threads ready to run at this time—all of the threads are either blocked or scheduled to run at a future time—the event manager picks the thread that is scheduled to run nearest in the future, increments the global time to that thread's scheduled time, and finally, the simulator starts executing the thread.

Since event calendar processing can consume as much as 40 percent of simulation time ([6], [21]) it is important to choose the right data structure to manage event lists. Events lists are represented by priority queues, which are abstract data types supporting the following two operations: (1) insertion of an element (event) with a key (timestamp), and (2) removal of the element (event) with the smallest

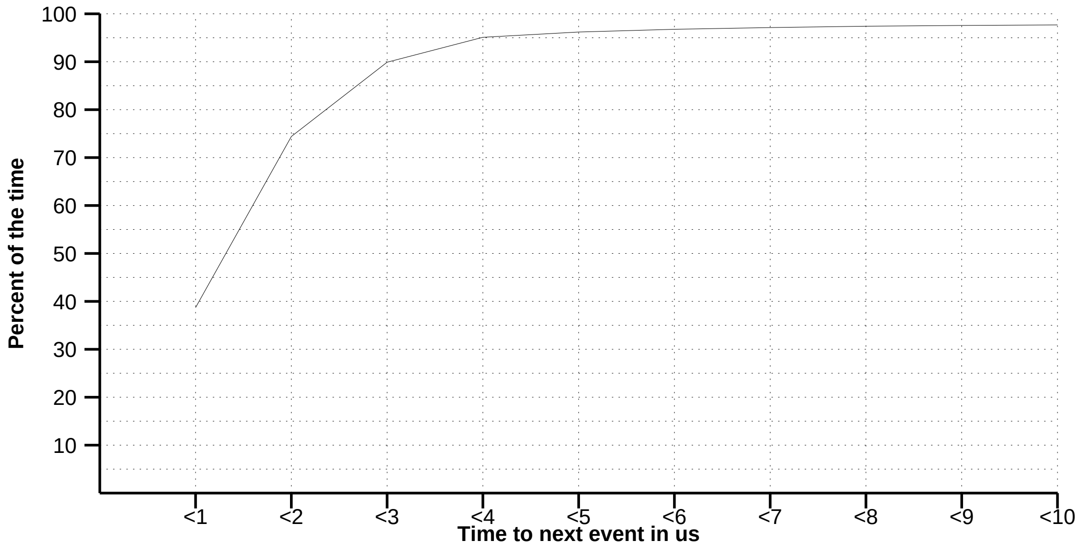


Figure 2.2: Time difference between current time and time of next event.

key (timestamp). There are numerous data structures that can be used to implement priority queues (see [5], [18] and [3]). x-Sim uses calendar queues based on an analysis of the distribution of event schedule times obtained during simulation runs. Calendar queues consist of buckets called *days*, with a variable number of days making up a *year*. Each bucket contains all events scheduled for that day—independently of the year—in sorted order. Insertion is $O(1)$ when the number of events per bucket is small. Removal is also $O(1)$ as long as few buckets need to be searched to get the next event. Figure 2.2 shows the cumulative distribution of the time difference between the current time and the time of the next scheduled event during a simulation. It shows that 96 percent of the time the next event is within 5 microseconds of the current time.

The message subsystem was also modified. When simulating a network with 300 Mb/s links and 200ms round trip times (RTT), the network may contain up to 7.5 MB of data in packets within its links. Since reliable protocols must save a copy of the data until they receive an acknowledgment indicating the data was successfully delivered, the network may contain more than 15 MB of data either as packets within its links or stored within the buffers of reliable protocols. When doing simulations, one normally does not care about the contents of most of the data in the messages. Only the beginning of the message, which contains the protocol header information, is important. The message subsystem was modified so messages carry only about 200 bytes of non-zero data.

In the standard x-kernel, the protocol graph is static and it is specified at compile time. In contrast, x-Sim uses enhanced versions of the routines dealing with the creation of the protocol graph, and as a consequence, the protocol graph can be specified at runtime. The simulator reads a file specifying the network topology: number and type of networks, number of hosts and routers, how the networks, hosts and routers are connected, the protocol graph for each host, and configuration parameters for all of the elements in the simulation scenario. These configuration parameters specify the addresses of each host and network, the speed and delay of each point-to-point link and network, whether Ethernet networks should simulate collisions, and so on.

Finally, the simulator augments the x-kernel interface with a few new functions. For example, the x-kernel's event manager has a resolution of 1 microsecond; this is perfectly adequate for events scheduled by protocols but not for a network simulator which may need to simulate a 600 Mb/s ATM link that only takes 0.71 microseconds to consume each packet (ATM packets are 53 bytes long). The simulator allows event scheduling with nanosecond accuracy through a new event scheduling function.

2.2 Simulator

x-Sim is a network simulator based on the x-kernel. It provides a framework for developing, analyzing, and testing network protocols; for evaluating congestion control mechanisms; and for studying network dynamics.

x-Sim is tightly integrated with the x-kernel architecture; it runs x-kernel protocols and protocols can be moved between the simulator and any x-kernel implementations without any change in the protocol code. This allows us to experiment with protocol implementations both in the controlled environment of the simulator as well as in real networks like the Internet.

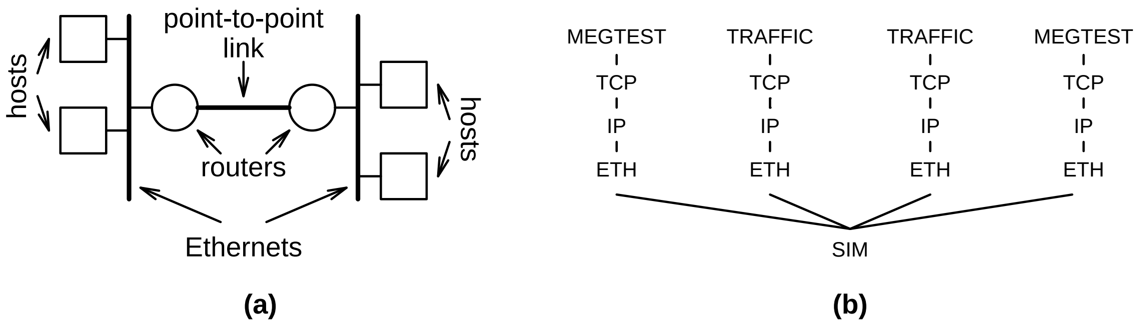


Figure 2.3: Example network and corresponding protocol graph.

When designing a network simulator, one must balance accuracy and simulation speed. In general, higher accuracy results in slower simulations. As computing power continues to increase, the cost of the simulations becomes less significant, but it is never irrelevant. Our approach was to first create a simulator that is as accurate as we could make it, and then, when simulation speed becomes an issue, to introduce optional mechanisms that decrease the accuracy and increase the simulation speed. This approach, of going from more accurate to less accurate simulations, allows us to gauge the effect that the lower accuracy has on the simulations; only then can we decide if the resulting less accurate simulations are good enough to be useful.

The modular nature of the x-kernel lends itself to packaging the code simulating an underlying network as a protocol (SIM) which lies at the bottom of the protocol graph. Hosts are represented by protocol subgraphs lying on top of the SIM protocol. Routers are represented either by protocol subgraphs or by SIM modules. Protocol subgraphs are used for implementing complex routers, such as those containing complete implementations of IP. Simpler routers are easier to implement as SIM modules. Figure 2.3 shows a network on the left and the corresponding x-kernel protocol graph on the right, with routers implemented as SIM modules. In this model, passing a message to the SIM protocol is equivalent to putting it into a network's physical wire. The SIM protocol then simulates its journey through the modeled network until it reaches a protocol subgraph representing the final destination or an intermediate router, at which point SIM passes the message up to the protocol subgraph.

The following describes the structure of the SIM protocol in terms of three fundamental components—*links*, *nodes*, and *load*. We then compare x-Sim to existing simulators and emulators, presenting the relative strengths and weaknesses of each approach.

2.2.1 Links

Physical links can usually be simulated in software very accurately, at least to the level of the written specification. Simulating a link's real behavior accurately may entail some analysis of real implementations to find, among other things, the likelihood of losses due to noise. For example, accurately simulating an ethernet should include the ability to simulate collisions and the exponential backoff strategy. As another example, simulating ATM networks implies the use of small packets, increasing the complexity level—and execution time—of a simulation.

The architecture of x-Sim closely models the interactions between a link and the host/network adapter. The lowest software layer in each node (driver) is tightly coupled to the type of link (network) to which it is connected; together they model the interactions that occur in a real computer between the network, the adapter, and the device driver.

The three main operations of the driver/network interface in the simulator are: (1) queries to the network, for example to check if it is busy; (2) transmissions of packets, and (3) callbacks from the network, which are used to model interrupts. For example, a node connected to an Ethernet first queries the network to see if it is busy. If it is not, it sends the packet. Otherwise, the node registers a callback to be notified when the network is not busy. Once the packet has been transmitted successfully, the driver gets a callback notifying it of this fact so it can delete its copy of that packet. However, since Ethernets allow collisions, the interactions can become more complicated. For example, a query may return that the network is not busy, including a transmission right after another host has started transmitting, this will result in a collision of packets. In this case, both hosts get a callback at some later time indicating that the transmission failed. Each host then schedules a retransmission at a later time based on an exponential backoff mechanism.

The simulator currently supports point-to-point links, Ethernet links (with and without collisions), and a multiple access link that is a generalization of an Ethernet link. The first and last type of links (networks) allow the user to specify several parameters, including the bandwidth, delay, byte overhead per packet, a minimum and maximum packet size, and a loss or corruption probability. The second link type exactly models an Ethernet.

2.2.2 Nodes

Physical links are used to connect network nodes, which are usually either routers or hosts. To capture their behavior, we need to consider the characteristics of both the processor and the software, thereby simulating the delays introduced by both the nodes, and the algorithms that they run.

In x-Sim, the software running on a node is represented in one of two ways: (1) by an x-kernel style protocol graph or (2) by a program that implements an abstract specification of the node. When using a protocol graph, each protocol in the graph is given by the actual C code that implements the complete protocol. In other words, the first case can be viewed as supporting a *direct execution* simulation. The latter case is commonly used to implement a simplified (less accurate) model of a node.

For example, a router can be simulated with an actual implementation of IP (plus the necessary lower level protocols) by using existing x-kernel protocol modules, or by implementing a simplified specification that models some queuing discipline, such as FIFO. Note that when a protocol is given as a module in the protocol graph, then this is exactly the same implementation that runs in the x-kernel. This makes it possible to move a protocol back-and-forth between a simulated network and a real network.

Note that even though the goal is accurate simulation, there are times when practical realities force us to make small deviations from the real code. An illustrative example is the length of the TIME_WAIT state in TCP, which is one minute in the latest BSD implementation. When doing complex WAN simulations, with more than one thousand connections starting and ending during every simulated second, it is not feasible to keep the connection state (after the connection has been closed) around for the full simulated minute as this ties up too much memory. Our solution is to reduce this delay. We can do this because we know our particular simulation does not require the full one minute delay for correctness.

To simulate the processor accurately, one needs to account for the delays resulting from the execution of the code invoked to handle each packet. For an even more accurate simulation, one needs to also add some variance to these delays to model interactions between different subsystems. Simulating the delays and variance with total accuracy is probably not practical, as the variance may be due to cache and TLB effects. Even if one could simulate all of these details, it may not be the best thing to do as it ties the results of the simulation to a specific model of hardware. x-Sim allows the protocol code to introduce delays by specifying periods of time during which the node does no other processing; incoming packets cannot be processed during such a delay. These delays might be based on measurements of the code running on an actual processor and can be specified by a uniform or normal distribution.

2.2.3 Load

Network load is usually the result of many applications executing on numerous hosts. In the case where one is interested in studying the behavior under realistic load, realistic traffic sources are of the utmost importance. For example, traffic created by multiple bulk TCP transfers is radically different from traffic generated by multiple FTP, Telnet, SMTP and NNTP connections.[1]

The first step in the creation of realistic traffic is to use a common implementation of the underlying protocols—x-Sim gets this from the protocol graph. Next, we need a way to model the traffic. Here, x-Sim contains a protocol that simulates Internet traffic based on *tcplib* [7]. *Tcplib* is a library that models Internet traffic sources

[1]The File Transfer Protocol (FTP) is used to transfer files between computers, Telnet is used to connect (login) to remote computer systems, SMTP is the protocol used for email delivery, NNTP is used to deliver Usenet news.

based on empirical data collected at different Internet gateways, and it has been shown to produce realistic traffic patterns [23]. This traffic model is realized as a protocol called TRAFFIC, which sits at the top of the protocol graph generating work.

The traffic simulation protocol (TRAFFIC) starts conversations with inter-arrival times given by an exponential distribution. Each conversation can be of type Telnet, FTP, NNTP or SMTP; type-specific parameters are also specified. For example, FTP conversations are parameterized by the number of items to transmit, control segment sizes, and the individual item sizes. All of these parameters are obtained from *tcplib* and are based on probability distributions obtained from real traffic traces. Each of these conversations runs on top of its own TCP connection, so each conversation adapts to the network conditions by way of TCP's congestion control mechanisms.

One last feature of the traffic simulation protocol is that it is instrumented in such a way that the major traffic *characteristics* are independent of the underlying protocols and router congestion mechanisms—the detailed traffic itself is of course affected by these differences. What we mean by major traffic characteristics are the start times of each conversation, the type of conversation, and the parameters for that conversation. This feature allows us to compare the effects that changes—e.g., to the TCP implementation—have on the traffic. For example, it allows us to answer questions of the form "what are the effects on the routers and bottleneck utilization when we go from the one implementation of TCP to another?"

Finally, there are other protocols (everything is a protocol in the x-kernel) that model other types of transfers; for example, the MEGTEST protocol models TCP bulk transfers.

2.2.4 Related Work

Some of best known freely available network simulators are *REAL* [19], *Netsim* [10] and the recently released *ns* from LBL's Network Research Group. None of these simulators support common implementations of protocols, that is, direct execution.

Instead there is code to simulate the major characteristics of the protocol. One problem with this approach is that it misses some of the behavior present in the common protocol implementations. In particular, it is our experience that in evaluating the different BSD implementations of TCP, running the actual TCP code is preferred to running an abstract specification of the protocol; the latter approach is mostly useful for rapid experimentation.

For example, we have found the actual behavior of timers in the BSD implementations of TCP to be quite important. Specifically, the round-trip time (RTT) measurements are done with a clock that ticks every half a second, resulting in very coarse estimates of the RTT. This affects the real time behavior of the protocol because retransmit timeouts tend to be much longer than the real RTT. Many existing simulators use more accurate RTT measurements, resulting in retransmission timeouts that are much smaller than those that normally occur in BSD TCP. As a consequence, packet losses that result in timeouts have a lesser effect on the throughput in the simulator than they do in the BSD implementation.

It turns out that it would be very hard, if not impossible, to do a full implementation of TCP under either Netsim or REAL due to architectural constraints. As a result, neither simulator could be used to study the exact behavior of one of the common implementation of TCP (like the BSD implementations) to see if there are any problems with it.

Another important feature is that x-Sim protocols are just x-kernel protocols. One can use the simulator while implementing and debugging a new protocol, and when done, just move it to any x-kernel based system.

An alternative to simulation is emulation [1], a technique that uses workstations connected by a real network, with modifications to the operating system to simulate slower links and larger propagation delays. Under this framework, simulated time and real time are the same. Some of the advantages of emulation are that it runs real protocols, actual behavior like processing overheads need not be simulated, and a two minute simulation only takes two minutes to run. There are some disadvantages, however, such as the cost of the necessary hardware (to emulate a network with five hosts you need at least five workstations), the fact that you cannot emulate a host or a link that is *faster* than the actual hardware, and the fact that real time behavior of the software is tied to specific hardware. A final disadvantage has to do with the available timer resolution, which can result in artifacts affecting the accuracy of the emulation. For example, an emulator with timer accuracy of 1ms implies that one cannot emulate the exact delay involved in sending packets smaller than 500 bytes on a 500KB/s link (at 500KB/s it takes exactly 1ms to send 500 bytes).

Finally, we should not think of simulation versus emulation. Simulation, emulation, and tests on the actual network should all be considered part of the practitioner's repertoire. The confirmation of one technique's results with another technique serves an important function in the scientific process.

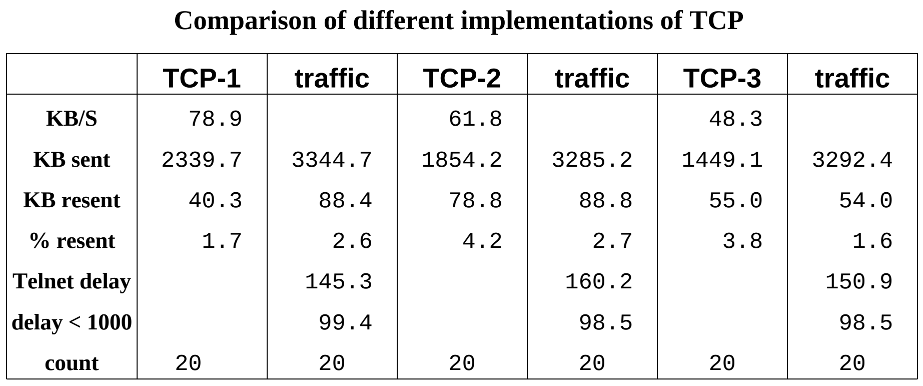
2.3 Visualization Tools

Running realistic simulations is only the first step, and not necessarily the most time consuming step, in the process of studying network behavior. Analyzing the results usually involves more work than running the simulations themselves, emphasizing the need for tools to aid in this analysis. The rest of the section describes some of the tools available as part of the x-Sim simulator package.

2.3.1 Coarse Grained Tools

These tools can be divided into two levels. At the high level, some of the protocols write summary information to a database at the end of each experiment. For example, TCP bulk transfers save the number of kilobytes successfully transmitted, the kilobytes resent, the number of TCP timeouts, the largest timeout, and so on. Routers save average link utilization, average queue sizes, number of packets and bytes lost.

This summary information can then be used for high level analysis of the network and its components. The coarse grained visualization tools are used to display different views of this information, allowing the user to focus on the relevant parts. For example, as part of our research we were interested in comparing the performance



Comparison of different implementations of TCP

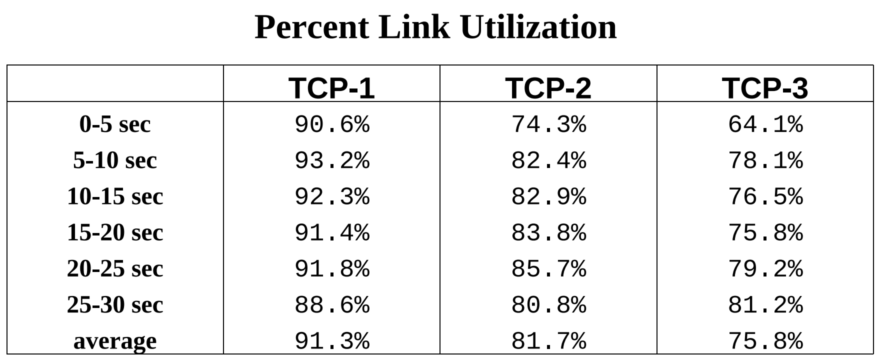
	TCP-1	traffic	TCP-2	traffic	TCP-3	traffic
KB/S	78.9		61.8		48.3	
KB sent	2339.7	3344.7	1854.2	3285.2	1449.1	3292.4
KB resent	40.3	88.4	78.8	88.8	55.0	54.0
% resent	1.7	2.6	4.2	2.7	3.8	1.6
Telnet delay		145.3		160.2		150.9
delay < 1000		99.4		98.5		98.5
count	20	20	20	20	20	20

Figure 2.4: Database output comparing TCP implementations.

of three implementations of TCP which we will refer to as TCP-1, TCP-2 and TCP-3. The simulation topology used is that on Figure 2.3. Two hosts, one on each ethernet, run the TRAFFIC protocol to simulate background traffic while the other two hosts run the MEGTEST protocol which does a TCP bulk transfer from one host to the other. The point-to-point link connecting the two ethernets has a maximum bandwidth of 200KB/s and it is the bottleneck since it is slower than the 1.25MB/s bandwidth of the ethernet networks.

Each simulation was uniquely specified by choosing one of the three TCP implementations and by a seed that was used to create a specific traffic pattern. Each simulation lasted 30 (simulated) seconds during which a bulk transfer shared the bottleneck link with background traffic. Sixty simulations were run, twenty with each TCP implementation using different traffic patterns. The twenty traffic patterns were the same for each TCP implementation.

Figure 2.4 shows the average of some of the information saved in the database files. Each pair of columns correspond to one of the TCP implementations: the first two are for TCP-1, the next two for TCP-2 and the final two for TCP-3. In each pair of columns, the left one—with the TCP implementation name as a heading—shows the averages corresponding to the bulk transfer, and the right one shows the averages for the background traffic. Hence, the bulk transfer in the TCP-1 implementation achieved an average throughput of 78.9 KB/s, and during the 30



Percent Link Utilization

	TCP-1	TCP-2	TCP-3
0-5 sec	90.6%	74.3%	64.1%
5-10 sec	93.2%	82.4%	78.1%
10-15 sec	92.3%	82.9%	76.5%
15-20 sec	91.4%	83.8%	75.8%
20-25 sec	91.8%	85.7%	79.2%
25-30 sec	88.6%	80.8%	81.2%
average	91.3%	81.7%	75.8%

Figure 2.5: Database output comparing TCP implementations.

seconds it transfered an average of about 2340KB. An average of 40.3KB had to be retransmitted, probably due to losses. This 40.3KB represents 1.7% of the data successfully transmitted.

The background traffic in the TCP-1 implementation transfered 3345KB and retransmitted 2.6% of the data. An important measure is the average RTT for Telnet traffic (Telnet delay in ms); this measure is an indication of how the response of the network is affected by the traffic. The next value is the percent of Telnet packets with a RTT less than 1 second. From this table we see that TCP-1 is able to achieve a much higher throughput without negatively affecting the background traffic, while at the same time retransmitting less data.

Figure 2.5 shows the average bottleneck link utilization during the 30 second simulation. It shows the reason why TCP-1 does so much better than both TCP-2 and TCP-3: TCP-1 makes better use of the bottleneck link, achieving an average of 91.3% link utilization, as compared to 81.7% utilization by TCP-2 and 75.8% by TCP-3

The last type of output from the database file is one that shows individual, rather than average, values. It is important to always look at individual values to check that the average value is not being affected by a spurious result. Just as importantly, results that are out of the ordinary may point to protocol problems. Figure 2.6 shows the individual values for the throughput and kilobytes resent for

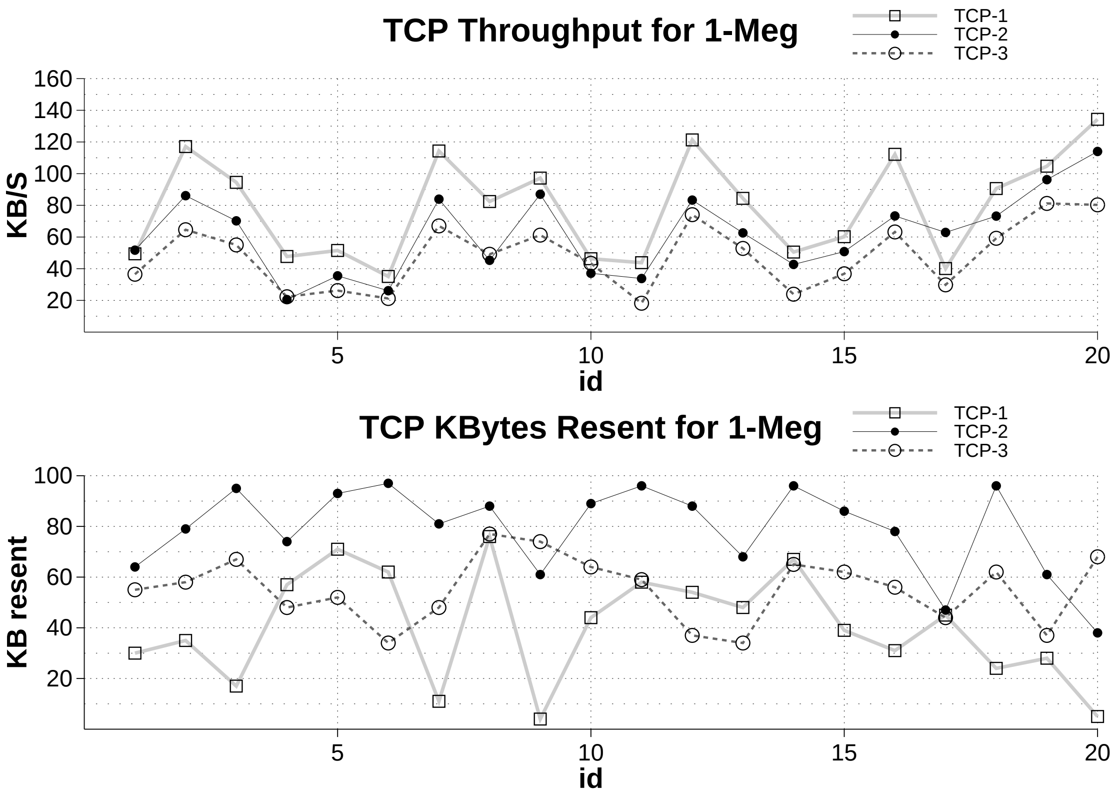


Figure 2.6: Database output comparing TCP implementations.

the bulk transfers.

2.3.2 Fine Grained Tools

The database files contain coarse information about the simulations. Often it is useful to be able to do detailed examination of particular network components. For this purpose, some protocols and routers can save detailed information into tracing files. These traces are analyzed and displayed using a set of common tools which are part of the x-Sim package. The most detailed traces are obtained from the more complex protocols, and they have proven invaluable in the analysis and understanding of these protocols. For example, the x-kernel/x-Sim versions of the different TCP implementations have been augmented with calls to trace all the relevant state of the protocol, such as the size of the different windows and buffers,

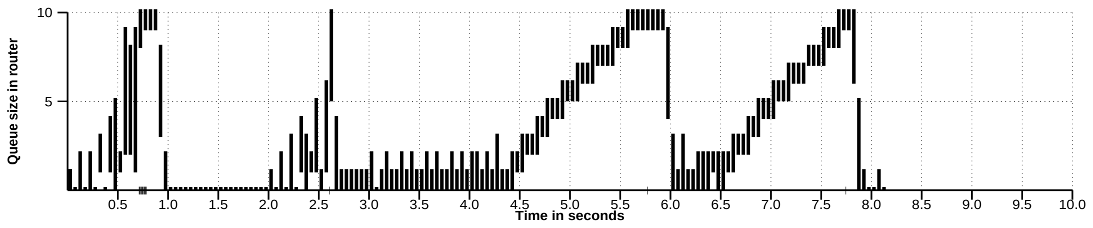


Figure 2.7: Graph showing router's queue size.

the time when packets are either sent or received, information about retransmitted data, and so on.

These traces can be examined either graphically, showing the main characteristics of a protocol or router, or in full detail as a list of each trace event. The common procedure is to first examine the traces graphically, then if anything interesting or unusual shows up, to examine the traces in full detail.

As mentioned earlier, the most complicated traces are those belonging to protocols, particularly TCP. These traces will be explained in the next chapter when we describe TCP. Here we describe two of the traces saved by routers. The first one saves the queue size, or more specifically, it saves the minimum, maximum and average queue sizes during intervals of a specified length. For example, Figure 2.7 shows the queue size during a TCP bulk transfer. Each bar represents a 100ms interval, the bottom of the bar indicates the minimum queue size and the top the maximum queue size during each 100ms intervals. The average queue size is not displayed in this example.

The second type of trace information that routers save is the average output

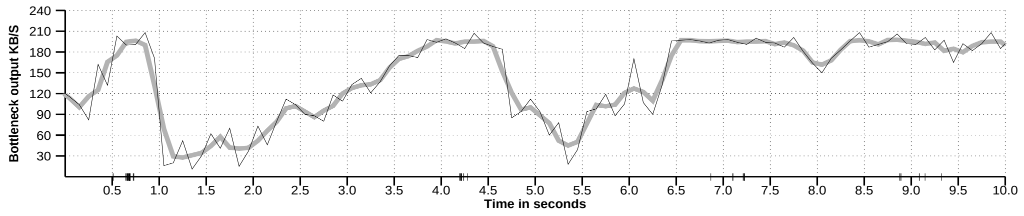


Figure 2.8: Graph showing router's average output bandwidth for a specified link.

bandwidth of specified links. This information is similar to that shown in Figure 2.5 except that instead of presenting 5 second averages it presents averages over user specified intervals. Figure 2.8 shows the output bandwidth averaged over 100ms intervals (thin lines) and 300ms intervals (thick line). The available bandwidth for this link is 200KB/s, and it is reached a couple of times during this particular simulation. The small hash marks on the x-axis indicate a packet was dropped at that time as a result of queue overflow.

CHAPTER 3

Transmission Control Protocol

We know from experience that internets with no congestion control behave badly under high loads [13]. Since most of the traffic over the Internet comes from TCP connections, TCP's congestion control mechanisms play a pivotal role in maintaining the health of the Internet. This chapter outlines the main characteristics of TCP, describes its congestion control mechanisms, and provides an insight into its general behavior.

This chapter is organized as follows. First, we give a brief description of some of the early applications that were the catalysis for the invention of computer networks. It is easy to understand TCP's main characteristics in this context. Second, we describe some of the most important mechanisms in TCP, focusing in those dealing with reliability and congestion control. Third, we provide some insight into TCP's congestion control mechanisms by analyzing graphs depicting some of the most important internal state of TCP.

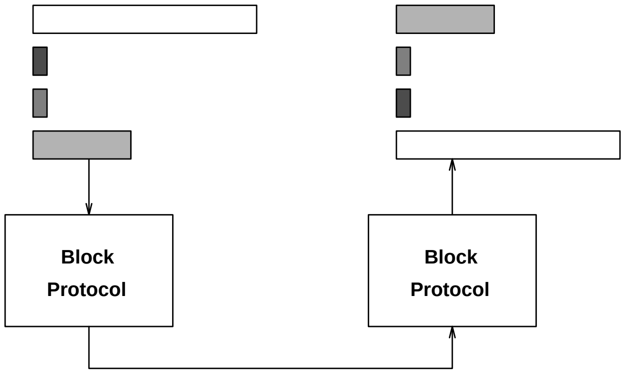


Figure 3.1: Block Protocols maintain block boundaries.

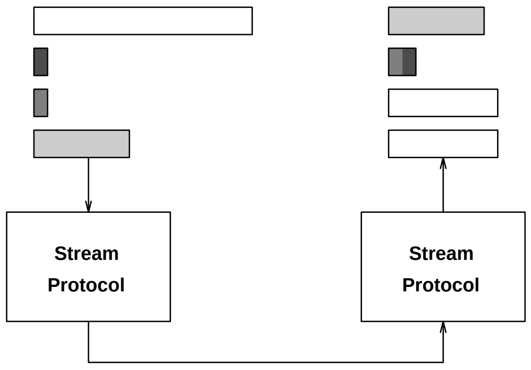


Figure 3.2: Stream Protocols do not maintain block boundaries.

3.1 Background

As mentioned in Chapter 1, IP forms the infrastructure that integrates a collection of heterogeneous networks. It is a block (packet) based protocol; IP accepts blocks of data and either delivers the whole block or nothing at all, in such a way that the block boundaries are known to the receiving application (or higher level protocol); see Figure 3.1.

Once computers had the ability to communicate with each other, it did not take long for the first network applications to surface. Among these first applications was Telnet. Telnet is a computer program that allows a user to *connect* to a remote computer; it mimics the behavior of a terminal connected directly to the remote computer. Each time the user presses a keyboard key, the ASCII representation of the key is sent to the remote computer. This computer in turn echos the key back to the user's computer, together with any output that may have resulted from the keystrokes sent to the remote computer. Telnet is an example of an application that requires a *duplex* (two way) communication channel.

In this scenario, there are no inherently important block boundaries. Furthermore, since Telnet involves the transmission of very small units of data—and there is always some overhead when transmitting packets due to packet header—it makes sense to try and send as much information as possible on each packet. A *stream based* protocol does not preserve block boundaries, so the protocol has the option of aggregating blocks of data before sending them (see Figure 3.2). The term *segment* is used to refer to a block of data passed by a stream protocol to a lower protocol.

A common requirement for most types of communications is *reliability*. For example, one expects email to arrive at the destination without any missing parts. As previously mentioned, IP is not a reliable protocol. It makes perfect sense not to burden a protocol whose primary purpose is the integration of heterogeneous networks with reliability. There are high costs associated with reliability both in terms of code complexity and storage—a reliable protocol must keep a copy of the data it sends until the protocol knows the data has been successfully received, in case the protocol needs to send the data again.

Stream based protocols also need to be ordered since there is no way for the receiving higher level protocol (or application) to know the position of a block of data just received. The usual way of communicating the position of a block is by writing this information at the beginning of the block—in the header. Since a stream based protocol may break a block into smaller components, it is possible to receive parts of the block without the block header.

Another important issue to consider when dealing with computer communications is how to deal with sources that are able to send at a higher rate than the destination can deal with. This mismatch can be due to: (1) the source may be a more powerful (higher instruction execution rate) computer than the destination, or, (2) the destination may have to perform some time consuming task—such as in client-server computing. *Flow control* is the term used to denote some sort of synchronization between the source and destination, so that the source does not overrun the destination.

It is important to distinguish the differences between flow and congestion control. Flow control mechanisms prevent the source from overrunning the receiver. They are usually simple to design since there are only two entities involved. Congestion, on the other hand, occurs at intermediate nodes (the routers) and may involve multiple sources. As a result, congestion control mechanisms are more complex and harder to design.

A useful analogy is to think of a connection's path as a water pipe. Using all of the bandwidth available to the connection is equivalent to keeping the *pipe full.*

TCP is a protocol designed to provide the communication paradigms just described. It is a connection based protocol that provides an ordered, stream based, reliable duplex communication channel with flow and congestion control. It runs on top of IP, so it has at its disposal the full set of network services provided by IP. The next subsections present more detailed descriptions of how reliability, flow and congestion control are implemented in TCP.

3.2 Reliability

Having reliability as a goal implies the possibility of having to retransmit some of the data already sent. Thus data needs to be buffered until it is known to have been successfully received on the other end. Successful reception implies not only that a datagram arrived, but also that it arrived uncorrupted. Hence there is a need for mechanisms that insure that the data was not corrupted and that it was successfully received. A checksum is used to validate both the TCP header and data.

Two possible mechanisms for informing the source that data arrived successfully are: (1) having the receiver send a *positive* acknowledgment (ACK) announcing that a block of data arrived safely, or (2) having the receiver send a *negative* acknowledgment (NACK) announcing that a block of data is missing.

To specify that a block of data has been received or that it is missing, one must have a naming scheme. The usual approach used in block based protocols is to number each block of data in the order in which it was passed to the source. This approach can be expanded to cover stream based protocols by thinking of each unit of data—bytes in this case—as a block. TCP uses a *sequence number* to uniquely name each byte sent over the communication channel. These sequence numbers are used by the receiver to correctly order segments that may be received out of order and to eliminate duplicates.

TCP uses positive rather than negative acknowledgments. Furthermore, to conserve packet space and for reliability reasons, TCP's designers decided on *cumulative* acknowledgments. Acknowledging a block of data requires two numbers to code the sequence numbers of the beginning and end of the block. A cumulative ACK acknowledges all data up to a given sequence number, so only one number is needed to acknowledge a block of data—assuming all previous blocks have been received. A further advantage of cumulative ACKs is the reliability of this approach; the loss of an ACK is not very damaging in the common case when later ACKs will be received, as the information on the lost ACK will be contained in succeeding ACKs.

Cumulative acknowledgments inform the source that segments were successfully received; an additional mechanism is still needed to determine when segments are lost. TCP uses a timeout mechanism to detect and then retransmit lost segments. If an ACK is not received by the source within a timeout interval, the data is retransmitted.

The original protocol specification does not require any particular mechanism (algorithm) for determining the timeout interval. However, it gives an example retransmission timeout procedure based on the measured round trip time (RTT)—the elapsed time between sending the segment and the reception of the ACK. The retransmit timeout procedure in use today is based on both round trip time and variance estimates computed by sampling the time between when a segment is sent and an ACK arrives [13].

3.3 Flow Control

Flow control is the mechanism used to prevent a faster host from overruning a slower host. TCP uses a window based flow control mechanism by which the receiving TCP allows the source to send only as much data as the receiver can buffer. The receiver controls the flow of data by *advertising* the amount of free space in its receive buffer. This space decreases as new packets arrive and increases as the application reads more data. Each acknowledgment the receiver transmits to the source contains the total number of free bytes in its buffer space; it is known as the *advertised window*.

A convenient way to visualize the dynamics of the advertised window as seen by the source is shown in Figure 3.3. The thick gray line represents the advertised

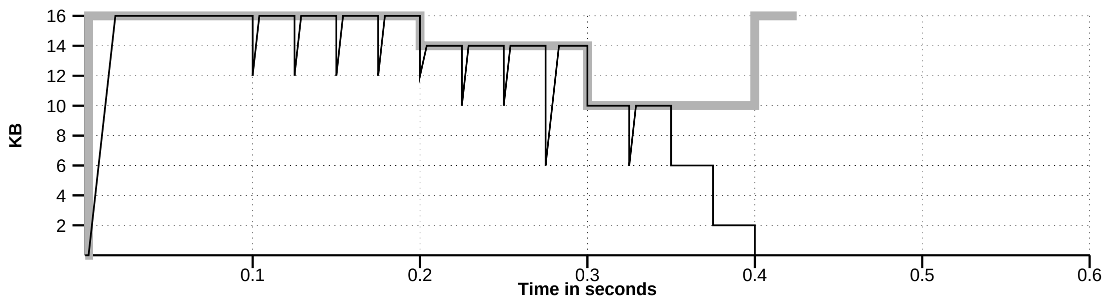


Figure 3.3: Window flow control.

window, which is set to 16KB as the connection is established. The thin black line represents the number of bytes that have been sent and have not yet been acknowledged (UNACK_COUNT). Its value should not exceed the advertised window, as doing otherwise could result in the loss of packets due to overflowing the receiver's buffer.

The source quickly transmits 16KB of data, the full amount allowed by the advertised window. At 100ms, the UNACK_COUNT line decreases to 12KB, signifying that the receiver has acknowledged the first 4KB that were transmitted (acknowledgments are cumulative). Note that the destination is not bound to send an acknowledgment immediately after it receives a packet. The destination can delay sending the ACK for a short time in case more packets arrive (reducing the number of ACKs it sends) or to give the application an opportunity to read the newly arrived data.

The fact that the advertised window does not decrease at 100ms implies that the 4KB was read by the application—freeing the receiver's buffer space—before the ACK was sent. This same behavior is repeated at 125, 150 and 175ms.

At 200ms the UNACK_COUNT line once again decreases to 12KB, but unlike previous times, the advertised window also decreases (to 14KB), signifying that the application had only read 2KB at the time the acknowledgment was sent (about 50ms earlier). The other 2KB are held in the receiver's buffer and the source can only transmit 2KB at this time. At 275ms the UNACK_COUNT line decreases from

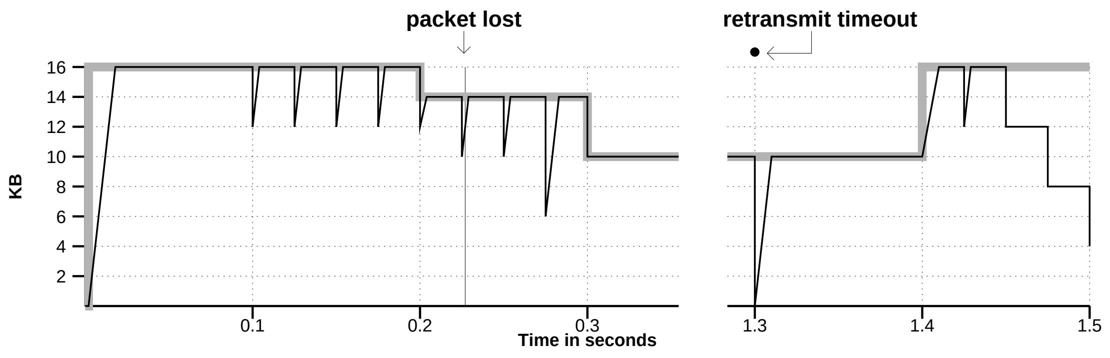


Figure 3.4: Window flow control.

14KB to 6KB and the source immediately transmits 8KB of data since the advertised window stayed at 14KB. Between 350ms and 400ms the UNACK_COUNT line decreases to zero indicating that the sending application has no more data to send for the time being.

This type of graph can also be used to visualize the dynamics of the reliability mechanism. Figure 3.4 introduces two more symbols to the previous type of graph. A packet loss, which in this example occurs at 225ms, is represented by a vertical line spanning the whole graph. More specifically, vertical bars mark the time when a packet is sent which is later retransmitted, likely because it was lost. Since TCP is an ordered and reliable protocol, it does not allow the application to read any data beyond the lost packet.

Since TCP uses cumulative ACKs, it cannot acknowledge any packets containing data following the lost packet[1]. As a result, UNACK_COUNT stays at the same level as the advertised window; preventing the source from sending any more packets. This situation is remedied at a later time, 1.3 seconds in this example, when a retransmit timeout occurs, represented by the dot at the top of the graph, which restarts the process. The source assumes *all* unacknowledged data has been lost and resends it.

[1]TCP does not decrease its advertised window as a result of data it cannot acknowledge.

3.4 Congestion Control

The original version of TCP did not contain any congestion control mechanisms. Congestion is only mentioned in the reliability section of the protocol specification as a possible cause of packet losses. This lack of congestion control eventually resulted in what is called *congestion collapses* as a result of the increasing traffic on the Internet. The term is used to indicate a persistent state of congestion, where the actual throughput of the connections involved is almost zero.

The dynamics of congestion collapse can be deduced from Figure 3.4. Once a link becomes congested, a few of the connections whose path contains the congested link are likely to lose one or more packets. This congestion can occur by increasing the number of connections going through the link. All connections that lose packets timeout and start retransmitting at around the same time (the delay before the retransmit timer fires is a function of the connection's RTT). Figure 3.4 shows that when the retransmit timeout occurs, the connection resends all the data it had previously sent but not gotten an acknowledgment for. The link becomes even more congested as a result of one or more connections quickly sending so much data on an already congested link. It is not hard to imagine how the process could continue indefinitely once it got started.

3.4.1 Jacobson's Original Proposal

In 1988, Van Jacobson published a paper [13] in which he describes mechanisms to prevent congestion collapse by detecting and controlling congestion. During periods when there is no congestion—and therefore no losses—there is a directly proportional relationship between UNACK_COUNT and throughput:

$$Throughput = \frac{UNACK\_COUNT}{RTT}$$

As UNACK_COUNT increases, the throughput should also increase. When UNACK_COUNT decreases, so should the throughput. Jacobson introduced the *congestion window* as a way of dynamically controlling UNACK_COUNT. The amount

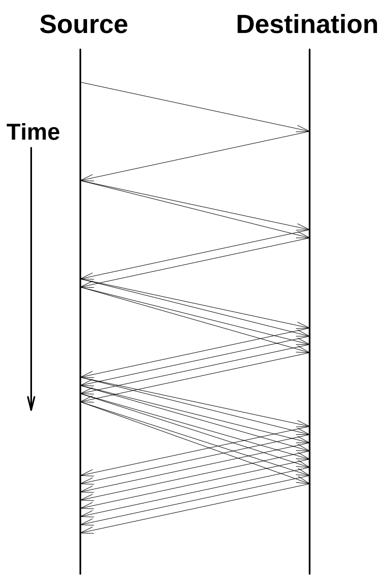


Figure 3.5: Timeline of Slow-start.

of UNACK_COUNT that is allowed is now bounded by both the advertised window and by the congestion window. The congestion window is allowed to change dynamically with the purpose of controlling congestion. He described two mechanisms to control the the size of the congestion window: (1) slow-start and (2) linear-increase/multiplicative-decrease.

Slow-start is used to control the rate by which UNACK_COUNT increases at the beginning of the connection, or after a retransmit timeout. When executing the slow-start mechanism, the congestion window is initially set to the size of the largest packet TCP is allowed to send (*max-packet*). Each time an acknowledgment is received, the congestion window is increased by *max-packet*. If the receiver is acknowledging each packet, then the congestion window will double in size during each RTT, resulting in exponential growth of the congestion window.

Figure 3.5 shows the dynamics of slow-start. The source starts by sending one packet to the destination, which sends an acknowledgment in return. When the source gets the acknowledgment, it immediately sends two packets. The destination once again sends an acknowledgment immediately after receiving each packet. Each of these two ACKs results in two packets sent by the source, and so on.

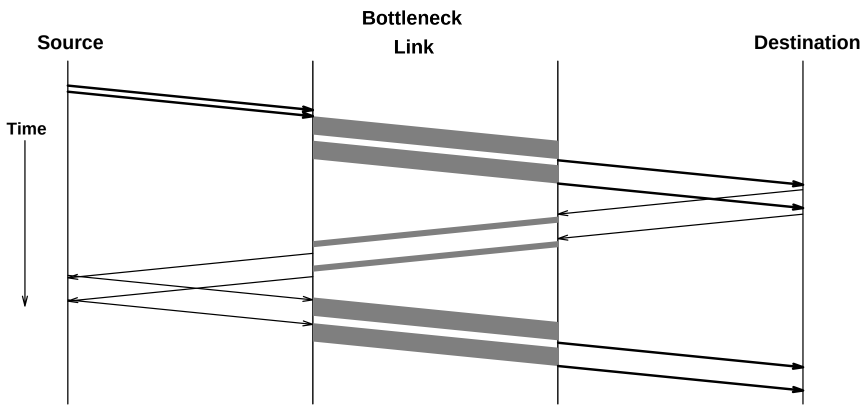


Figure 3.6: Packets spaced by slow link.

Even though the source sent two packets back-to-back, they are spaced out when they arrive at the destination. The reason for this is shown graphically in Figure 3.6. In this figure, the thickness of a line is inversely proportional to the speed of the link it traverses—the thickness of the line represents the time needed to insert the packet into the link. Initially, two packets leave the source close together, on a fast link. When they get to the bottleneck link it takes longer to insert each packet into the link, either because it is a slower link or because other packets are waiting in a queue to be sent over this link. For this discussion it is assumed that the bottleneck link is slower, so it takes longer to put each packet into the link, and the time between each packet increases. When the packets leave the slower link and go into a faster link, the time difference between the packets is preserved. If an acknowledgment is sent immediately after each packet arrives, the acknowledgments will carry the time difference back to the source (assuming there are no queues at the routers). If the source now sends one packet as it receives each acknowledgment, they will be sent at the rate of the bottleneck link.

The exponential growth of the congestion window during slow-start continues until one of three things happens: (1) it reaches the size of the advertised window, (2) it reaches a pre-established value (*threshold window*), or losses are detected. Consider the second case in more detail: at the beginning of the connection, the threshold window is usually set to 64KB. However, when slow-start is used after a

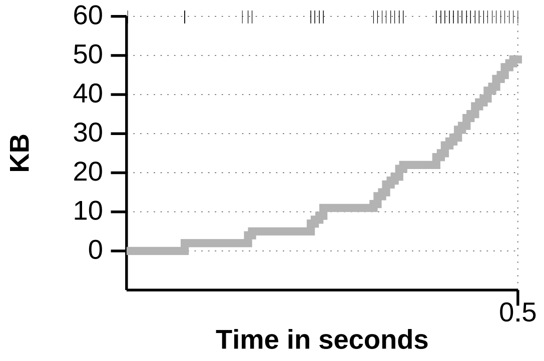


Figure 3.7: Slow-start: Congestion window size versus time.

retransmit timeout, the connection has an idea of the available bandwidth and the threshold window is set in such a way so as to stop the exponential growth right before the available bandwidth is reached.

Figure 3.7 shows the dynamics of slow-start by plotting the congestion window as a function of time. The tick marks at the top of the graph are used to indicate the time when a packet is sent. Notice how the congestion window increases exponentially and how the tick marks seem to be nicely spaced. However, since packets are sent back-to-back in pairs, each tick mark is really two overlapping tick marks.

After slow-start terminates, the linear-increase and multiplicative-decrease mechanism comes into play. This mechanism increases the congestion window by one *max-packet* on each RTT (a linear increase) and decreases the congestion window by half (a multiplicative decrease) whenever congestion is detected. *TCP interprets any loss as a sign of congestion.* The coarse behavior—without the slow-starts—of the linear-increase and multiplicative-decrease mechanism is shown in Figure 3.8.

The exact mechanism is as follows. When congestion is detected, the threshold window is set to half the size of the congestion window. Next, the congestion window is set to one. All unacknowledged packets are assumed lost, and slow-start is used to increase the size of the congestion window exponentially until it reaches the size of the threshold window.

A mechanism is *efficient* if it results in a full utilization of the available band-

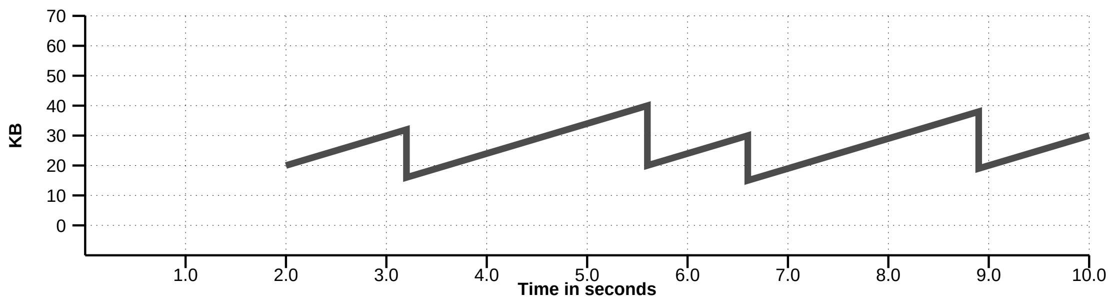


Figure 3.8: TCP trace: Congestion window size versus time.

width. It is *fair* if it results on all connections getting an equal share of the bandwidth. An analytical study by Chiu and Jain [4] has shown that linear-increase and multiplicative-decrease mechanisms converge to an efficient and fair state, whereas linear-increase and linear-decrease mechanisms can converge to efficiency but not fairness.

3.4.2 Fast Retransmit and Fast Recovery Mechanisms

The Fast Retransmit mechanism was developed by Jacobson to help reduce the dependency on the retransmit timeout by attempting to detect losses earlier. The BSD implementations of TCP use a coarse clock, with a granularity of 500ms, to measure RTT and to implement the retransmit timer. The coarseness inherent in this mechanism results in timeout intervals which are much longer than necessary. For example, during a series of tests on the Internet, it was found that for losses that resulted in a timeout, it took TCP an average of 1100ms from the time it sent a packet that was lost until it timed out and resent the packet, whereas less than 300ms would have been the correct timeout interval had a more accurate clock been used.

With the Fast Retransmit mechanism, TCP not only retransmits when a coarse-grained timeout occurs, but also when it receives n duplicate ACKs (n is usually 3). TCP sends a duplicate ACK whenever it receives a new segment that it cannot acknowledge because it has not yet received all the previous segments [25]. In

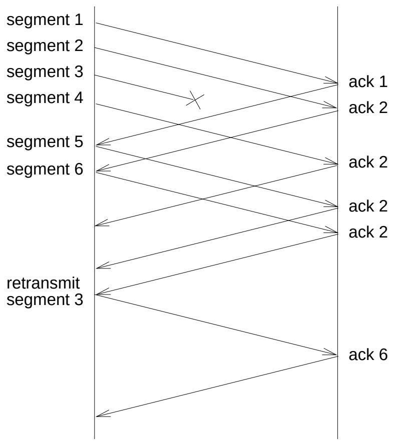


Figure 3.9: Example time-line of Fast Retransmit mechanism.

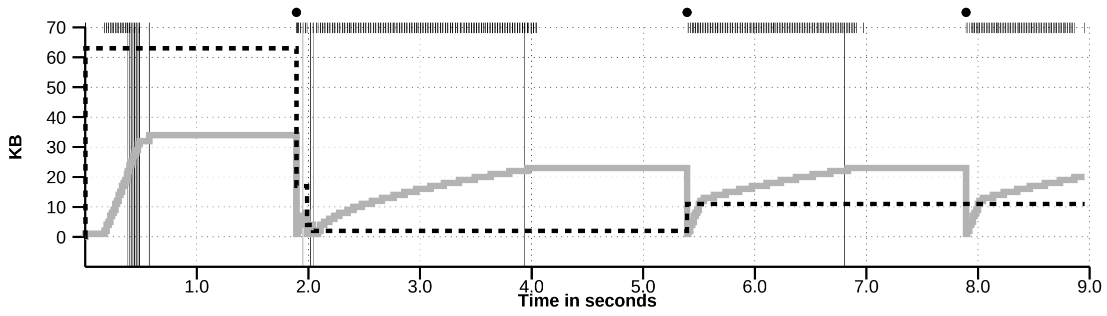


Figure 3.10: Behavior of TCP without Fast Retransmit.

effect, we can think of this mechanism as sending implicit *negative* acknowledgments (NACKs).

For example, if Tahoe receives segment 2 but segment 3 is dropped, it will send a duplicate ACK for segment 2 when segment 4 arrives, again when segment 5 arrives, and so on (see Figure 3.9). When the source sees the third duplicate ACK for segment 2 (the one sent because the receiver had gotten segment 6) it retransmits segment 3. The Fast Retransmit mechanism is very successful—it prevents more than half of the coarse-grained timeouts that occur on TCP implementations without it.

Figure 3.10 shows the behavior of TCP without the Fast Retransmit mechanism when it is the only connection active in the network (i.e., there is no other traffic).

The network configuration used for this experiment is shown in Figure 3.11, and the only traffic consisted of Host 1a doing a TCP transfer to Host 1b (all other hosts were inactive). The thick dotted line is the threshold window and the gray line is the congestion window. The congestion window increases exponentially at the beginning of the connection until losses are detected at around 300ms. The solid vertical lines running the whole height of the graph indicate when a segment that is eventually retransmitted was originally sent. Nothing is transmitted for a period of over a second, as shown by the lack of tick marks at the top of the graph, until the retransmit timer fires at around 2 seconds. The dots at the top of the graph are used to indicate that a retransmit timer fired at this time. The threshold window is set to half the value of the congestion window so slow-start will terminate when the congestion window reaches half its previous value. In this particular test, losses are detected which further reduce the threshold window. Between 2 and 4 seconds one sees a linear increase of the congestion window, until a loss occurs a little before 4 seconds. After the retransmit timer expires at 5.3 seconds, the threshold window is set to half the value of the congestion window. The congestion window is then set to the size of one packet, increasing exponentially until it reaches the threshold window. This periodic behavior of exponential growth, followed by linear growth, followed by a retransmit timeout, is repeated while there is data to send.

Figure 3.12 shows the behavior of TCP Tahoe, an implementation with the Fast Retransmit mechanism. Note that all retransmit timeouts have been eliminated except for the first one. The Fast Retransmit mechanism can eliminate most re-

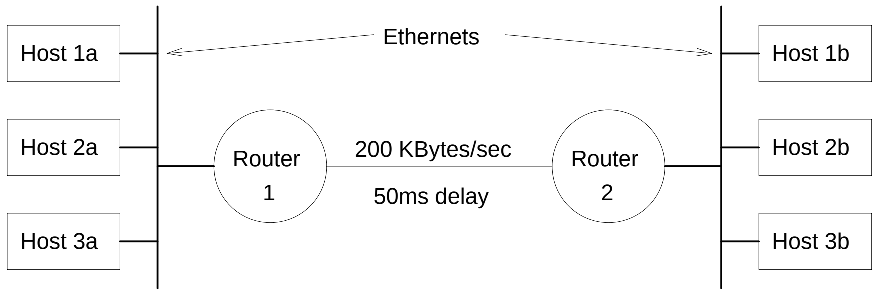


Figure 3.11: Network configuration for simulations.

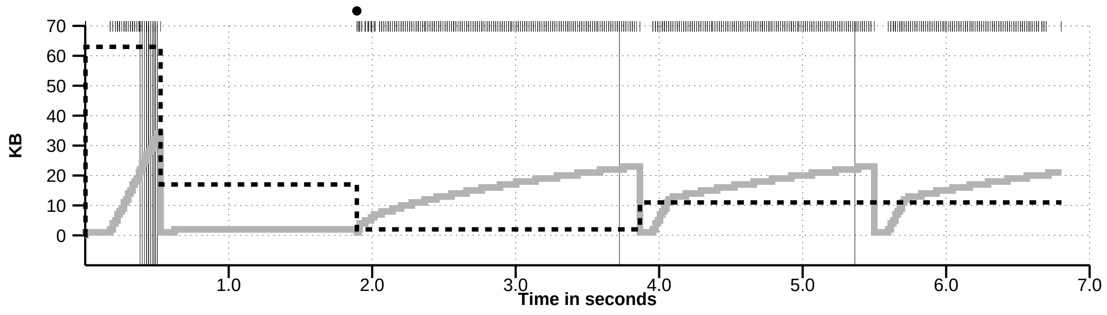


Figure 3.12: Behavior of TCP with Fast Retransmit.

transmit timeouts as long as only one or two packets are lost during a period of time equal to the RTT of the connection. Note that after each loss there is a small period of time when no packets are transmitted—as seen by the lack of tick marks on the top of the graph. These pauses are the result of slow-start, since only one packet is sent during the first RTT.

Jacobson realized that there was no need to use slow-start when the losses are detected by the Fast Retransmit mechanism. Instead of slow-start, TCP could the arrival of duplicate acknowledgments to trigger the transmission of new packets with the right spacing between them (see Figure 3.6). In particular, the Fast Recovery mechanism decreases the congestion window by half (rather than setting it to one), waits until it detects the arrival of half the expected number of duplicate acknowledgments ($UNACK\_COUNT/(2 \times max\_seg)$), and then uses the incoming duplicate acknowledgments to strobe new packets into the network. In other words, with the Fast Recovery mechanism, TCP is able to slow down directly to the reduced rate rather than slowing down to one packet per RTT, and then using slow-start to increase the rate to the appropriate value.

The Fast Recovery mechanism is included in the Reno implementation of TCP. Figure 3.13 shows the behavior of Reno and also includes graphs showing the sending rate (middle graph) and the queue size at the bottleneck router (bottom graph). This figure is explained in the next section.

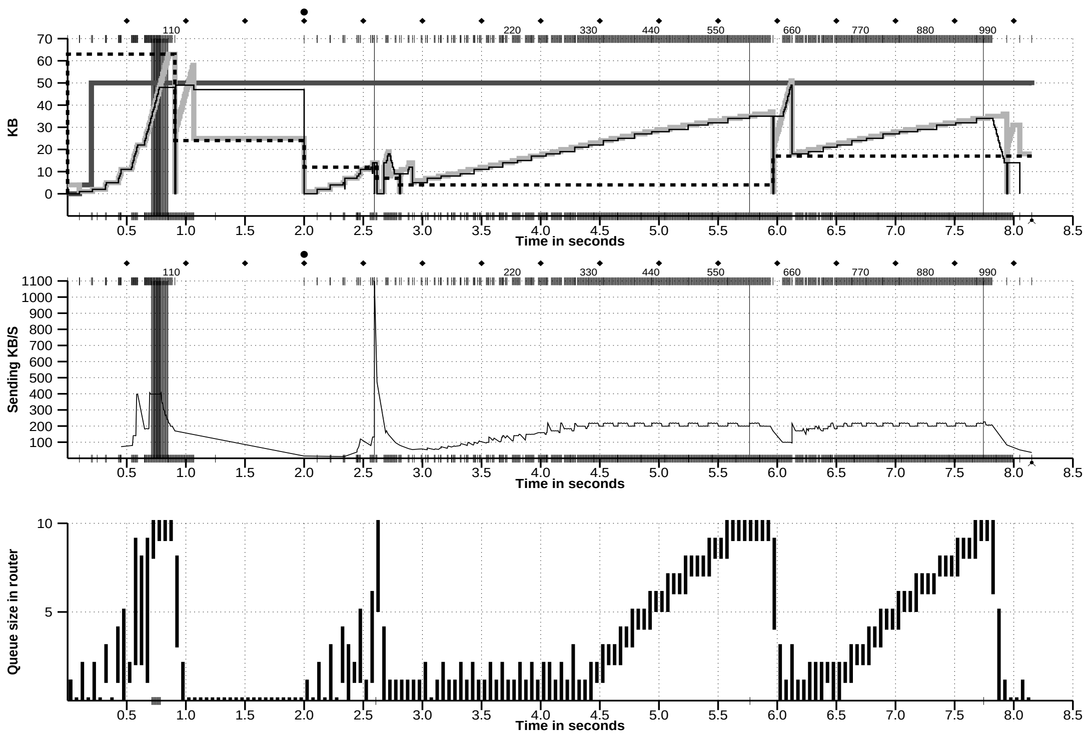


Figure 3.13: Behavior of TCP with Fast Retransmit and Fast Recovery.

3.5 General Graph Description

In the early stages of this research, it became clear that there was a needed for good facilities to analyze the behavior of TCP. As a result, code was added to the x-kernel to trace the relevant changes in the connection state. Particular attention was paid to keeping the overhead of this tracing facility as low as possible, so as to minimize the effects on the behavior of the protocol. Specifically, the facility writes trace data to memory, dumps it to a file only when the test is over, and keeps the amount of data associated with each trace entry small (8 bytes).

Various tools were developed to analyze and display the tracing information. The rest of this section describes one such tool that graphically represents relevant features of the state of the TCP connection as a function of time. This tool outputs multiple graphs, each focusing on a specific set of characteristics of the connection state. Figure 3.13 shows an example. What follows is a detailed explanation of how

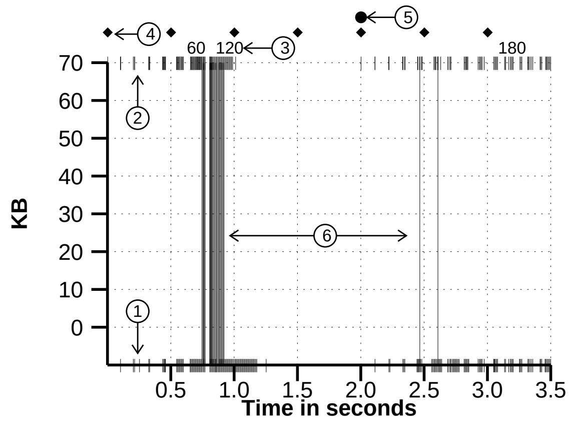


Figure 3.14: Common elements in TCP trace graphs.

to interpret such a graph.

First, all TCP trace graphs have certain features in common, as illustrated in Figure 3.14. The circled numbers in this figure are keyed to the following explanations:

1. Hash marks on the x-axis indicate when an ACK was received.

2. Hash marks at the top of the graph indicate when a segment was sent.

3. The numbers on the top of the graph indicate when the n^{th} kilobyte (KB) was sent.

4. Diamonds on top of the graph indicate when the periodic coarse-grained timer fires. This does not imply a TCP timeout, just that TCP checked to see if any timeouts should happen.

5. Circles on top of the graph indicate that a coarse-grained timeout occurred, causing a segment to be retransmitted.

6. Solid vertical lines running the whole height of the graph indicate when a segment that is eventually retransmitted was originally sent, presumably because

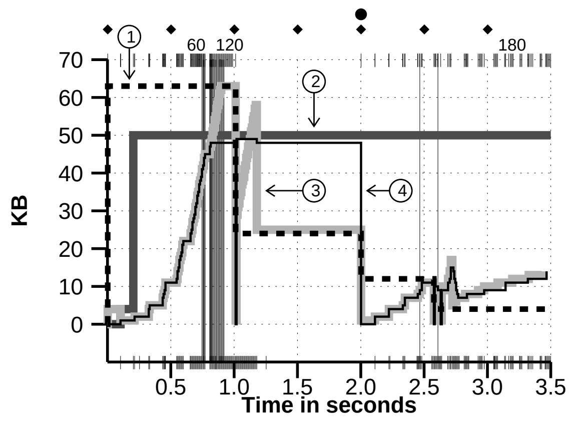


Figure 3.15: TCP windows graph.

it was lost.[2] Notice that several consecutive segments are retransmitted in the example.

In addition to this common information, each graph depicts more specific information. The middle graph in Figure 3.13 is the simplest—it shows the average sending rate, calculated from the last 12 segments. The bottom graph was described in Chapter 2, it shows the behavior of the queue at the bottleneck router. The top graph in Figure 3.13 is more complicated—it gives the size of the different windows TCP uses for flow and congestion control. Figure 3.15 shows these in more detail, again keyed by the following explanations:

1. The dashed line gives the threshold window. It is used during slow-start, and marks the point at which the congestion window growth changes from exponential to linear.

2. The dark gray line gives the send window. It is the minimum of the source's buffer size and receiver's advertised window, and defines an upper limit to the number of bytes sent but not yet acknowledged.

[2]For simplicity, we sometimes say a segment was lost, even though all we know for sure is that the source retransmitted it.

3. The light gray line gives the congestion window. It is used for congestion control, and limits the number of bytes sent but not yet acknowledged (UNACK_COUNT). More specifically, a packet will not be sent if the resulting UNACK_COUNT is larger than the congestion window.

4. The thin line gives the actual number of bytes in transit; that is, those sent but not yet acknowledged (UNACK_COUNT).

The graphs just described are obtained from tracing information saved by the protocol, and are thus available whether the protocol is running in the simulator or over a real network.

3.6 Detailed Graph Description

To assist the reader to develop a better understanding of the graphs used throughout this paper, and to gain a better insight of Reno's behavior, this section contains a detailed description of one of these graphs. Figure 3.15 is a trace of Reno when there is other traffic through the bottleneck router. The numbers in parenthesis refer to the type of line in the graph.

In general, output is allowed while the UNACK_COUNT (4) is less than the congestion window (3) and less than the send window (2). The purpose of the congestion window is to control congestion. The send window is used for flow control; it prevents data from being sent when there is no buffer space available at the receiver.

The threshold window (1) is set to some maximum value—e.g. 64KB— at the beginning of the connection. Soon after the connection is started, both sides exchange the size of their receive buffers, and the send window (2) is set to the minimum of the source's send buffer size and the receiver's advertised window size.

The congestion window (3) increases exponentially while it is less than the threshold window (1). At 0.75 seconds, losses start to occur, as indicated by the tall vertical lines. More precisely, the vertical lines represent segments that are later retransmitted, usually because they were lost. At around 1.0 second, a loss is detected after receiving 3 duplicate ACKs and Reno's Fast Retransmit and Fast Recovery mechanisms go into action. The purpose of these mechanisms is to detect losses before a retransmit timeout occurs, and to continue transmitting—although at a reduced rate—while recovering from these losses.

The congestion window (3) is set to the maximal allowed segment size for this connection and the UNACK_COUNT is set to zero momentarily, allowing the lost segment to be retransmitted. The threshold window (1) is set to half the value that the congestion window had before the losses. TCP assumes that this is a safe level, and that losses will not occur at this window size.

The congestion window (3) is also set to this value after retransmitting the lost segment, but it increases with each duplicate ACK. Since the receiver sends a duplicate ACK when it receives a segment that it cannot acknowledge because it has not received all previous data, the reception of a duplicate ACK implies that a packet has left the network.

This implies that the congestion window (3) will reach the UNACK_COUNT (4) when half the data in transit has been received at the other end. From this point on, the reception of any duplicate ACKs will allow a segment to be sent. This prevents the "pipe" from becoming empty, and allows TCP to continue sending at half its previous rate.

Since losses occurred at the previous value, it is assumed that the available bandwidth is now only half its previous value. Earlier versions of TCP immediately begin the slow-start mechanism when losses are detected. This resulted in the "pipe" becoming almost empty and then filling up again. Reno's mechanism allows the "pipe" to stay full.

At around 1.2 seconds, a non-duplicate ACK is received, and the congestion window (3) is set to the value of the threshold window (1). The congestion window was temporarily inflated when duplicate ACKs were received as a mechanism to keep the pipe full. When a non-duplicate ACK is received, the congestion window is reset to half the value it had when losses occurred.

Since the congestion window (3) is below the UNACK_COUNT (4), no more data can be sent. At 2 seconds, a retransmit timeout occurs—see black circle on top—and data starts to flow again. The congestion window (3) increases exponentially while it is below the threshold window (1). A little before 2.5 seconds, a segment is sent that will later be retransmitted. Skipping to 3 seconds, one can see the congestion window (3) increasing linearly since it is above the threshold window (1).

CHAPTER 4

New Approach to Congestion Detection and Avoidance

The congestion detection and control mechanisms in TCP Reno—and all other current implementations of TCP—use the loss of segments as a signal that there is congestion in the network. These implementations have no mechanisms to detect the incipient stages of congestion—before losses occur—so that no attempt is made to prevent loss. These mechanisms are reactive, rather than proactive, in this respect.

A preferable mechanism is one that detects the early stages of congestion, such as increased queue sizes at the bottleneck routers, so that it can slow down the source and avoid the losses associated with the later stages of congestion. Such a mechanism is described in this chapter.

The chapter starts by providing an analysis of the behavior of Reno's congestion control mechanism. This is followed by a description of previously proposed congestion avoidance mechanisms. Next, it describes a new congestion detection and avoidance mechanism which forms the cornerstone of a new TCP implementation. We call this new implementation TCP Vegas, thereby continuing the practice started by the BSD distributions of Unix of naming the releases after cities in Nevada. The chapter concludes by describing other performance improving mechanisms which are also part of the Vegas implementation.

4.1 Analysis of TCP's Congestion Control Mechanism

As a consequence of using a reactive congestion control mechanism, Reno *needs* to create losses to find that it is exceeding the bandwidth of the connection. This can be seen in Figure 4.1, which shows the trace of a Reno connection sending 1MB of data over the network configuration seen in Figure 4.2, with no other traffic sources; i.e., only Host1a sending to Host1b. In this case, the router queue size is 10, each

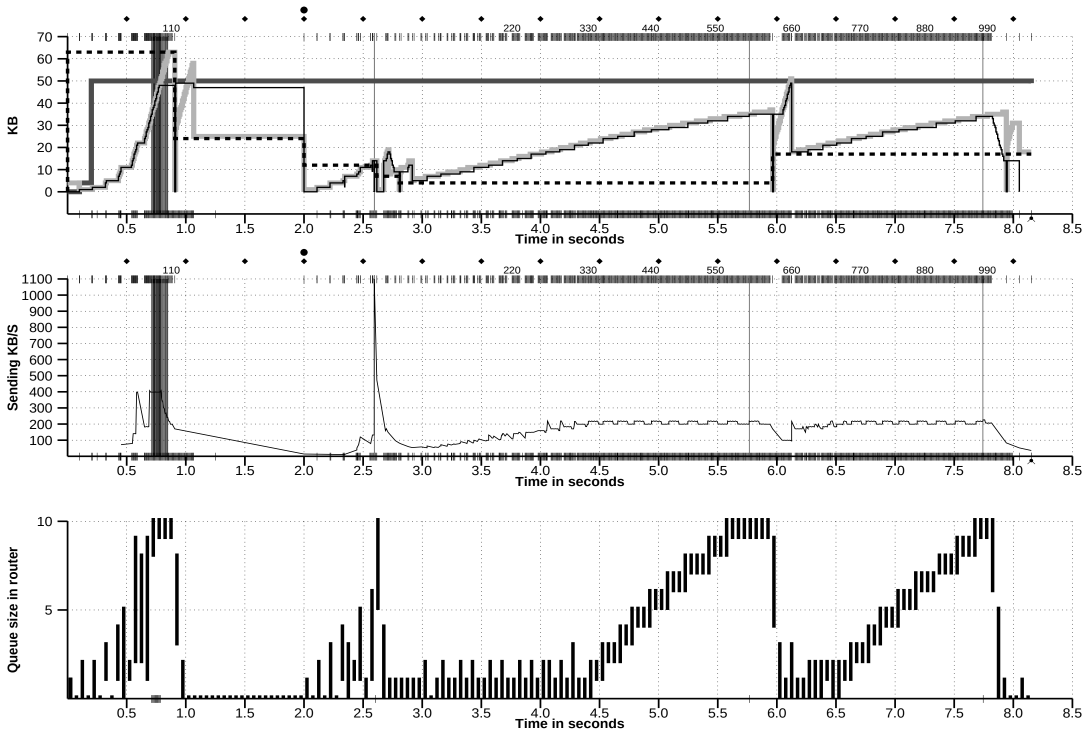


Figure 4.1: TCP Reno with no other traffic (throughput: 123 KB/s)

packet is 1.4KB, and the queuing discipline is FIFO.

As seen in Figure 4.1, Reno's mechanism to detect the available bandwidth is to continually increase its window size, using up buffers along the connection's path, until it congests the network and segments are lost. It then detects these losses and decreases its window size. Consequently, Reno is continually congesting the network and creating its own losses. These losses may not be expensive if the Fast Retransmit and Fast Recovery mechanisms catch them, as seen with the losses at 5.8 and 7.7 seconds, but by unnecessarily using up buffers at the bottleneck router it is creating losses for other connections sharing this router.

It is possible to set the experiment in such a way that there is little or no loss by limiting the maximum window size so that it never exceeds the delay-bandwidth product of the connection plus the maximum queue sizes at the routers. However, this it is not a realistic experiment since this information on the state of the routers is normally unknown to users of real networks.

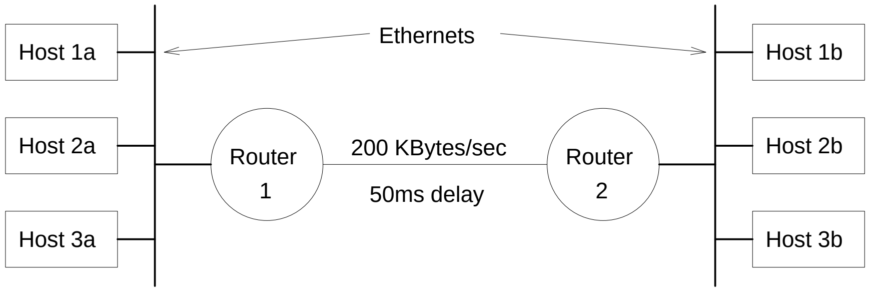


Figure 4.2: Network configuration for simulations.

4.2 Previous Congestion Avoidance Mechanisms

There are several previously proposed approaches for proactive congestion detection based on a common understanding of how the network changes as it approaches congestion (an excellent development is given in [16]). These changes can be seen in Figure 4.1 in the time interval from 4.5 to 7.5 seconds. One change is the increased queue size in the intermediate nodes of the connection, resulting in an increase of the RTT for each successive segment. Wang and Crowcroft's DUAL algorithm [28] is based on reacting to this increase of the round-trip delay. The congestion window normally increases as in Reno, but every two round-trip times the algorithm checks to see if the current RTT is greater than the average of the minimum and maximum RTTs seen so far. If it is, then the algorithm decreases the congestion window by one-eighth.

Jain's CARD (Congestion Avoidance using Round-trip Delay) approach [16] is based on an analytic derivation of a socially optimum window size for a deterministic network. The decision as to whether or not to change the current window size is based on changes to both the RTT and the window size. The window is adjusted once every two round-trip times based on the sign of the product ($WindowSize_{current}$ - $WindowSize_{old}$) $\times$ ($RTT_{current}$ - RTT_{old}) as follows: if the result is positive, decrease the window size by one-eighth; if the result is negative or zero, increase the window size by one maximum segment size. Note that the window changes during every adjustment, that is, it oscillates around its optimal point.

Another change seen as the network approaches congestion is the flattening of the sending rate. Wang and Crowcroft's Tri-S scheme [27] takes advantage of this fact. Every RTT, they increase the window size by one segment and compare the throughput achieved to the throughput when the window was one segment smaller. If the difference is less than one-half the throughput achieved when only one segment was in transit—as was the case at the beginning of the connection—they decrease the window by one segment. Tri-S calculates the throughput by dividing the number of bytes outstanding in the network (UNACK_COUNT) by the RTT.

A final congestion avoidance mechanism is Keshav's Packet Pair mechanism [20]. As the name implies, this mechanism sends a pair of packets, one immediately after the other, and uses the spacing of the acknowledgments to determine the available bandwidth; see Figure 3.6.

As described in the original papers, these mechanisms have weaknesses which make them unsuitable for current networks. As a matter of fact, none of these mechanisms was ever tested on a real network. DUAL uses individual measurements of the RTT to determine if the congestion window should increase or decrease. This mechanism works well in the simple queueing networks in which it was tested, but breaks down in real networks where the RTT is highly variable.

CARD continually changes the window size and measures the effect this has on the RTT. Based on this effect the window will be either increased or decreased. For example, a window size increase ($WindowSize_{current}$ - $WindowSize_{old} > 0$) which does not increase the RTT results in a further increase of the window. CARD suffers from the same weakness described in the previous paragraph—it uses individual measurements of the RTT.

As long as no losses are detected, Tri-S can either increase its window size, or leave it unchanged by decreasing it right after it was increased. As a result, Tri-S cannot adapt to decreases in the available bandwidth unless the decrease is so large that it results in packet losses. That is, Tri-S cannot avoid congestion created by increasing background traffic levels.

The spacing of acknowledgments in Packet Pair can be easily lost when there is traffic in the network. For example, the first acknowledgment may be delayed in a router by encountering a large queue at that router. If the second acknowledgment is not delayed by a similar amount, the spacing between them cannot be used to determine the available bandwidth. As a result, Packet Pair can only be used in networks where all routers use queueing disciplines that preserve the spacing—such as Fair Queueing [8].

4.3 Congestion Avoidance in Vegas

Vegas's approach is most similar to Tri-S in that it looks at changes in the throughput rate, or more specifically, changes in the sending rate. However, it differs from Tri-S in the way it calculates throughput and instead of looking for a change in the throughput slope, it compares the measured throughput rate with an expected throughput rate. The basis for this idea can be seen in Figure 4.1 in the region between 3.0 and 5.5 seconds. As the window size increases one expects the throughput (or sending rate) to also increase. But the throughput cannot increase beyond the available bandwidth; beyond this point, any increase in the window size only results in the segments taking up buffer space at the bottleneck router.

Vegas uses this idea to measure and control the amount of *extra* data this connection has in transit; that is, data that would not have been sent if the bandwidth used by the connection exactly matched the available bandwidth of the network. The goal of Vegas is to maintain the "right" amount of extra data in the network. Obviously, if a connection is sending too much extra data, it will end up distributed over system buffers and cause congestion. Less obviously, if a connection is sending too little extra data, it cannot respond rapidly enough to transient increases in the available network bandwidth. Vegas's congestion avoidance actions are based on changes in the estimated amount of extra data in the network, and not only on dropped segments.

We now describe the algorithm in detail. Note that the algorithm is not in effect during slow-start. Vegas's behavior during slow-start is described in Section 4.4.

First, define a given connection's *BaseRTT* to be the RTT of a segment when the connection is not congested. In practice, Vegas sets *BaseRTT* to the minimum of all measured round trip times; it is commonly the RTT of the first segment sent by the connection, before the router queues increase due to traffic generated by this connection. Although we do not know the exact value for the BaseRTT, our experience suggests our algorithm is not sensitive to errors in the BaseRTT. Assuming the connection is not overflowing the link, the expected throughput is given by:

$$\textit{Expected} = \textit{WindowSize} \;/\; \textit{BaseRTT}$$

where *WindowSize* is the size of the current congestion window, assumed to be equal to the number of bytes in transit.

Second, Vegas calculates the current *Actual* sending rate. This is done by recording the sending time for a distinguished segment, recording how many bytes are transmitted between the time that segment is sent and its acknowledgment is received, computing the RTT for the distinguished segment when its acknowledgment arrives, and dividing the number of bytes transmitted by this RTT. This calculation is done once per round-trip time.

Third, Vegas compares *Actual* to *Expected*, and adjusts the window accordingly. Let *Diff* = *Expected* - *Actual.* Note that *Diff* is positive or zero by definition, since *Actual* > *Expected* implies the need to change *BaseRTT* to the latest sampled RTT. Also define two thresholds, $\alpha < \beta$ roughly corresponding to having too little and too much extra data in the network, respectively. When *Diff* $\leq \alpha$, Vegas increases the congestion window linearly during the next RTT, and when *Diff* $\geq \beta$, Vegas decreases the congestion window linearly during the next RTT. Vegas leaves the congestion window unchanged when $\alpha <$ *Diff* $< \beta$.

Intuitively, the farther away the actual throughput gets from the expected throughput, the more congestion there is in the network, which implies that the sending rate should be reduced. The β threshold triggers this decrease. On the other hand, if the bottleneck router queue is empty (no congestion), any transient—lasting less than one RTT—increase in the available bandwidth will be wasted since

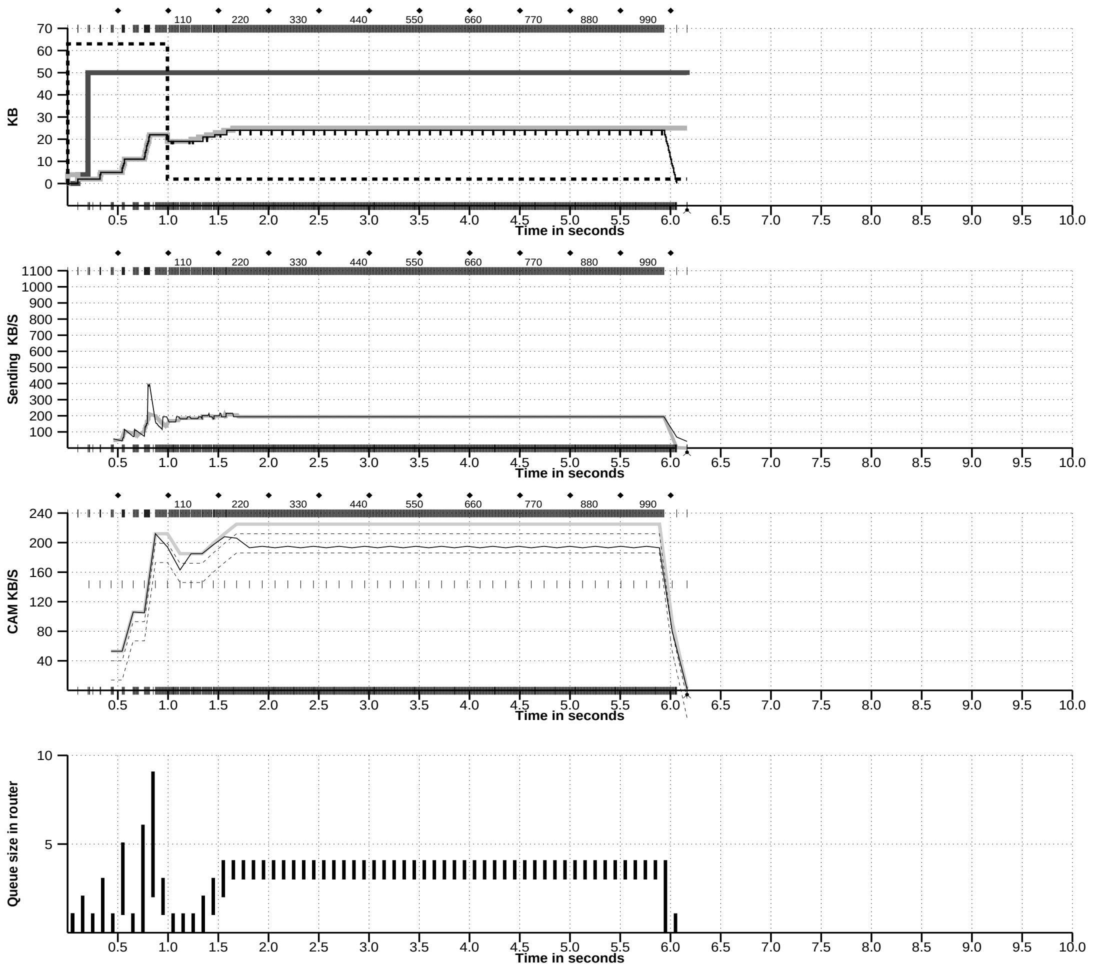


Figure 4.3: TCP Vegas with no other traffic (throughput: 169 KB/s).

the sources only increase their window size every RTT. That is, it is to the connection's advantage to use a few of the buffers at the bottleneck router. The α threshold triggers this increase. The overall goal is to keep between α and β extra bytes in the network.

We conducted a series of experiments on a pair of SparcStations connected by an Ethernet to determine the processing overhead due to the code in Vegas's congestion avoidance mechanism. These experiments show that Vegas uses about 5% more CPU cycles than Reno.

Because the algorithm, as just presented, compares the difference between the actual and expected throughput rates to the α and β thresholds, these two thresholds are defined in terms of KB/s. However, it is perhaps more accurate to think in terms of how many extra *buffers* the connection is occupying in the network. For example, on a connection with a *BaseRTT* of 100ms and a segment size of 1KB, if $\alpha = 30$KB/s and $\beta = 60$KB/s, one can think of α as saying that the connection needs to be occupying at least three extra buffers in the network, and β saying it should occupy no more than six extra buffers in the network.

In practice, we express α and β in terms of buffers rather than extra bytes in transit. During linear increase/decrease mode—as opposed to the slow-start mode described below—α is set to one and β to three. This can be interpreted as an attempt to use at least one, but no more than three extra buffers in the connection. These values for α and β were chosen because they are the smallest feasible values. The value of α needs to be greater than zero so the connection uses at least one buffer at the bottleneck router. Then, when the aggregate traffic from the other connections decreases (as is bound to happen every so often), the connection can take advantage of the extra available bandwidth immediately without having to wait for the one RTT delay necessary for the linear increase to occur. β should be larger than α so small sporadic changes in the available bandwidth will not create oscillations in the window size. In other words, the use of the $\alpha - \beta$ region provides a damping effect. It was decided to set β to 3 based on experiments in the simulator and on real networks.

Note that even though the goal of this mechanism is to avoid congestion by limiting the number of buffers used at the bottleneck, it may not be able to achieve this when there are a large number of "bulk data" connections going through a bottleneck with a small number of buffers. In such a case the mechanisms in TCP Vegas are unable to avoid congestion.

Figure 4.3 shows the behavior of TCP Vegas when there is no other traffic present; this is the same condition that Reno ran under in Figure 4.1. Unlike Reno, Vegas has no losses when there is no other traffic. Vegas's congestion window stops growing once it is using all of the available bandwidth.

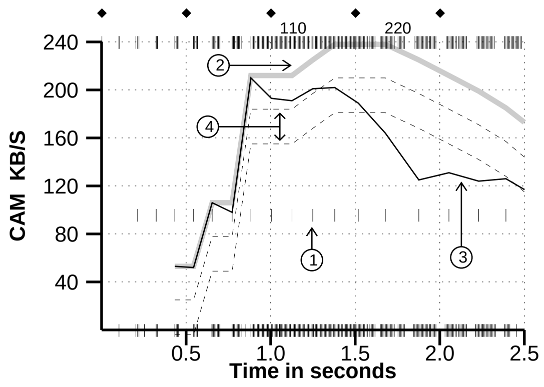


Figure 4.4: Congestion detection and avoidance in Vegas.

There is one new type of graph in this figure, the third one, which depicts the congestion avoidance mechanism (CAM) used by Vegas. As before, we give a detailed graph (Figure 4.4) keyed to the following explanation:

1. The small vertical line—once per RTT—shows the times when Vegas makes a congestion control decision; i.e., computes *Actual* and adjusts the window accordingly.
2. The gray line shows the *Expected* throughput. This is the throughput one should get if all the bytes in transit are able to get through the connection in one *BaseRTT*.
3. The solid line shows the *Actual* sending rate. It is calculated from the number of bytes sent in the last RTT.
4. The dashed lines are the thresholds used to control the size of the congestion window. The top line corresponds to the α threshold and the bottom line corresponds to the β threshold.

Figure 4.5 shows a trace of a Vegas connection transferring one MByte of data, while sharing the bottleneck router with *tcplib* traffic (i.e., with the TRAFFIC protocol configured in the simulator). The third graph shows the output produced by the TRAFFIC protocol—the thin line is the sending rate in KB/s as seen in 100ms intervals and the thick line is a running average (size 3). The bottom graph shows

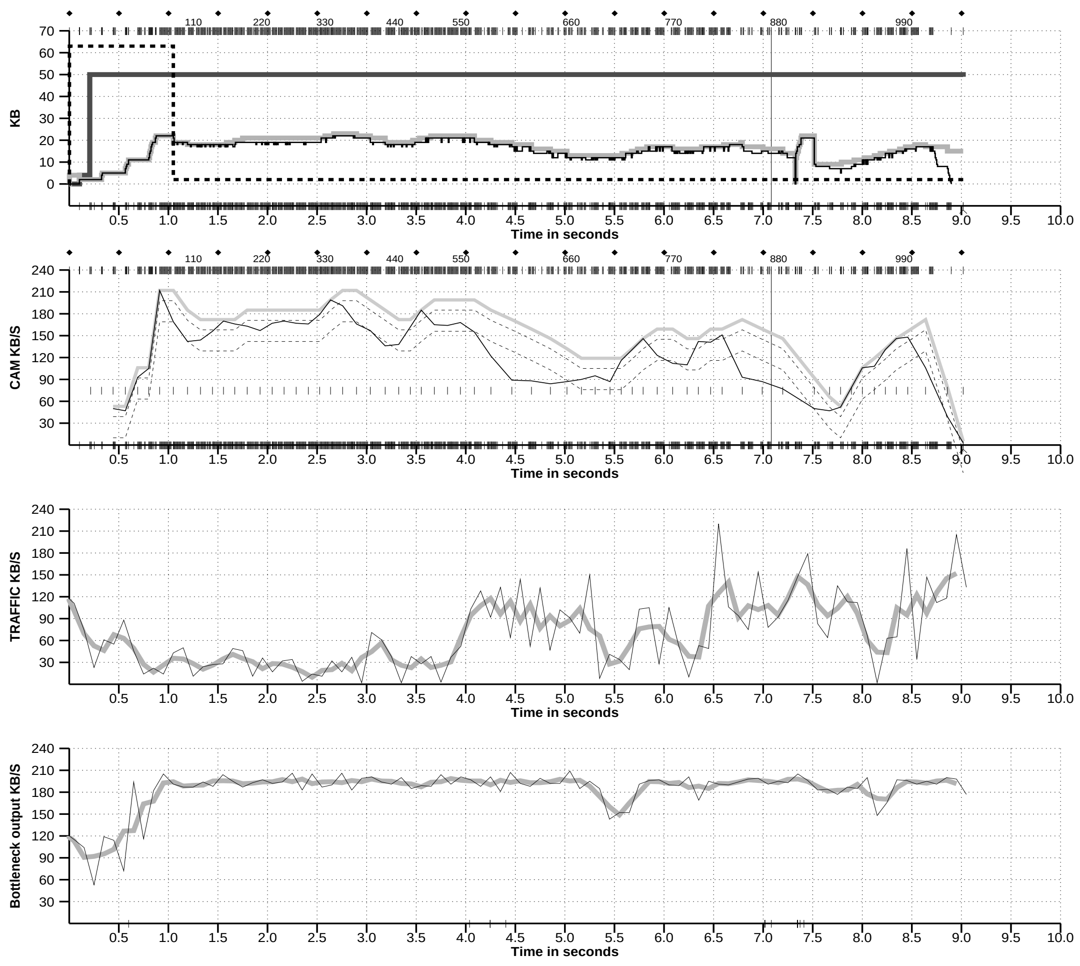


Figure 4.5: TCP Vegas with *tcplib*-generated background traffic.

the output of the bottleneck link which has a maximum bandwidth of 200KB/s. The figure clearly shows Vegas's congestion avoidance mechanisms at work and how its throughput adapts to the changing conditions on the network. For example, as the background traffic increases at 3.7 seconds (third graph), the Vegas connection detects it and decreases its window size (top graph) which results in a reduction in its sending rate (second graph). When the background traffic slows down at 5, 6 and 7.5 seconds, the Vegas connection increases its window size, and correspondingly its sending rate. The bottom graph shows that most of the time there is a 100% utilization of the bottleneck link.

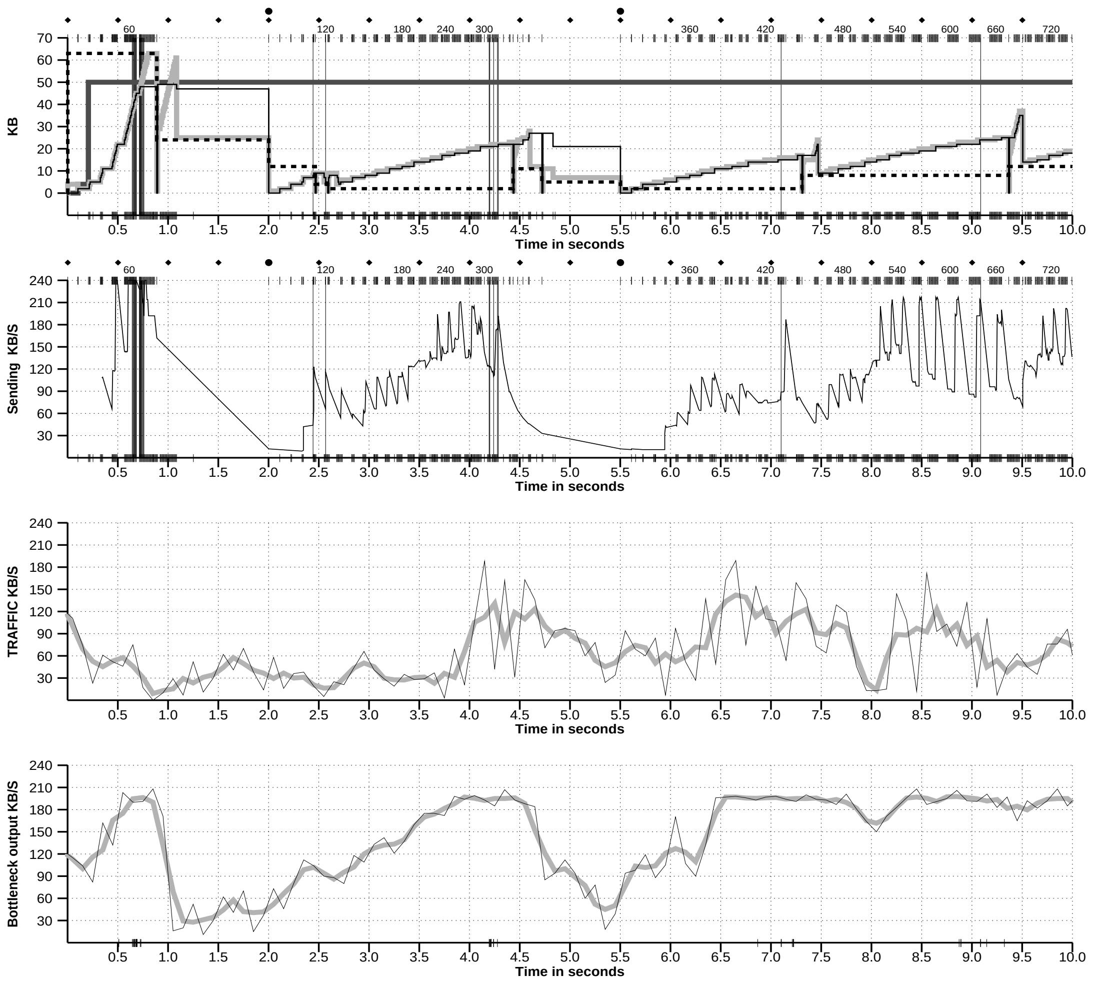


Figure 4.6: TCP Reno with *tcplib*-generated background traffic.

In contrast, Figure 4.6 shows the behavior of Reno under similar conditions. This figure shows that, as long as there are no losses, Reno continuously increases its window size regardless of the background traffic level. For example, as the background traffic increases at 3.7 seconds, the Reno connection keeps increasing its window size until there is congestion. This results in losses, both to itself and to connections which are part of the background traffic. The graph only shows the first 10 seconds of the one MByte transfer; it took 14.2 seconds to complete the transfer. The bottom graph shows that there is under-utilization of the bottleneck link.

The important thing to take away from these figures is that Vegas is able to make better utilization of the bottleneck link than Reno. In other words, and this is discussed further in Section 6.4, Vegas's increased throughput is not a result of its taking bandwidth away from Reno connections, but due to a more efficient utilization of the bottleneck link. In fact, Reno connections do slightly better when the background traffic is running on top of Vegas as compared to when the traffic is running on top of Reno[1].

4.4 Congestion Avoidance During Slow-Start

As mentioned in the previous chapter, the slow-start mechanism in TCP will continue until the window reaches a preset maximum or when losses occur. Since there is usually not enough information to determine the right preset maximum during the initial slow-start, slow-start continues the exponential growth of the congestion window until it results in large packet losses—see Figure 4.1. Later occurrences of slow-start have enough information to set the maximum: they set it to half the size the congestion window had when losses occurred.

What is needed is a way to find a connection's available bandwidth which does not incur these losses. Towards this end, the congestion avoidance mechanism was incorporated into slow-start with only minor modifications. To be able to detect and avoid congestion during slow-start, Vegas allows exponential growth only every other RTT. In between, the congestion window stays fixed so that a valid comparison of the expected and actual rates can be made. When the actual rate falls below the expected rate by the equivalent of one router buffer, Vegas changes from slow-start mode to the linear increase/decrease mode—described in the previous section—to prevent packet losses.

The behavior of the Vegas's slow-start mechanism can be seen in Figure 4.3 and Figure 4.5. Vegas's slow-start mechanism is highly successful at preventing the losses incurred during Reno's initial slow-start period (see Chapter 5).

Two problems remain during any slow-start period. First, segments are sent at

[1]Remember that the background traffic is generated by a protocol (TRAFFIC) that runs on top of TCP.

a rate higher than the available bandwidth—up to twice the available bandwidth, depending on the ACKing frequency (e.g., every segment or every two segments). This causes the bottleneck router to buffer up to half of the data sent on each RTT, thereby increasing the likelihood of losses during the slow-start period. Moreover, as network speeds increase, so does the amount of buffering needed. Second, while Vegas's congestion avoidance mechanism during the initial slow-start period is quite effective, it can still overshoot the available bandwidth, and depends on enough buffering at the bottleneck router to prevent losses until realizing it needs to slow down. Specifically, if the connection can handle a particular window size, then Vegas will double that window size—and as a consequence, double the sending rate—on the next RTT. At some point the available bandwidth will be exceeded.

We have experimented with a solution to both problems. To simplify the following discussion, we refer to the alternative version of Vegas with an experimental slow-start mechanism as Vegas*. Vegas* is based on using the spacing of the acknowledgments to gauge the available bandwidth. The idea is similar to Keshav's Packet-Pair probing mechanism [20], except that it uses the average spacing between four segments sent during the slow-start period rather than the spacing from two segments. Looking at four segments results in a more robust algorithm than looking at two because we can use the average of three values rather than just one value. This available bandwidth estimate is used to set the threshold window with an appropriate value, which makes Vegas* less likely to overshoot the available bandwidth.

Specifically, as each ACK is received, Vegas* schedules an event at a certain point in the future, based on its available bandwidth estimate, to increase the congestion window by one maximum segment size. This is in contrast to increasing the window immediately upon receiving the ACK. For example, assume the RTT is 100ms, the maximum segment size is 1 KByte, and the available bandwidth estimate is currently 200 KB/s. During the slow-start period, time is divided into intervals of length equal to one RTT. If 4 ACKs are expected to arrive during the current RTT interval, Vegas* uses the bandwidth estimate (200KB/s) to guess the spacing

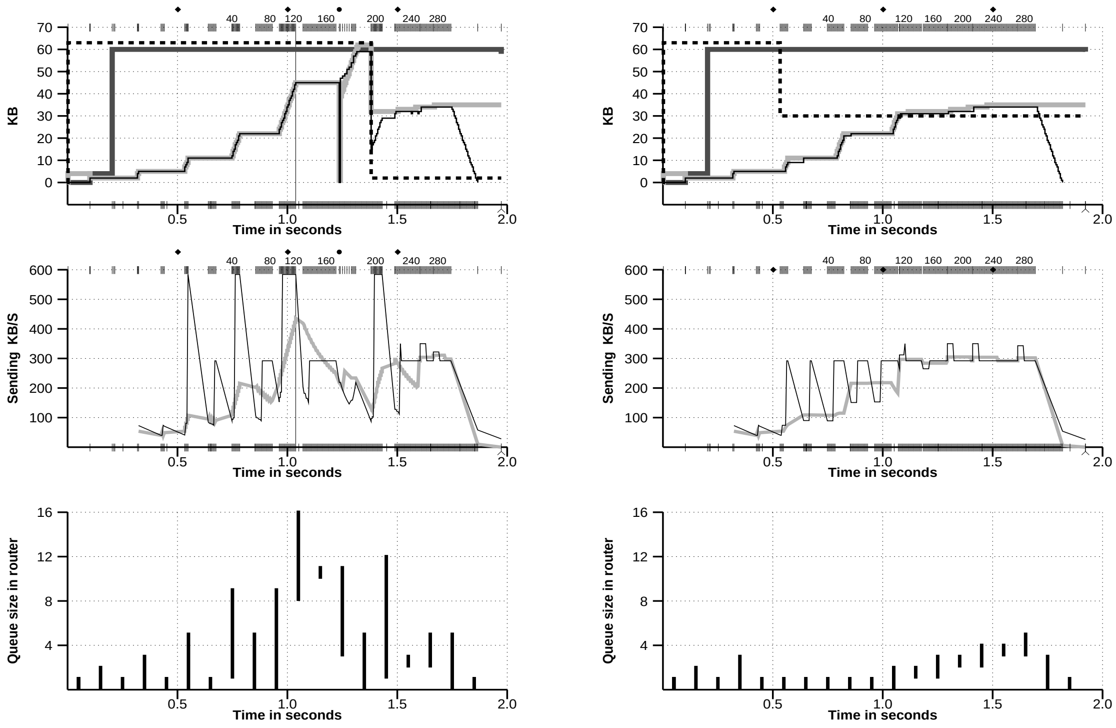


Figure 4.7: TCP Vegas on the left, experimental on the right.

between the incoming ACKs (1KB / 200KB/s = 5ms) and as each ACK is received, it schedules an event to increase the congestion window (and to send a segment) at 20ms (5×4) in the future.

The graphs in Figure 4.7 show the behavior of Vegas (left) and Vegas* (right) during the initial slow-start. For this set of experiments, the available bandwidth was 300KB/s and there were 16 buffers at the router. Looking at the graphs on the left, one sees that a packet is lost at around 1 second (indicated by the thin vertical bar) as a result of sending at 400KB/s. This is because Vegas detected no problems at 200KB/s, so it doubled its sending rate, but in this particular case, there were not enough buffers to protect it from the losses. The bottom graph demonstrates the need to buffer half of the data sent on each RTT as a result of sending at a rate twice the available bandwidth.

The graphs on the right illustrate the behavior of Vegas*. It sets the threshold window (dashed line) based on the available bandwidth estimate. This results in the congestion window halting its exponential growth at the right time—when the sending rate equals the available bandwidth and preventing the losses. The middle graph shows that the sending rate never exceeds the available bandwidth (300KB/s) by much. Finally, the bottom graph shows that Vegas* does not need as many buffers as Vegas.

Notice that while the available bandwidth estimate could be used to jump immediately to the available bandwidth by using rate control during one RTT interval, congestion can result if more than one connection does this at the same time. Even though it is possible to congest the network if more than one connection does slow-start at the same time, there is an upper bound on the number of bytes sent during the RTT when congestion occurs regardless of the number of connections simultaneously doing slow-start—about twice the number of bytes that can be handled by the connection. There is no such limit if more than one connection tries to use the available bandwidth at once. Hence, we strongly recommend against doing this unless it is known *a priori* that there are no other connections sharing the path, or if there are, that they will not increase their sending rate at the same time.

Although these traces illustrate how Vegas*'s experimental slow-start mechanism does in fact address the two problems with Vegas outlined above, simulation data indicates that the new mechanism does not have a measurable impact on throughput, and only marginally improves the loss rate. While additional simulations might expose situations where Vegas* is more beneficial, we have decided to not include these modifications in Vegas. The results presented in Chapter 5 are for Vegas, not Vegas*.

4.5 Other Mechanisms in TCP Vegas

As mentioned earlier, Reno uses two mechanisms to detect and then retransmit lost segments. The first mechanism is the retransmit timeout. It determines a timeout interval based on round trip time (RTT) and variance estimates computed by sampling the time between when a segment is sent and an ACK arrives. If a segment is not acknowledged within the timeout interval, it is presumed lost and is then retransmitted.

The second mechanism is the Fast Retransmit Mechanism. It was incorporated into Reno (and newer) implementations of TCP to help detect losses earlier than the retransmit timeout mechanism. As part of Fast Retransmit Mechanism, Reno sends a duplicate ACK whenever it receives a new segment that it cannot acknowledge because it has not yet received all the previous segments. A source then interprets 3 consecutive duplicate ACKs as an indication that a segment was lost and retransmits the appropriate segment.

The Fast Retransmit Mechanism is very successful—it prevents more than half of the coarse-grained timeouts that occur on TCP implementations without it. However, some of our preliminary analysis indicated that eliminating the dependency on coarse-grained timeouts—i.e. losses are detected "quickly"—would result in at least a 19% increase in throughput.

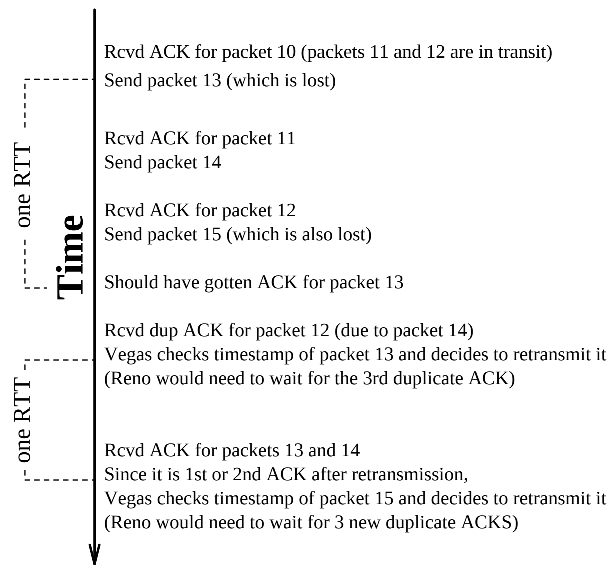

Time
one RTT
one RTT
Rcvd ACK for packet 10 (packets 11 and 12 are in transit)
Send packet 13 (which is lost)
Rcvd ACK for packet 11
Send packet 14
Rcvd ACK for packet 12
Send packet 15 (which is also lost)
Should have gotten ACK for packet 13
Rcvd dup ACK for packet 12 (due to packet 14)
Vegas checks timestamp of packet 13 and decides to retransmit it
(Reno would need to wait for the 3rd duplicate ACK)
Rcvd ACK for packets 13 and 14
Since it is 1st or 2nd ACK after retransmission,
Vegas checks timestamp of packet 15 and decides to retransmit it
(Reno would need to wait for 3 new duplicate ACKS)

Figure 4.8: Example of retransmit mechanism.

Vegas, therefore, extends Reno's retransmission mechanisms as follows. First, Vegas reads and records the system clock each time a segment is sent. When an ACK arrives, Vegas reads the clock again and does the RTT calculation using this time and the timestamp recorded for the relevant segment. Vegas then uses this more accurate RTT estimate to decide to retransmit in the following two situations (a simple example is given in Figure 4.8).

- When a duplicate ACK is received, Vegas checks to see if the difference between the current time and the timestamp recorded for the relevant segment is greater than the timeout value. If it is, then Vegas retransmits the segment without having to wait for n (3) duplicate ACKs. In many cases, losses are either so great or the window so small that the source will never receive three duplicate ACKs, and therefore, Reno has to rely on the coarse-grained timeout mentioned above.

- When a non-duplicate ACK is received, if it is the first or second one after a retransmission, Vegas again checks to see if the time interval since the segment was sent is larger than the timeout value. If it is, then Vegas retransmits the segment. This will catch any other segment that may have been lost prior to the retransmission without having to wait for a duplicate ACK.

In other words, Vegas treats the receipt of certain ACKs as a hint to check if a timeout should occur. Since it only checks for timeouts in rare occasions, the overhead is small. Notice that even though one could reduce the number of duplicate ACKs used to trigger the Fast Retransmit from 3 duplicate ACKs to either 2 or 1, it is not recommended as it could result in many unnecessary retransmissions and because it makes assumptions about the likelihood that packets will be delivered out of order.

The goal of the new retransmission mechanism is not just to reduce the time to detect lost packets from the third duplicate ACK to the first or second duplicate ACK—a small savings—but to detect lost packets in cases where there is no second or third duplicate ACK. The new mechanism is very successful at achieving this goal, as it further reduces the number of coarse-grained timeouts in Reno by more than half. Note that Vegas contains Reno's coarse-grained timeout code in case the new mechanisms fail to recognize a lost segment.

Related to making timeouts more timely, notice that the congestion window should only be reduced due to losses that happened at the current sending rate, and not due to losses that happened at an earlier, higher rate. In Reno, it is possible to decrease the congestion window more than once for losses that occurred during one RTT interval[9]. In contrast, Vegas decreases the congestion window only if the retransmitted segment was previously sent *after* the last decrease. Any losses that happened before the last window decrease do not imply that the network is congested for the *current* congestion window size, and therefore, do not imply that it should be decreased again. This change is needed because Vegas detects losses much sooner than Reno.

CHAPTER 5
Experimental Results

This chapter reports and analyzes the results from both the Internet and the simulator experiments. The results from the Internet experiments are evidence that Vegas's enhancements to Reno produce significant improvements on both the throughput (37% higher) and the number of losses (less than half) under real conditions. The simulator experiments allow us to also study related issues such as how Vegas connections affect Reno connections and what happens when all connections are running over Vegas. Note that because it is simple to move a protocol between the simulator and the "real world", the all numbers reported in this section are for exactly the same code.

5.1 Internet Results

This section begins with measurements of TCP transfers over the Internet. Specifically, we measured TCP transfers between the University of Arizona (UA) and the National Institutes of Health (NIH). The connection goes through 17 routers, and passes through Denver, St. Louis, Chicago, Cleveland, New York and Washington DC. The results are derived from a set of runs over a seven day period from January 23-29, 1994. Each run consists of seven transfers from UA to NIH—Reno sends 1MB, 512KB, and 128KB, a version of Vegas with $\alpha = 1$ and $\beta = 3$ (denoted Vegas-1,3) sends 1MB, 512KB, and 128KB, and second version of Vegas with $\alpha = 2$ and $\beta = 4$ (denoted Vegas-2,4) sends 1MB. A run was started approximately once every hour, a 45 second delay was inserted between each transfer in a run to give the network a chance to settle down, and the order of the transfers within each run was chosen at random. In total, there were 85 runs of each set of transfers.

Table 5.1 shows the averaged results for the 1MB transfers. We don't show any

	Reno	Vegas-1,3	Vegas-2,4
Throughput (KB/s)	53.00	72.50	75.30
Throughput Ratio	1.00	1.37	1.42
Retransmissions (KB)	47.80	24.50	29.30
Retransmit Ratio	1.00	0.51	0.61
Coarse Timeouts	3.30	0.80	0.90

Table 5.1: 1MByte transfer over the Internet.

variance numbers because the network conditions were different for each set of transfers. Depending on the congestion avoidance thresholds, it shows between 37% and 42% improvement over Reno's throughput with only 51% to 61% of the retransmissions. When comparing Vegas and Reno within each run, Vegas outperforms Reno 89% of the time and across all levels of congestion; i.e., during both the middle of the night and during periods of high load. The average throughput difference in the 11% of runs where Reno outperforms Vegas is 10% (52.1 KB/s for Reno versus 46.7 KB/s for Vegas) and the maximum difference is 25% (49.0 KB/s for Reno versus 36.5 KB/s for Vegas).

Also, the throughput was a little higher with the bigger thresholds (2,4), since the Vegas connection used more buffers at the bottleneck router which could be used to fill bandwidth gaps occurring when the background traffic slowed down. However, the higher buffer utilization at the bottleneck also resulted in higher losses and slightly higher delays. We prefer the more conservative approach of using fewer resources, so have settled on avoidance thresholds of $\alpha = 1$ and $\beta = 3$. For the following experiments $\alpha = 1$ and $\beta = 3$ unless specified otherwise.

Experiments were also done with smaller transfer sizes to check if Vegas's throughput improvements were a result of the 1MB transfer sizes. Table 5.2 shows the effect of transfer size on both throughput and retransmissions for Reno and Vegas-1,3. First, observe that Vegas does better relative to Reno as the transfer size decreases. In terms of throughput, there is an increase from 37% to 71%. The results are similar for retransmissions, as the relative number of Vegas retransmissions goes from

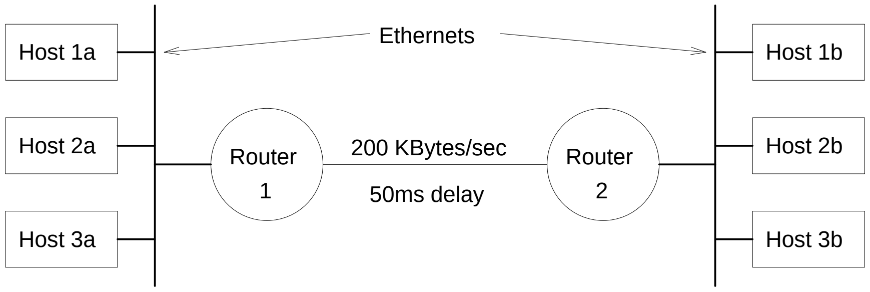


Figure 5.1: Network configuration for simulations.

51% of Reno's to 17% of Reno's.

Notice that the number of kilobytes retransmitted by Reno starts to flatten out as the transfer size decreases. When the transfer size is decreased by half, from 1MB to 512KB, there is a 42% decrease in the number of kilobytes retransmitted. When the transfer size is further decreased to one-fourth its previous value, from 512KB to 128KB, the number of kilobytes retransmitted only decreases by 18%. This indicates that Reno is approaching the average number of kilobytes retransmitted due to its slow-start losses. From these results, it can be concluded that, for the conditions of this experiments, there are around 20KBs retransmitted during slow-start.

On the other hand, the number of kilobytes retransmitted by Vegas decreases almost linearly with respect to the transfer size. This indicates that Vegas eliminates nearly all losses during slow-start due to its modified slow-start with congestion avoidance. Note that if the transfer size is smaller than about twice the bandwidth-

	1024KB		512KB		128KB	
	Reno	Vegas	Reno	Vegas	Reno	Vegas
Throughput (KB/s)	53.00	72.50	52.00	72.00	31.10	53.10
Throughput Ratio	1.00	1.37	1.00	1.38	1.00	1.71
Retransmissions (KB)	47.80	24.50	27.90	10.50	22.90	4.00
Retransmit Ratio	1.00	0.51	1.00	0.38	1.00	0.17
Coarse Timeouts	3.30	0.80	1.70	0.20	1.10	0.20

Table 5.2: Effects of transfer size over the Internet.

	Reno/Reno	Reno/Vegas	Vegas/Reno	Vegas/Vegas
Throughput (KB/s)	60/109	61/123	66/119	74/131
Throughput Ratios	1.00/1.00	1.02/1.13	1.10/1.09	1.23/1.20
Retransmissions (KB)	30/22	43/1.8	1.5/18	0.3/0.1
Retransmit Ratios	1.00/1.00	1.43/0.08	0.05/0.82	0.01/0.01

Table 5.3: One-on-one (300KB and 1MB) transfers.

delay product, then there will be no losses for either Vegas nor Reno, assuming the bottleneck router has enough buffers to absorb temporary sending rates above the connections available bandwidth.

5.2 Simulation Results

This subsection reports the results of series of experiments using *x*-Sim. The simulator allows us to better control the experiment, and in particular, gives us the opportunity to see whether or not Vegas gets its performance at the expense of Reno-based connections. Note that all the experiments used in this subsection are on the network configuration shown in Figure 5.1. Experiments with different bandwidth-delay parameters have shown similar results.

5.2.1 One-on-One Experiments

To understand how two TCP connections interfere with each other, a 1MB transfer is started from one host, and then after a variable delay, it is followed by a 300KB transfer from another host. The transfer sizes and delays are chosen to ensure that the smaller transfer runs concurrently with the larger and terminates before it.

Table 5.3 gives the results for the four possible combinations, where the column heading Reno/Vegas denotes a 300KB transfer using Reno concurrent with a 1MByte transfer using Vegas. For each combination, the table gives the measured throughput and number of kilobytes retransmitted for both transfers; e.g., in the case of Reno/Vegas, the 300KB Reno transfer achieved a 61KB/s throughput rate and the 1MByte Vegas transfer achieved a 123KB/s throughput rate. Comparing the small transfer to the large transfer in any given column is not meaningful because the large transfer was able to run by itself during most of the test. Small transfers should only be compared to other small transfers (similarly with large transfers). The ratios for both throughput rate and kilobytes retransmitted are relative to the Reno/Reno column. The values in the table are averages from 12 runs, using 15 and 20 buffers in the routers, and with the delay before starting the smaller transfer ranging between 0 and 2.5 seconds.

The main thing to take away from these numbers is that Vegas does not adversely affect Reno's throughput. Reno's throughput changes very little when it is competing with Vegas rather than itself—the ratios for Reno are 1.02 and 1.09 for Reno/Vegas and Vegas/Reno, respectively. Also, when Reno competes with Vegas rather than itself, the combined number of kilobytes retransmitted for the pair of competing connections drops significantly. The combined Reno/Reno retransmits are 52KB compared with 45KB for Reno/Vegas and 19KB for Vegas/Reno. Finally, note that the combined Vegas/Vegas retransmits are less than 1KB on the average—an indication that the congestion avoidance mechanism is working.

The 43% increase in the losses of Reno for the Reno/Vegas case is explained as follows. The Vegas connection starts first, and is using the full bandwidth (200KB/s) by the time the Reno connection starts. When Vegas detects that the network is starting to get congested, it decreases its sending rate to between 80 and 100KB/s. The losses incurred by Reno (about 48KB), are approximately the losses Reno experiences when it is running by itself on a network with 100 to 120KB/s of available bandwidth and around 15 available buffers at the bottleneck router. The reason the losses where smaller for the 300KB transfer in the Reno/Reno experiment is that by the time the 300KB transfer starts, the 1MB connection has stopped transmitting due to the losses in its slow-start, and it will not start sending again until it times out at around 2 seconds. A Reno connection sending 300KB when there is 200KB/s of available bandwidth and 20 buffers at the bottleneck router only losses about 3KB.

This type of behavior is characteristic of Reno: by slightly changing the param-

	Reno	Vegas-1,3	Vegas-2,4
Throughput (KB/s)	58.30	89.40	91.80
Throughput Ratio	1.00	1.53	1.58
Retransmissions (KB)	55.40	27.10	29.40
Retransmit Ratio	1.00	0.49	0.53
Coarse Timeouts	5.60	0.90	0.90

Table 5.4: 1MByte Transfer with *tcplib*-Generated Background Reno Traffic.

eters in the network, one can observe major changes in Reno's behavior. Vegas, on the other hand, does not show as much discontinuity in its behavior.

Since the probability that there are exactly two connections at one time is small in real life, the experiment was modified by adding *tcplib* background traffic. The results were similar except for the Reno/Vegas experiment in which Reno only had a 6% increase in its retransmission, versus the 43% when there was no background traffic.

5.2.2 Background Traffic

We next measured the performance of a distinguished TCP connection when the network is loaded with traffic generated from *tcplib*. That is, the protocol TRAFFIC is running between Host1a and Host1b in Figure 5.1, and a 1MByte transfer is running between Host2a and Host2b. In this set of experiments, the *tcplib* traffic is running over Reno.

Table 5.4 gives the results for Reno and two versions of Vegas—Vegas-1,3 and Vegas-2,4. Two sets of thresholds were used to measure the sensitivity of the congestion avoidance algorithm to changes in the threshold. The numbers shown are averages from 57 runs, obtained by using different seeds for *tcplib*, and by using 10, 15 and 20 buffers in the routers.

The table shows the throughput rate for each of the distinguished connections using the three protocols, along with their ratio to Reno's throughput. It also gives the number of kilobytes retransmitted, the ratio of retransmits to Reno's, and the average number of coarse-grained timeouts per transfer. For example, Vegas-1,3 had 53% better throughput than Reno, with only 49% of the losses. Again note that there is little difference between Vegas-1,3 and Vegas-2,4.

These simulations tell us the expected improvement of Vegas over Reno: more than 50% improvement on throughput, and only half the losses. The results from the one-on-one experiments indicate that the gains of Vegas are not made at the expense of Reno; this belief is further supported by the fact that the background traffic's throughput is mostly unaffected by the type of connection doing the 1Mbyte transfer.

The tests were rerun using Vegas for the background traffic instead of Reno. These experiments simulate the situation where the whole world uses Vegas. The throughput and the kilobytes retransmitted by the 1MByte transfers (Reno and Vegas) did not change significantly (less than 4%) as compared to the previous experiment.

5.2.3 Other Experiments

Variations of the previous experiments showed similar results, except when TCP's send-buffer size was changed. Below is a summary of these experiments and their results.

There have been reports of change in TCP's behavior when the background traffic is two-way rather than one-way [30], so *tcplib* traffic was added from Host3b to Host3a. The throughput ratio stayed the same, but the loss ratio was much better: 0.29. Reno resent more data and Vegas remained about the same. The fact that there was little change is probably due to the fact that *tcplib* already creates some 2-way traffic—Telnet connections send one byte and get one or more bytes back, and FTP connections send and get control packets before doing a transfer.

For all the experiments reported so far, TCP's send-buffer size was set to 50KB. Experiments were also done with send-buffer sizes between 50KB and 5KB. Vegas's throughput and losses stayed unchanged between 50KB and 20KB; from that point on, as the buffer decreased, so did the throughput. This was due to the protocol not being able to keep the pipe full.

Reno's throughput initially *increased* as the buffers got smaller, and then it decreased. It always remained under the throughput measured for Vegas. We have previously seen this type of behavior while running Reno on the Internet. Figure 4.1 shows that as Reno increases its congestion window, it uses more and more buffers in the router until it loses packets by overruning the queue. Limiting the congestion window by reducing the size of the send-buffer may prevent it from overrunning the router's queue.

5.3 Other Evaluation Criteria

Throughput and losses are not the only metrics by which a transport protocol is evaluated. This section discusses several other issues that must be addressed.

5.3.1 Fairness

If there is more than one connection sharing a bottleneck link, one would like for each connection to receive an equal share of the bandwidth. Unfortunately, given the limited amount of information currently available at the connection endpoints, this is unlikely to happen without some help from the routers. Given that no protocol is likely to be perfectly fair, one needs a way to decide whether its level of fairness is acceptable or not. Given that so far the Internet community has found Reno's level of fairness acceptable, it was decided to compare Vegas's fairness levels to Reno's and judge it in those terms.

Jain's *fairness index* [17] is used as the metric. It is defined as follows: given a set of throughputs $(x_1, x_2, \ldots, x_n)$ the following function assigns a fairness index to the set:

$$f(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2} \qquad \left(= \frac{(E[X])^2}{E[X^2]} \right)$$

For all nonnegative numbers, it follows by Jensen's Inequality $(E[X^2] \geq (E[X])^2)$ that the fairness index always results in numbers between 0 and 1. If all throughputs are the same, the fairness index is 1. If only k of the n users receive equal throughput and the remaining $n - k$ users receive zero throughput, the fairness index is k/n (large dispersion implies small index).

We ran simulations with 2, 4 and 16 connections sharing a bottleneck link, where all the connections either had the same propagation delay, or where one half of the connections had twice the propagation delay of the other half. Many different propagation delays were used, and the results averaged.

In the case of 2 and 4 connections, with each connection transferring 8 MB, Reno was slightly more fair than Vegas when all connections had the same propagation delay (0.993 vs. 0.989), but Vegas was slightly more fair than Reno when the propagation delay was larger for half of the connections (0.962 vs. 0.953). In the experiments with 16 connections, with each connection transferring 2MB, Vegas was more fair than Reno in all experiments regardless of whether the propagation delays were the same or not (0.972 vs. 0.921).

To study the effect that Reno connections have on Vegas connections (and vice versa) we ran 8 connections, each sending 2 MB of data. The experiment consisted of running all the connections on top of Reno, all the connections on top of Vegas, or one half on top on Reno and the other half on top of Vegas. There was little difference between the fairness index of the eight connections running a particular TCP implementation (Vegas or Reno) and the fairness index of the four connections running the same TCP implementation and sharing the bottleneck with the four connections running the other TCP implementation. Similarly, there was little difference in the average size of the bottleneck queue.

In another experiment, four connections ran over background traffic. For this experiment, Vegas was always more fair than Reno. Overall, we conclude that Vegas is no less fair than Reno.

5.3.2 Stability

A second concern is stability—it is undesirable for a protocol to cause the Internet to collapse as the number of connections increases. In other words, as the load increases, each connection must recognize that it should decrease its sending rate.

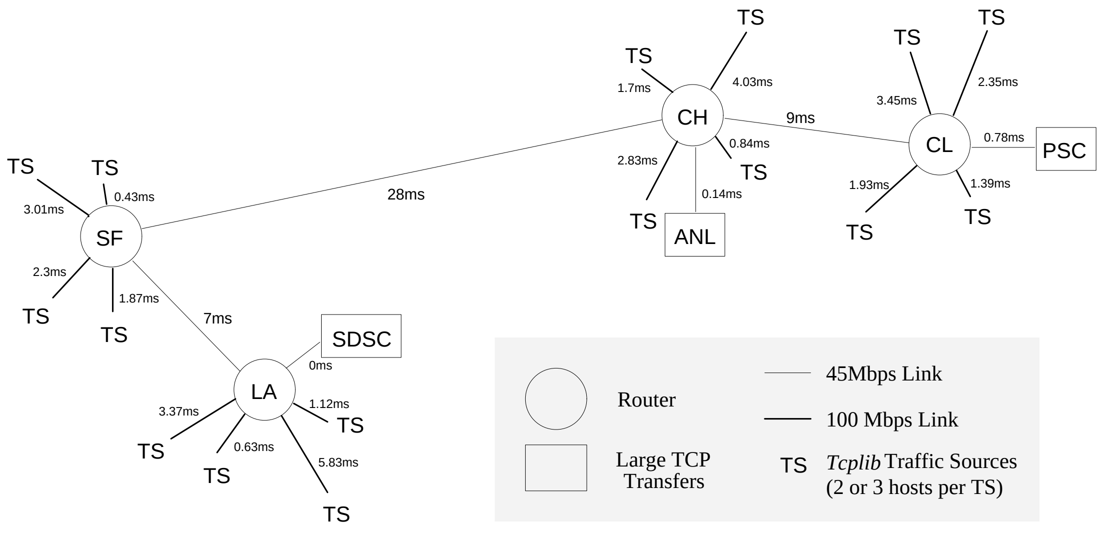


Figure 5.2: Complex simulation network.

Up to the point where the window can be greater than one maximum segment size, Vegas is much better than Reno at recognizing and avoiding congestion—we have already seen that Reno does not avoid congestion, on the contrary, it periodically creates congestion.

Once the load is so high that on average each connection can only send less than one maximum segment's worth of data, Vegas behaves like Reno. This is because this extreme condition implies that coarse-grained timeouts are involved, and Vegas uses exactly the same coarse-grained mechanism as Reno. Experimental results confirm this intuition: running 16 connections, with a 50ms one-way propagation delay, through a router with either 10 or 20 buffers and 100 or 200KB/s of bandwidth produced no stability problems.

We have also simulated complex network topologies like the one shown in Figure 5.2, which consists of 16 traffic sources each of which contains two or three hosts. Each host, in turn, is running *tcplib*-based traffic. The rectangular boxes represent sources of "bulk data" transfers. The resulting traffic consists of nearly a thousand new connections being established per simulated second, where each connection is either a Telnet, FTP, SMTP or NNTP conversation. No stability problems have occurred in any of our simulations when all of the connections are running Vegas.

Intuitively, there is no reason to expect Vegas to lead to network collapse. One reason for this is that most of Vegas's mechanisms are conservative in nature—its congestion window never increases faster than Reno's (one maximum segment per RTT), the purpose of the congestion avoidance mechanism is to *decrease* the congestion window before losses occur, and during slow-start, Vegas stops the exponential growth of its congestion window before Reno would under the same conditions.

5.3.3 Delay

Given that Vegas purposely tries to occupy between one and three extra buffers along the path for each connection, it seems possible that persistent queues could form at the bottleneck router if the whole world ran Vegas. These persistent queues would, in turn, add to the latency of all connections that crossed that router.

Since the analytical tools currently available are not good enough to realistically model and analyze the behavior of either Reno or Vegas, we must rely on simulations to answer this issue. Our simulations show that average queue sizes under Reno and Vegas are approximately the same. However, they also show that Telnet connections in *tcplib* experience between 18 and 40% less delay, on average, when all the connections are Vegas instead of Reno. This seems to suggest that if the whole world ran Vegas, Internet delay would not be adversely affected.

5.4 Discussion

We conclude this chapter with a discussion of issues that are relevant to the process of evaluating the practical value of TCP Vegas. It also comments on the relationship between this work and other efforts to improve end-to-end performance on the Internet.

5.4.1 BSD Variations

TCP has been a rather fluid protocol over the last several years, especially in its congestion control mechanism. Although the general form the original mechanism described in [13] has remained unchanged in all BSD-based implementations (e.g., Tahoe, Reno, BNR2, BSD 4.4), many of the "constants" have changed. For example, some implementations ACK every segment and some ACK every other segment; some increase the window during linear growth by one segment per RTT and some increase by half a segment per RTT plus 1/8th the maximum segment size per ACK received during that RTT; and finally, some use the timestamp option and some do not.

We have experimented with most of these variations and have found the combination used in the *x*-kernel implementation of TCP Reno to be the among the most effective. For example, the latest version of TCP, that found in BSD 4.4-lite, achieves 14% worse throughput than the *x*-kernel's implementation of TCP Reno during Internet type simulations (see Chapter 6).

5.4.2 Alternative Approaches

In addition to improving TCP's congestion control mechanism, there is a large body of research addressing the general question of how to fairly and effectively allocate resources in the Internet.

One example gaining much attention is the question of how to guarantee bandwidth to real-time connections. The basic approach requires that a more intelligent buffer manager be placed in the Internet routers [2]. One might question the relevance of TCP Vegas in light of such mechanisms. We believe end-to-end congestion control will remain very important for two reasons. First, a significant fraction of the data that will flow over the Internet will not be of a real-time nature; it will be bulk-transfer applications (e.g., image transfer) that want as much bandwidth as is currently available. These transfers will be able to use Vegas to compete against each other for the available bandwidth. Second, even for a real-time connection, it would not be unreasonable for an application to request (and pay for) a minimally acceptable bandwidth guarantee, and then use a Vegas-like end-to-end mechanism to acquire as much additional bandwidth as the current load allows.

As another example, selective ACKs [14, 15] (as opposed to cumulative ACKs) have been proposed as a way to decrease the number of unnecessarily retransmitted packets and to provide information for a better retransmit mechanism than the one in Reno. Although the selective ACK mechanism is not yet well defined, we make the following observations about how it compares to Vegas. First, it only relates to Vegas's retransmission mechanism; selective ACKs by themselves affect neither the congestion nor the the slow-start mechanisms. Second, there is little reason to believe that selective ACKs can significantly improve on Vegas in terms of unnecessary retransmissions, as there were only 6KB per MB (0.6%)unnecessarily retransmitted by Vegas in the Internet experiments. Third, selective ACKs have the potential to retransmit lost data sooner on future networks with large delay-bandwidth products. It would be interesting to see how Vegas and the selective ACK mechanism work in tandem on such networks. Finally, we note that selective ACKs require a change to the TCP standard, whereas the Vegas modifications are an implementation change that is isolated to the sender.

5.4.3 Independent Validation

An important part of experimental science is the validation of results by independent parties. Peter Danzig's group at the University of Southern California published a paper [1] confirming that TCP Vegas yields higher network throughput and transfers bytes more efficiently than TCP Reno. Their Internet experiments showed that Vegas achieved between 20 and 300% higher throughput depending on the TCP version implemented on the receiving host. With Reno receivers, Vegas sources averaged 20% higher throughput than Reno sources; with Tahoe receivers, Vegas sources averaged 300% higher throughput than Reno sources. This is a significant result, as Danzig points out that most receivers in the Internet are running Tahoe implementations of TCP. Their experiments also show that with 90% confidence Vegas retransmits fewer segments—1/5th as many retransmissions—and lowers RTT average and variance.

CHAPTER 6

Experiences with x-Sim

The simulator and its tools are an important contribution of this work. The ability of x-Sim to run real protocol code allows us to analyze existing protocol implementations under realistic conditions. This is an important feature of the simulator, as newer implementations of protocols not only include desirable features but also sometimes introduce performance bugs. The first half of this chapter documents our experiences analyzing the latest BSD implementation of TCP. This analysis uncovered serious performance problems with this implementation of TCP and led to solutions, which when implemented, increased its performance by more than 20%.

Although there are excellent books on simulation providing good general guidelines, it is not clear how to map these guidelines to the area of network simulation. Furthermore, just as important as the guidelines is being able to quantify the effect of ignoring them. The second half of this chapter provides a set of guidelines for network simulation and provides examples that help to quantify the result of ignoring them.

6.1 Analysis of BSD4.4-Lite TCP

This section presents a case study illustrating how to do performance debugging with x-Sim. While the work with TCP Vegas was underway, a new BSD implementation of TCP—TCP Lite—was released. We ported this release to the x-kernel with the goal of comparing its performance to that of x-kernel's implementation of TCP Reno and TCP Vegas. Early simulation results showed that TCP Lite performed significantly worse than TCP Reno, the TCP implementation used to measure the benefits of using TCP Vegas. In one simple simulation scenario—when no other traffic is present—TCP Reno sends one MByte of data in 8.2 seconds (123 KB/s)

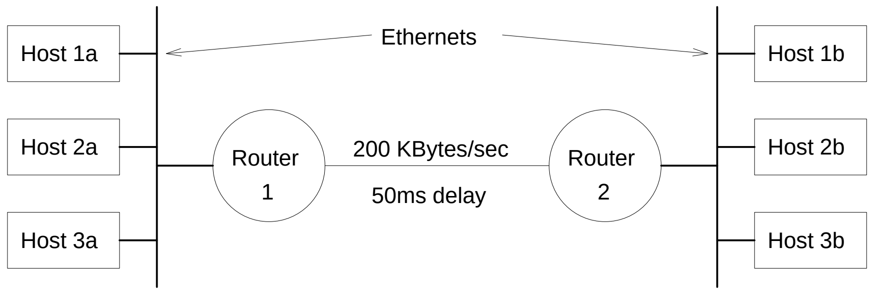


Figure 6.1: Network configuration for simulations.

and TCP Vegas sends one MByte in 6.2 seconds (166 KB/s), while TCP Lite takes 14.6 seconds (70 KB/s). TCP Lite's performance degradation is also seen when the data transfers share the bottleneck router with background traffic.

TCP Lite was analyzed in the simulator to find the reasons for its reduced throughput. The first step of the analysis involved examining the graphical and textual traces of the simulations (described in the next section), which helped find the problem areas quickly. This was followed by a detailed inspection of the code to find the exact problems and to ascertain that the problems were not created during the porting of the code to the x-kernel. Finally, the code was modified to fix the exposed problems, and tested to gauge the effect of the fixes. The fixed version of TCP Lite showed a throughput increase of up to 21% under realistic conditions.

The rest of this section describes and analyses the problems found in TCP Lite, suggests fixes for these problems, and measures the effect these fixes have on the throughput and number of retransmissions. The effect of these fixes depend on the particular details of the connection, such as RTT and available network bandwidth. For example, the problems in TCP Lite may have no effect on a connection over a local area network, yet have a dramatic impact on an Internet connection.

6.1.1 Network Configuration

The main simulation configuration consisted of two Ethernet LANs connected by two gateways through a 200 KB/s link with a 50ms delay (see Figure 6.1). Except for the TCP connections that are part of the simulated background traffic, the send

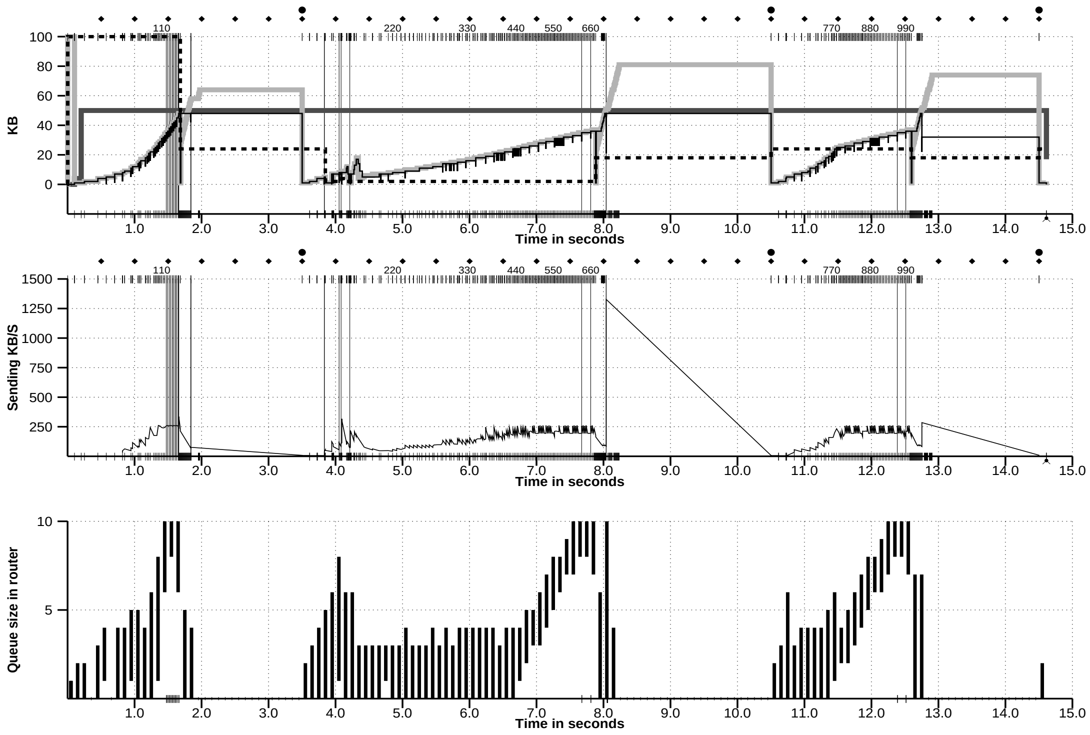


Figure 6.2: TCP Lite with No Other Traffic (Throughput: 70 KB/s).

buffer in each TCP connection was set to 50KB. Using a size that is too small would have limited the transfer rate. Note that even though there is an optimal buffer size which maximizes the throughput (by minimizing the losses), it is not realistic to use it since it is a function of the available bandwidth, which is not known under normal circumstances.

Figure 6.2 shows the trace graph of an isolated TCP Lite connection transferring one Mbyte of data, and Figure 6.3 shows the trace graph of a TCP Reno connection under the same conditions. In both figures, the topmost graph gives the window information, as described in Section 2; the middle graph shows the average sending rate, calculated from the last 12 segments; and the bottom graph shows the average queue length at the bottleneck router.

In these tests, the TCP Lite connection performs more than 70% worse than the TCP Reno connection. A detailed analysis was carried out to find the cause of TCP Lite's performance problems, and to ascertain that they were not introduced

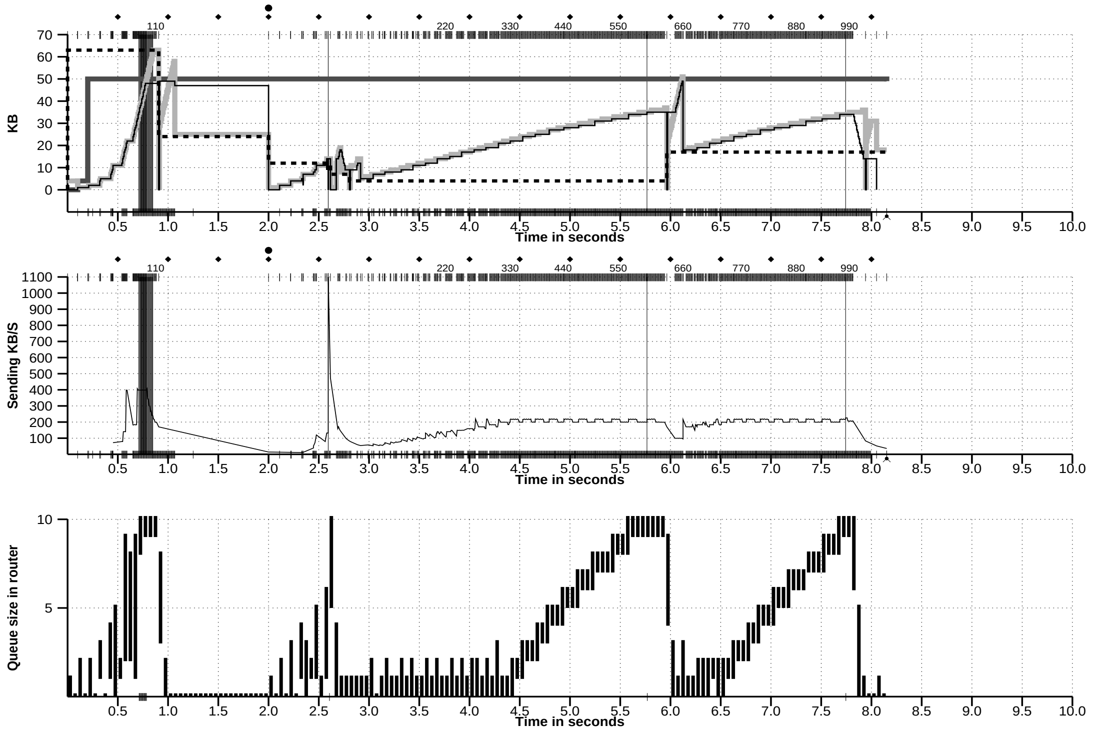


Figure 6.3: TCP Reno with No Other Traffic (Throughput: 123 KB/s).

during the porting of the BSD code to the x-kernel. The performance problems found in TCP Lite also manifest themselves under other simulation configurations. For example, experiments with a distinguished TCP connection sharing the network with traffic generated from *tcplib* show that the distinguished connection does 14% worse when it is running TCP Lite than when it is running TCP Reno.

The format of the analysis will be first to deduce the problem from the graphical traces, then to analyze it further through the textual traces and code inspection. For each problem area, we also describe its fix, and the effect of the cumulative fixes on TCP Lite's performance.

6.1.2 Error in Header Prediction Code

One of the first things to notice in the trace graphs of TCP Lite is the large number of retransmit timeouts. These timeouts are represented by the black circles at the top of the first graph in Figure 6.2 and occur at 3.5, 10.5 and 14.5 seconds. The Fast

Retransmit and Fast Recovery mechanisms seem unable to do their job, which is to prevent retransmit timeouts and to keep the pipe full. Looking at the trace graphs of TCP Reno in Figure 6.3, one can see that after the losses associated with the initial slow-start (resulting in a timeout at 2.0 seconds), Reno recovers from losses without a retransmit timeout, unlike TCP Lite.

The source of the problem can be observed by looking at the top graph of Figure 6.2 at around 8 seconds. Right before the 8 second mark, there are two thin vertical lines indicating future retransmissions, and right after that the congestion window and the UNACK_COUNT lines suddenly go down and up (top graph, light gray and black lines). This signals that 3 duplicate ACKs were received, and that the Fast Retransmit and Fast Recovery mechanisms went into action.

A little after 8 seconds, the UNACK_COUNT line goes straight down and up again. The fact that it went down signifies that a packet was received acknowledging some data, and the length of the lines tells us that it acknowledged more than 20KB. The congestion window should have been fixed at this point (made equal to the threshold window), since it was inflated to allow the pipe to stay full. However, the congestion window was not fixed, and more than 20KB of data were sent at that time (the amount of data acknowledged), creating a huge spike in the sending rate (middle graph).

The textual traces show that the packet whose acknowledgment should have triggered the reduction of the congestion window was handled by the header prediction code. This is the root of the problem. The header prediction code is only supposed to handle packets that involve little work, and it does not check for inflated congestion windows, which are, after all, a rare occurrence.

The fix is to add one more test to the part of the header prediction that handles pure ACKs for outstanding data, replacing

```
if (tlen == 0) {
        if (SEQ_GT(tHdr.th_ack, tp->snd_una) &&
            SEQ_LEQ(tHdr.th_ack, tp->snd_max) &&
            tp->snd_cwnd >= tp->snd_wnd) {
```

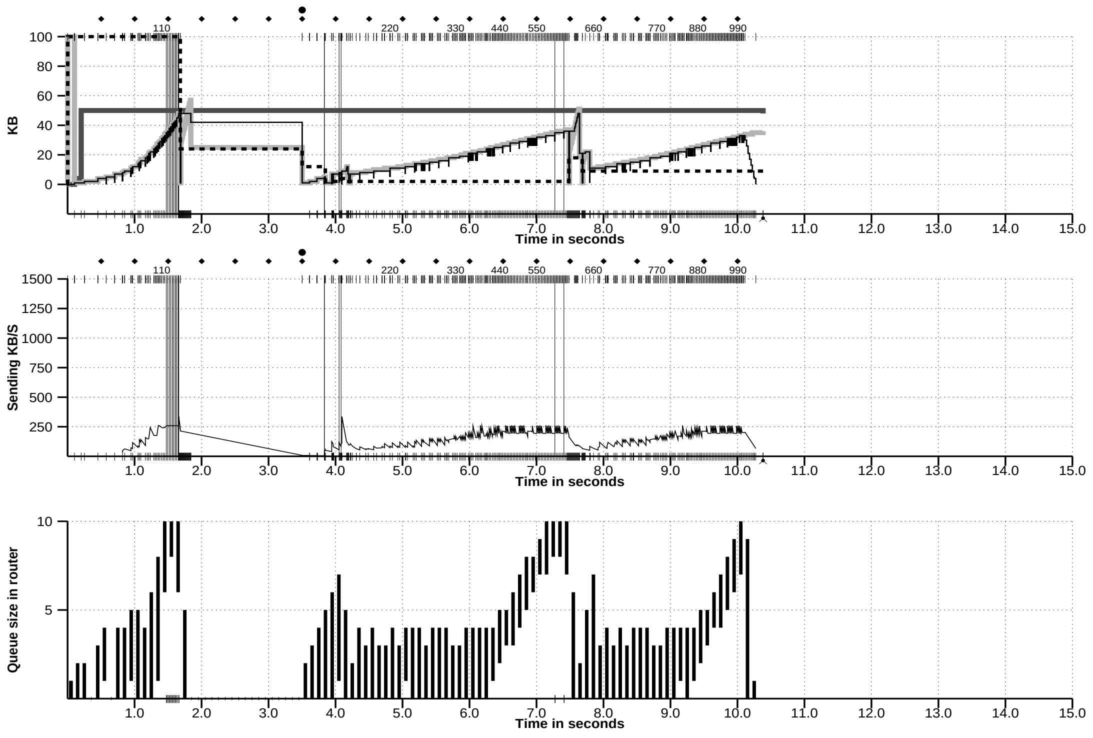


Figure 6.4: TCP Lite.1 with No Other Traffic (Throughput: 99 KB/s).

with

```
if (tlen == 0) {
        if (SEQ_GT(tHdr.th_ack, tp->snd_una) &&
            SEQ_LEQ(tHdr.th_ack, tp->snd_max) &&
            tp->snd_cwnd >= tp->snd_wnd
            && tp->t_dupacks < tcprexmtthresh) {
```

The last test (`tp->t_dupacks < tcprexmtthresh`) checks whether the number of duplicate acknowledgments that have been received is less than the Fast Recovery threshold (usually 3). If it is not, the test fails and the header prediction code is skipped.

The behavior of TCP Lite with this fix (Lite.1) is shown in Figure 6.4. Even though the performance increases considerably for this specific simulation, where TCP is running by itself (no background traffic), there is an average 2% decrease in throughput—as compared to the original implementation of TCP Lite— on the more complex simulations which also include *tcplib* traffic. The reason why Lite.0 (the original version) performs better than Lite.1 when there is background traffic is as follows. The effect of the problem—not fixing the congestion window size after losses—is proportional to the window size when the losses occur. When TCP Lite is running by itself, it has a large window when the losses occur, and the problem causes a congestion window that is large enough to result in losses and a timeout. When TCP Lite is sharing the bandwidth with background traffic, its window size is much smaller, and the problem causes a congestion window that is not large enough to result in losses. Furthermore, it turns out that the larger congestion window size can partially neutralize the problem described in Section 4.5: under some circumstances, TCP decreases its congestion window more than necessary. These results help to illustrate just how complex and nonintuitive TCP behavior can be.

6.1.3 Suboptimal Retransmit Timeout Estimates

The retransmission timeout value (RTO) calculation in TCP Lite closely follows the code described in the updated version of Jacobson's '88 paper[13]. The RTO is based on a, an average round-trip time (RTT) estimator, and d, a mean deviation estimator of the RTT, as follows:

$$rto \leftarrow a + 4d$$

Given m, a new RTT measurement, the estimators are updated as follows:

$$Err \equiv m - a$$

$$a \leftarrow a + g_0 Err$$

$$d \leftarrow d + g_1(|Err| - d)$$

The values chosen by Jacobson for the gain parameters are $g_0 = .125 = \frac{1}{8}$ and $g_1 = .25 = \frac{1}{4}$, which allow the use of integer arithmetic by keeping scaled versions of a and d. Jacobson multiplies both sides of the equations by the following factors:

$$2^3 a \leftarrow 2^3 a + Err$$

$$2^2 d \leftarrow 2^2 d + (|Err| - d)$$

Then, defining $sa = 2^3 a$ and $sd = 2^2 d$ to be scaled versions of a and d, this results in:

$$sa \leftarrow sa + Err$$

$$sd \leftarrow sd + (|Err| - (sd >> 2))$$

The whole algorithm can be expressed in C as:

```
m -= (sa >> 3);
sa += m;
if (m < 0)
        m = -m;
m -= (sd >> 2);
sd += m;
rto = (sa >> 3) + sd;
```

Where the RTO is in units of clock ticks.

Jacobson further describes how, in general, this computation will correctly round *rto*. Although this algorithm is a major improvement over the original algorithm described in RFC793 [26], there seems to be a problem with this algorithm based on the large delay observed in Figure 6.2, at the point before the retransmit timeout fires, as indicated by the large black circles at the top of the graph. For the first timeout at 3.5 seconds, there is a delay of 1.7 seconds between the timeout and the previous packet sent. For the second timeout at 10.5 seconds, the delay is 2.5 seconds.

Given that the average RTT is about 150ms, the timeout delays seem much longer than necessary. To gain a better understanding of the problem, we can look at the textual traces shown in Table 6.1 to see the behavior of the RTT, sa, sd and the RTO.

RTT	sa	sd	RTO
0	8	2	3
0	7	3	3
0	7	3	3
1	8	4	5
0	7	4	4
0	7	3	3
0	7	3	3
0	7	3	3
0	7	3	3
0	7	3	3
1	8	4	5

Table 6.1: Original RTO Related Values

The values shown for sa, sd and rto are taken after they have been updated based on the new rtt measurement. The RTT is measured using a clock that ticks every 500ms; hence it is usually zero except for those packets where the clock ticked in between sending the packet and receiving the acknowledgment. Also note that in practice, the RTO is not allowed to go below 2, the minimum feasible timer (see Figure 6.5).

Something seems amiss by the fact that the rto stays at 3 regardless of how often the rtt is zero. From the equations we see that when m (new RTT) equals a (averaged RTT, $a \equiv (sa >> 3)$), then sa is not modified. Furthermore, if $(sd >> 2)$ is zero, then the two lower bits of sd are not modified either. The fact that m is usually either zero or one results in the fractional bits of a and d (lower three bits of sa and lower two bits of sd) being set most of the time.

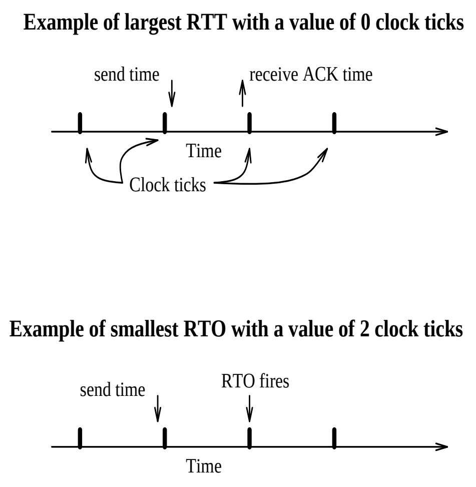

Example of largest RTT with a value of 0 clock ticks
send time
receive ACK time
Time
Clock ticks
Example of smallest RTO with a value of 2 clock ticks
send time
RTO fires
Time

Figure 6.5: RTT and RTO examples.

As a consequence, $4sd$ usually contributes 3 ticks to rto rather than the 1.75 described by Jacobson.[1] These three ticks would be of no great importance if m, the measured RTT, was measured with a finer clock, so that it usually had a larger value—3 out of 100 is only 3%, 3 out of 1 is 300%!

We now describe a simple modification to the RTO algorithm which solves the problem for the conditions of our experiment, where the RTT is either 0 or 1. This modification allows us to test the improvement in performance that is possible with a better RTO algorithm. However, we are not fully satisfied that this algorithm is the correct one under all possible scenarios and we plan to study it further.

Our modification is to use a larger scaling factor:

$$2^2 Err \leftarrow 2^2 m - 2^2 a$$

$$2^5 a \leftarrow 2^5 a + 2^2 Err$$

$$2^4 d \leftarrow 2^4 d + 2^2 |Err| - 2^2 d$$

Now, if we define $sErr = 2^2 Err$, $sa = 2^5 a$ and $sd = 2^4 d$ to be scaled versions of Err, a and d, we have:

[1]The 1.75 is a statistical average assuming a wide range of RTT values, but in practice, the RTT is usually either 0 or 1. See Figure 6.5 for the reason why the bias is needed.

$$sErr \leftarrow (m << 2) - (sa >> 3)$$

$$sa \leftarrow sa + sErr$$

$$sd \leftarrow sd + (|sErr| - (sd >> 2))$$

Then the low order bits that are not modified (and usually end up set) after m has the same value repeatedly do not affect the RTO calculation.

Since we want to be conservative when setting the RTO, and by conservative we mean that we never want to retransmit as a result of choosing too small an RTO, this implies that the RTO should always be at least two larger than the RTT. For example, the largest possible RTT with a value of x ticks has a real time length of just less than $x + 1$ clock ticks (see Figure 6.5), but the smallest possible RTO with value $x+2$ has real time length of just greater than $x+1$ ticks (again see Figure 6.5).

Note that if there is access to a more accurate clock, as in TCP Vegas, then the coarse RTO can be made more accurate since one does not have to worry about the extreme cases described in the previous paragraph. For example, if the more accurate RTO is less than 200ms, then as long as the segment is sent within the first 300ms after the coarse-grained clock ticked, the coarse rto can be set to 1 (instead of at least 2). A more accurate clock will also eliminate the large jumps in the size of the RTO which come from the RTT jumping between 0 and 1.

The whole algorithm can be expressed in C as:

```
sdelta = (m << 2) - (sa >> 3);
sa += sdelta;
if (sdelta < 0)
        sdelta = -sdelta;
sdelta -= (sd >> 2);
sd += sdelta;
rto = max( m + 2, ((sa >> 3) + sd) >> 2 );
```

RTT	original RTO	new RTO	float RTO
0	3	3	3.4
0	3	3	3.5
0	3	3	3.5
0	3	3	3.4
1	5	3	3.1
0	4	3	3.1
0	3	3	2.9
0	3	3	2.8
0	3	3	2.5
0	3	2	2.3
0	3	2	2.1
1	5	3	2.5

Table 6.2: Original and New RTO Values

The new mechanism approximates the real RTO, obtained by using floating point computations, more closely as can be seen in Table 6.2. Unlike the old RTO, the new RTO can go all the way down to a value of 2 after the RTT has a value of 0 repeatedly.

The version of TCP Lite with both the header prediction and RTO computation fixes, which we refer to as Lite.2, has a throughput of 102.8 KB/s for the 1 MByte transfer when there is no other traffic. The average throughput for the set of tests with *tcplib* background traffic is 55.9 KB/s, an increase over the original of 9%. However, this is still 6% below the throughput of TCP Reno.

6.1.4 Options and ACKing Frequency

The TCP header is normally 20 bytes long. However, one of the header fields—the options field—is variable in size, ranging between 0 to 40 bytes in length. TCP Lite's ACKing frequency is affected by the use of TCP options. When there are no options in use, TCP Lite ACKs every other packet. When options (e.g., timestamp)

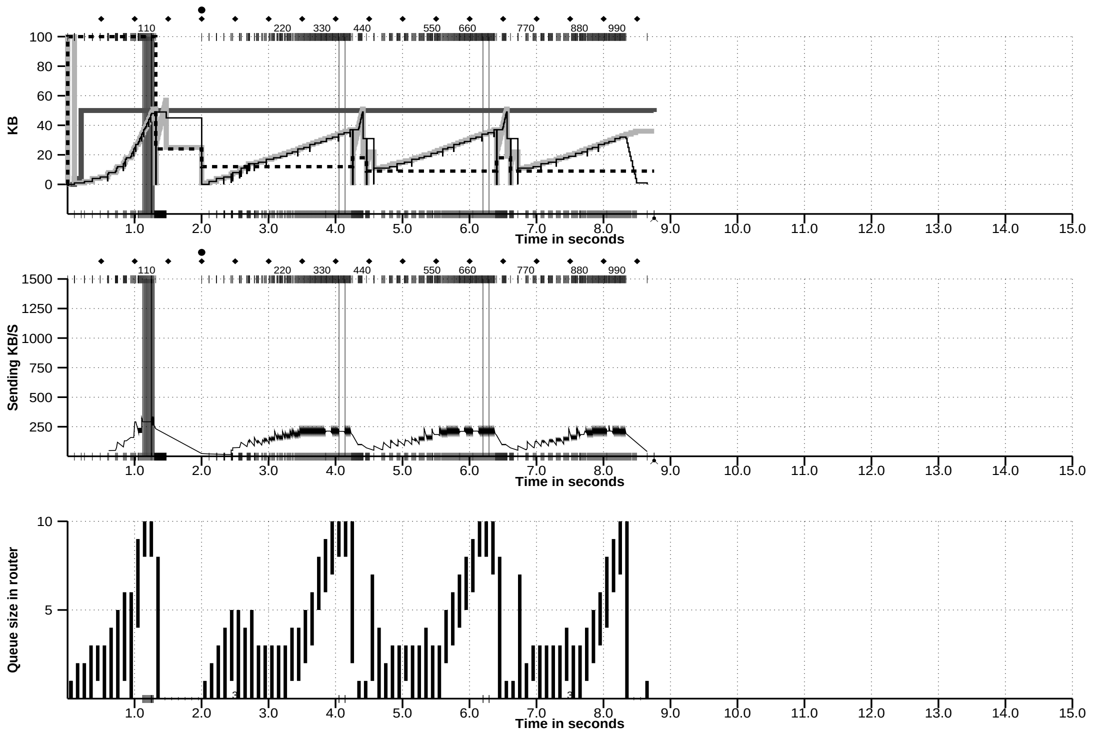


Figure 6.6: TCP Lite.3 with No Other Traffic (Throughput: 117 KB/s).

are used, however, it ACKs every third packet. The reason is that the test which decides to send an ACK is based on how much data has been received, and if it is greater than or equal to more than twice the maximum segment size, an ACK is sent. However, since TCP options take some of the available payload space, two segments that contain an option no longer hold enough data to trigger sending an acknowledgment.

As discussed below, ACKing frequency affects the growth of the congestion window during both slow-start and the linear increase period. We disabled the timestamp option as a quick way to test the effect of modifying the code which handles the ACKing frequency so it is not affected by the use of options. We refer to this version as Lite.3, and its throughput for the 1 MByte transfer experiments was 177 KB/s. However, only about half of the throughput gain was due to the higher ACKing frequency; the rest of the gain was due to the fact that the RTO happened to be 2 instead of 3 when the losses occurred, decreasing the delay until the RTO fired by half a second. In the experiment when there is also background traffic, the average throughput increased by 5%, but the losses increased by 28% when compared to TCP Lite.2.

Figure 6.6 shows the graphical traces of Lite.3 when transferring 1 MByte. The important thing to notice is how the losses are paired when they occur around 4 and 6 seconds, resulting in decreasing the congestion window twice. The reason the losses are paired is that the congestion window is increasing too rapidly—by more than one maximum segment per RTT. Before the first loss is detected, TCP Lite increases the congestion window, again resulting in an extra lost packet.

The reason the congestion window is increasing so fast is the extra $1/8^{th}$ of a maximum segment being added to it:

```
if (cw > tp->snd_ssthresh)
        incr = incr * incr / cw + incr / 8;
```

By removing this $1/8^{th}$ increase:

```
if (cw > tp->snd_ssthresh)
        incr = incr * incr / cw;
```

the losses are not paired any more, so the congestion window is decreased only once. In the experiments with background traffic, running TCP Lite.4, which includes the four previous modifications (header prediction fix, RTO fix, no timestamps, and eliminating the extra $1/8^{th}$ increase) results in an average throughput of 62.3 KB/s. This represents a 21% increase over the original TCP Lite, and a 6% increase over Lite.3 .

6.1.5 Linear Increase of the Congestion Window and ACKing Frequency

TCP should perform a multiplicative decrease of the congestion window when losses are detected, and a linear increase while there are no losses detected. Due to the bugs described above, TCP Lite follows no specific guidelines on how fast to increase the congestion window. If there are options, the acking frequency is reduced and the window increases more slowly. By adding the extra $1/8^{th}$ of a maximum segment size to the congestion window, the rate of increase is now a function of the window size (which means the increase is exponential).

It is easy to specify what should be the upper bound to the rate of increase for the congestion window during the linear increase mode: the congestion window should not be increased by more than one maximum segment size per RTT. Increasing the congestion window at a faster rate will result in the loss of two or more segments per RTT when the queue at the bottleneck router is full. This may result in a timeout since the fast retransmit and fast recovery mechanisms in TCP Lite cannot always recover from the loss of two segments when the losses occur during a one RTT time interval.

ACKing frequency can be seen as a tradeoff between higher congestion and lower latency. As shown here, it also affects the rate of the linear growth of the congestion window in TCP Lite—this is due to the implementation, and it could be changed. However, it is not clear that ACKing every two or three packets puts less stress in the network than acking every packet. On the one hand, lower ACKing frequencies imply fewer packets on the network, but on the other hand, lower ACKing frequencies result in more packets being sent at one time. For example, if the receiver ACKs after receiving three packets, then the source will transmit 3 or 4 packets at one time, the fourth one if the congestion window linear increase allows one more packet to go out. If two or more connections do this at the same time, then the likelihood of losses increases. Another example resulting in increased stress to the network is a server that does not reply to requests immediately. As a consequence, all of the delayed acknowledgments are sent at the same time when a TCP timer forces their transmission every 200ms.

We modified the code, creating Lite.5, so that it ACKs after every packet rather than after every two packets. The effect on the tests with traffic was a slight increase in the throughput (3%) but a much larger increase in the losses (48%). The increase in the losses may be due to the simulation scenario—e.g., number of buffers at the router. We are planning to look into this issue in more detail, since we do not believe it has been settled. Until then, we prefer to take the conservative approach and use delayed ACKs (after every two packets) in TCP Lite.

6.1.6 Handling Big ACKs

The response of TCP Lite (and previous BSD versions) when it receives a packet acknowledging x bytes, is to send x bytes immediately, as long as the windows allow it. If x is a large number, equivalent to 4 or more packets, this behavior can result in losses due to packets being dropped at the bottleneck.

Two causes for large acknowledgments (4 or more packets) are (1) losses that result in a retransmit timeout, and (2) packets received out of order. The fast retransmit and fast recovery mechanisms try to keep the pipe full after a loss is detected after receiving three duplicate ACKs. This means that after retransmitting the lost segment, TCP keeps sending (new) data at a rate half of what it was when the loss was detected. The receiver cannot acknowledge the new data since it is missing one or more of the earlier segments. If other packets following the packet originally lost are also lost, or if the retransmitted packet is lost, the pipe will likely empty and a retransmit timeout will be needed to start sending again. After the segment that was lost is retransmitted, the receiver will now be able to acknowledge not only the retransmitted packet, but also all of the segments following the retransmitted segment which were sent to keep the pipe full.

The fix to prevent sending too many segments at once is very simple. When a large acknowledgment is detected, if the threshold window is smaller than the congestion window, set it to the value of the congestion window. Then set the congestion window to a lower value so only 2 or 3 segments are sent at one time. As new acknowledgments are received, the congestion window will then increase exponentially to the correct level, as specified by the threshold window. An example of the fix follows:

```
if (acked >= 3*tp->t_maxseg  &&
    (tp->snd_cwnd - (tp->snd_nxt - tp->snd_una)) > 3*tp->t_maxseg) {
  if (tp->snd_cwnd > tp->snd_ssthresh)
```

```
      tp->snd_ssthresh = tp->snd_cwnd;
    tp->snd_cwnd = (tp->snd_nxt - tp->snd_una) + 3*tp->t_maxseg;
  }
```

This code needs to be inserted after the window information is updated. Adding this fix to Lite.4 (resulting in Lite.6) did not affect the results very much (less than 2%), indicating that the problem is rare, at least under the conditions of our experiments.

6.1.7 Final Details

There is one final detail. The test that checks if the congestion window needs to be fixed because it was inflated trying to keep the pipe full is slightly wrong. The original test

```
tp->dupacks > tcprexmtthresh
```

should be changed to

```
tp->dupacks >= tcprexmtthresh
```

However, this will probably not have much of a practical effect.

6.1.8 Comparing the Different Versions of TCP Lite

This section compares the average throughput and average losses of the different versions of TCP Lite. The experiment consisted of running 20 simulations in which there is a distinguished 20 second transfer sharing the bottleneck link with *tcplib* generated traffic running over the same version of TCP as the distinguished transfer. The background traffic uses between 30 and 80% of the bottleneck bandwidth.

There are 7 versions of TCP Lite, denoted by Lite.0 to Lite.6, which represent the different cumulative fixes applied to TCP Lite. Table 6.3 describes the fixes applied to the different versions. The purpose of Lite.3, which does not use the timestamp option, is to see the effect of fixing the code which decides when to send

TCP Version	Fixes
Lite.0	None. This is the original TCP BSD4.4-Lite code
Lite.1	Header prediction fix
Lite.2	Previous + RTO fix
Lite.3	Previous + not using timestamp option
Lite.4	Previous + removing the extra 1/8 increase of the congestion window
Lite.5	Previous + acking on every packet
Lite.6	Lite.4 + fix to prevent sending to much at one time due to large ACKs

Table 6.3: Description of Different Versions of TCP Lite.

an acknowledgment so the acknowledgment frequency is not affected by the use of options.

Table 6.4 shows the average throughput, the throughput ratio with respect to Lite.0, and the percent of bytes retransmitted. The numbers shown are the averages of the 20 runs using different background traffic patterns. As can be seen from the table, Lite.4 does 21% percent better in terms of throughput that the original code (Lite.0). Based on the current experiments, we recommend the modifications to TCP Lite corresponding to Lite.4 (keeping in mind the need for more testing of the RTO modifications).

Our version of Reno has lower throughput (5%) and higher losses (36%) than Lite.4. The reasons are that the x-kernel's version of Reno does not delay ACKs, and it does not contain the the RTO enhancements described in section 6.1.3. When comparing to Lite.4, TCP Vegas shows an improvement of 33% under the simulation parameters, and a 15% decrease in the number of retransmitted bytes.

	Lite.0	Lite.1	Lite.2	Lite.3	Lite.4	Lite.5	Lite.6	Reno
Throughput (KB/s)	51.3	50.4	55.9	58.5	62.3	64.3	61.5	59.3
Throughput Ratio	1.00	0.98	1.09	1.14	1.21	1.25	1.20	1.16
Retransmissions	3.7%	4.5%	3.6%	4.6%	3.3%	4.9%	3.8%	4.5%

Table 6.4: 20sec Transfer with *tcplib*-Generated Background Traffic.

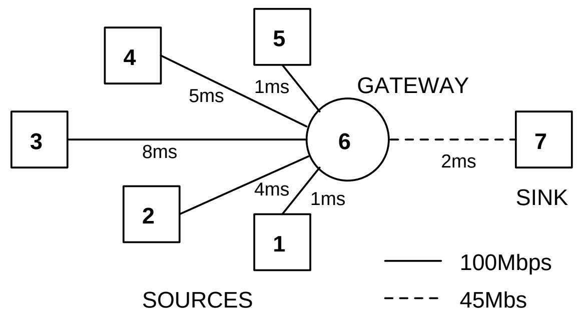


Figure 6.7: Simple Simulation Network.

6.2 Guidelines for Network Simulation

There is a wealth of literature on the subject of simulation; see [17] and [21] for a good overview. Among the issues presented in the literature are general principles all simulation experiments should follow to insure the validity of the results. Typically, these principles are presented in general terms, and it is up to the experimenter to apply them to their particular case.

This section goes beyond these general principles to present some very concrete guidelines dealing with *network* simulations, in particular. Most of these guidelines involve the application of a well known rule, but they are worth enumerating because they contain realistic evidence of what can happen when these rules are not followed. That is, these examples help to quantify the value of the general simulation principles, as well as the cost of ignoring them.

Our experience with the networking literature is that in many cases some of these guidelines are being ignored. In some cases, this is due to inexperience in the area of network simulation. In other cases it is an intentional decision, either as a result of misjudging their importance or as a result of practical concerns such as a lack of resources.

Most of the examples used in this section were obtained from the two simulation networks shown in Figures 6.7 and 6.8. The first, shown in Figure 6.7, consists of a simple topology with 7 nodes. The second simulation network, shown in Figure 6.8, is much more complex. It models a subset of one of the Internet backbones, with

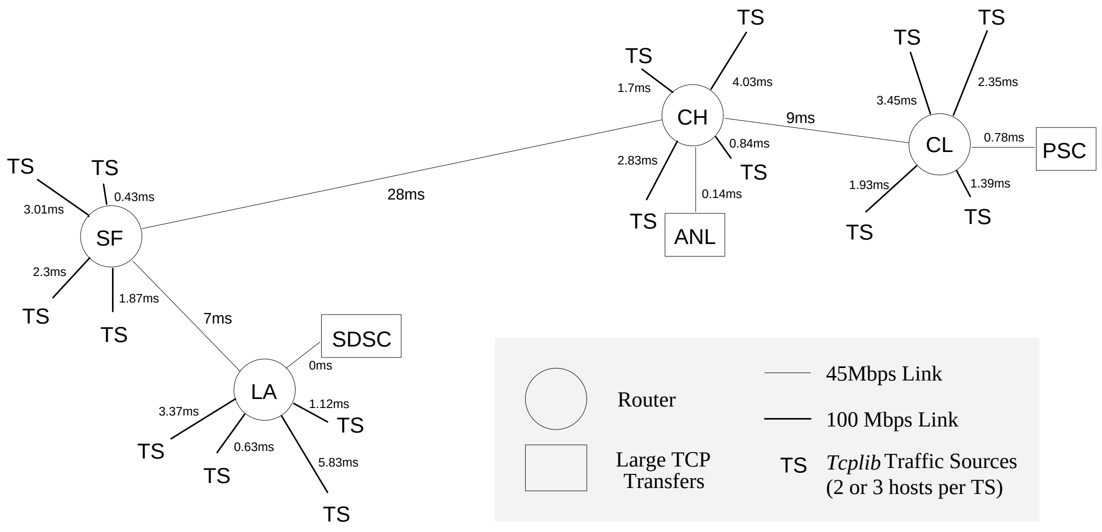


Figure 6.8: Complex Simulation Network.

major routers at Los Angeles, San Francisco, Chicago and Cleveland. Three supercomputer centers (rectangular boxes) provide sources of large "bulk data" transfers. There are 16 traffic sources, each consisting of two or three hosts, each of which is running *tcplib* based traffic. The resulting traffic consists of nearly a thousand new connections being established per simulated second, where each connection is either a Telnet, FTP, SMTP or NNTP conversation.

6.2.1 Sensitivity to Network Parameters

Protocols and communication networks generally deal with bits and packets, both of which are discrete entities. Many protocol components, such as retransmit timers, also behave in a discrete manner. Many network parameters, such as the number of buffers at a router, are also discrete quantities. Due to this discrete nature, small changes in the parameters defining a network experiment can have a considerable effect in the outcome of an experiment. The following two examples illustrate the sensitivity of a simulation to different settings of such values.

The first example involves the number of buffers at a router. Here we are running an experiment that simply measures the throughput TCP can achieve across a simulated network operating under load. In the BSD implementation of TCP, time is measured by counting the number of times a timer, which goes off every 500ms, has fired. As data is sent over the connection, losses due to congestion result in a temporary halt in the transmission. In some cases, if there are multiple losses, the transmission is restarted only after a retransmit timeout. This delay generally lasts for two ticks of the 500ms clock. There are situations where for some number of buffers at the router, the transmission hiatus (gap) starts right *before* the timer fires, and since the hiatus only lasts for two ticks of the clock, the result is the hiatus lasts for about 500ms. Increasing the number of buffers by just *one* can result in delaying the start of the transmission hiatus until *after* the timer fires, so now two clock ticks represent about 1000ms rather than 500ms. For example, we have seen simulations where increasing the number of buffers from 22 to 23 leads to a 34% decrease in throughput. The same effect can also be obtained when keeping the number of buffers constant, but by changing the time when the test starts relative to the tick of the 500ms clock.

A second example of the unexpected effects of changes in the simulation parameters is seen when running two TCP transfers at the same time. In this case, the experiment consists of two competing TCP transfers, with some delay between the start of each transfer. For a particular simulation, a delay of 170ms between the start of the two transfers resulted in a throughput of 76.1KB/s for the first transfer and 76.2KB/s for the second transfer. Changing the delay to 150ms results in throughputs of 87.8KB/s and 75.9KB/s, respectively, a change of more than 15%.

These examples demonstrate the sensitivity of a simulation to seemingly trivial differences in the settings of certain network parameters. The guideline is to vary the simulation parameters over a range as large as possible. The basic principles behind it are: (1) simulate the right cases, (2) perform sensitivity analysis.

6.2.2 Analyzing Results

One should also look at the results from every experiment to catch any instances in which the results vary greatly from the average. These instances may expose problems in the experiment, problems in the algorithms, or problems with the im-

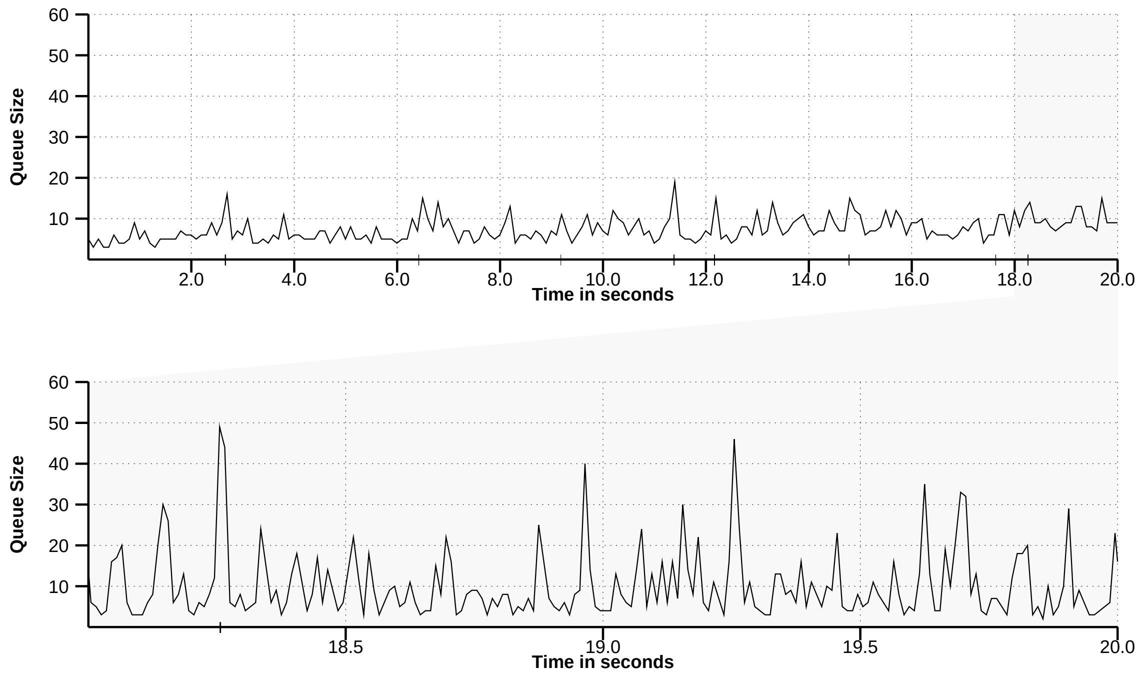


Figure 6.9: Graph of queue size showing information lost due to averaging.

plementation that surface only sporadically. Looking just at averages may result in missing vital information. In our experiments, we depend heavily on graphs that allow us to quickly see the important results from all the experiments, not just the averages.

Graphs of the average queue size (number of buffers in use) at a bottleneck router is the first example of where looking just at the averages may be misleading. One of the important uses of a simulator is to experiment, observing the effects on the experiment of modifying the parameters. This type of experimentation helps develop intuition that can be used to both find problems and the solutions. If we are interested in congestion control mechanisms at the router, it is very important to have a clear understanding of queue behavior under different traffic patterns. The top graph in Figure 6.9 shows the average queue size during a 20 second period of a simulation, where each point represents the average queue length over a 100ms interval. Note that under the conditions of the simulation, there are between three

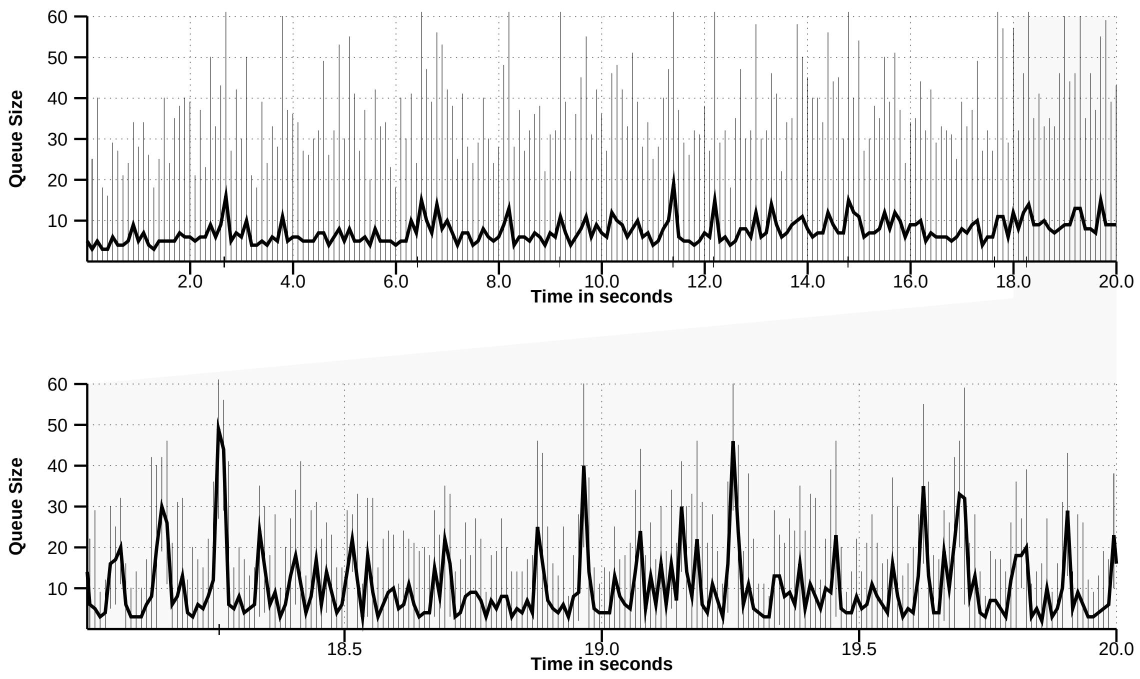


Figure 6.10: Graph of queue size at bottleneck including min-max bars.

and ten thousand packets arriving at the bottleneck router every second, so it is not practical to show the graph of the instantaneous queue size.

This graph indicates that the queue size does not vary much, and that it stays under 20 buffers. However, the bottom graph, which shows the time interval between 18 to 20 seconds but averages over a 10ms interval (rather than the 100ms interval), shows that the queue size reaches all the way to 50 and that it goes over 20 quite often. At this level of detail, one can see a markedly different behavior of the queue. Obviously, what one should do is to show both the maximum and minimum queue size (over the average interval) together with the average. Figure 6.10 shows the same graphs as Figure 6.9 but with the added information. Each vertical bars represents the queue size range over the measurement interval (100ms for the top graph, 10ms for the bottom graph). The top graph now tells us a whole new story—even though the average queue size is small, it varies dramatically during each interval. The mechanism of showing the range together with the average allows

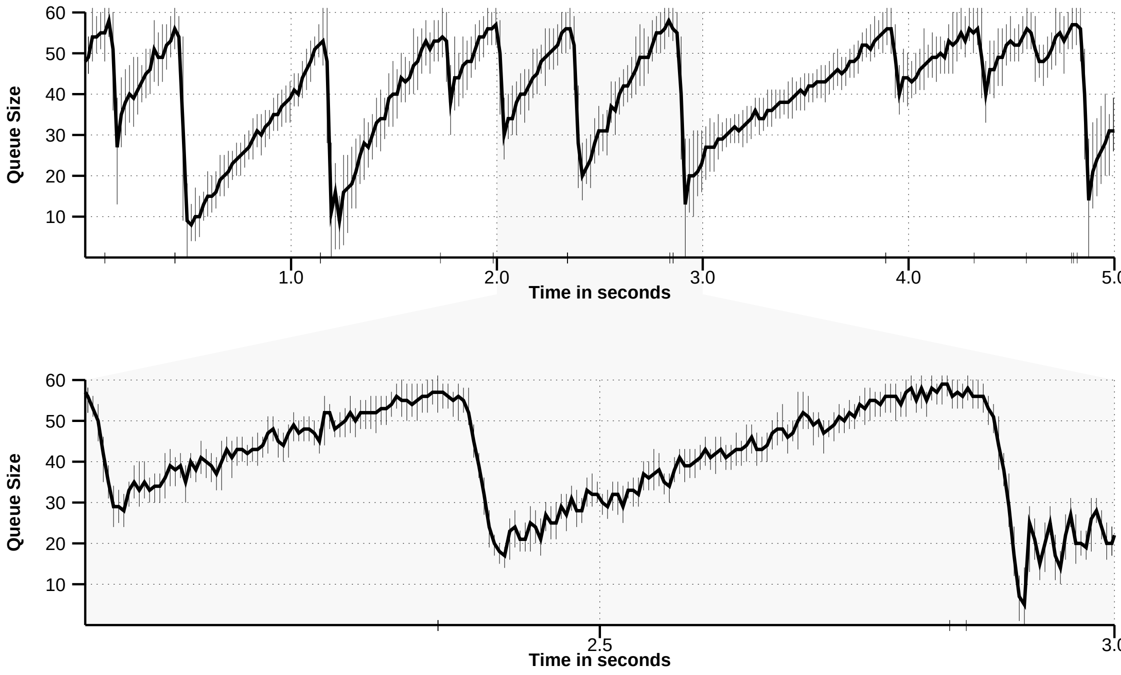


Figure 6.11: Queue behavior during four large TCP transfers.

us to get a much clearer picture, or at least keeps us from coming to the wrong conclusions.

The basic principle is to look at distributions and extremes, not just means, when analyzing results.

6.2.3 Realistic Traffic Sources

Traffic sources form an integral part of most network simulations—networks without traffic are not very interesting. Traffic sources are clearly needed when studying the behavior of routers under load, or when analyzing the performance of protocols under realistic conditions. The most common ways to simulate this traffic has been either through Poisson sources, or by having multiple "bulk transfer" TCP connections.

There are several problems with Poisson sources. First, it has been shown that Poisson sources fail to accurately model either LAN traffic [22] or WAN traffic [23]. Moreover, Poisson sources are non-reactive—they behave the same way regardless

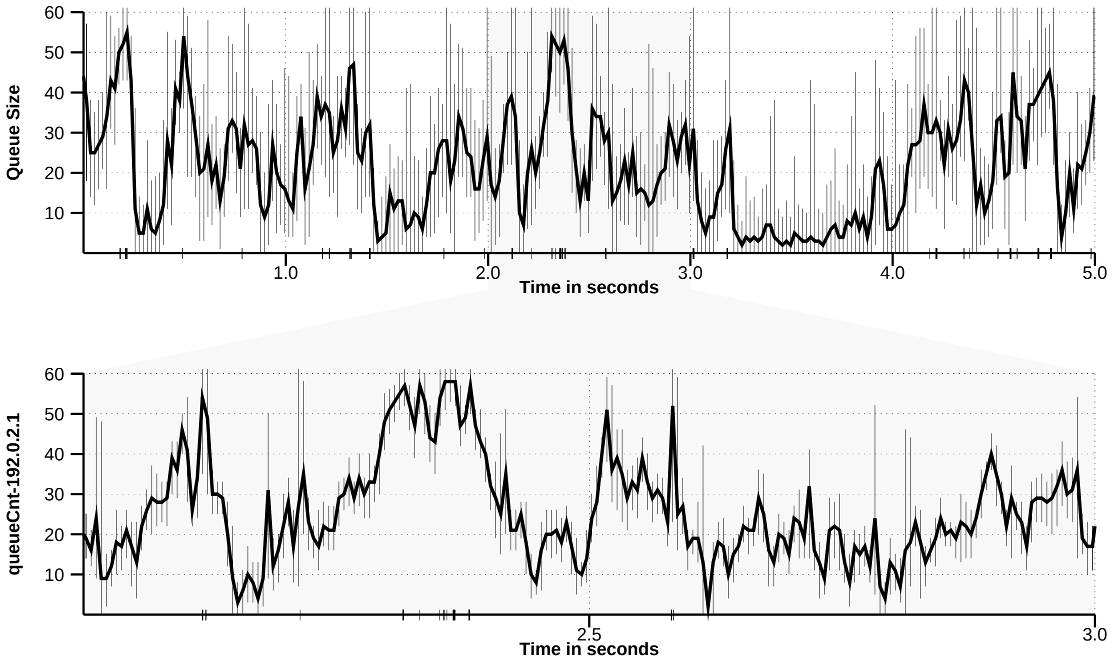


Figure 6.12: Queue behavior during two large TCP transfers and *tcplib* traffic.

of the network behavior. For example, unlike TCP sources which modify their sending rate under the presence of losses, Poisson sources do not modify their rate, or any aspect of their behavior for that matter, when packets are dropped due to congestion. Another example of non-reactive traffic sources are playback sources, which recreate (play back) previously recorded traffic traces. These traffic traces are at the packet level, and consist of the time when a packet entered the network and the packet's size.

The problem with non-reactive sources is that there is no way to measure the effect new connections have on the background traffic. This is an important issue, as new protocols should not be adopted before their effect on current traffic is known.

The other common way of simulating traffic—starting multiple bulk transfers—does produce reactive traffic, but the dynamics of this type of traffic are very different from those of real traffic. For example, Figure 6.11 shows the queue size at the router of the network in Figure 6.7 when nodes 1-4 are bulk transfer TCP sources. The graph shows a simple pattern and no information is gained by changing scales in the bottom graph. Compare this graph to the graph in Figure 6.12, in which hosts 1, 3 and 4 are *tcplib* traffic sources, and hosts 2 and 5 are bulk transfer TCP sources. The behavior at the router is totally different, not as predictable as in the previous graph, and changing scales shows new detail.

The degree of dissimilarity between the two graphs should serve as a warning against using bulk transfers as a traffic source when simulating wide-area networks. Moreover, studies of congestion control mechanisms should include traffic sources that are more realistic than either Poisson or bulk transfers.

A final issue regarding realistic traffic sources is the need for traffic reproducibility. This means that a traffic source should always produce the same traffic pattern when given the same set of initial parameters (such as initial seed and connection interarrival times) regardless of the underlying protocols, the number of nodes, or the type of network. Reproducing a traffic pattern does not mean that a packet containing x bytes is transmitted at time t. Instead, it refers to the structural characteristics of the traffic. For example, when dealing with *tcplib* based traffic, a traffic pattern is specified by: the time a conversation starts, the type of the conversation (FTP, Telnet, SMTP and NNTP) and the parameters for that type of conversation. An example of the parameters for an FTP conversation are the number of bulk transfers and the length of each transfer. Note that such traffic sources are not inherently reactive. However, running them on top of TCP produces reactive traffic as a consequence of TCP's congestion control mechanisms.

Traffic sources that meet this definition of reproducibility allow us to compare things such as the effect on the traffic's throughput or losses when it is running on top of two different protocols.

The basic principles behind these discussions are to always use realistic traffic sources and justify any traffic distributions used in the experiments.

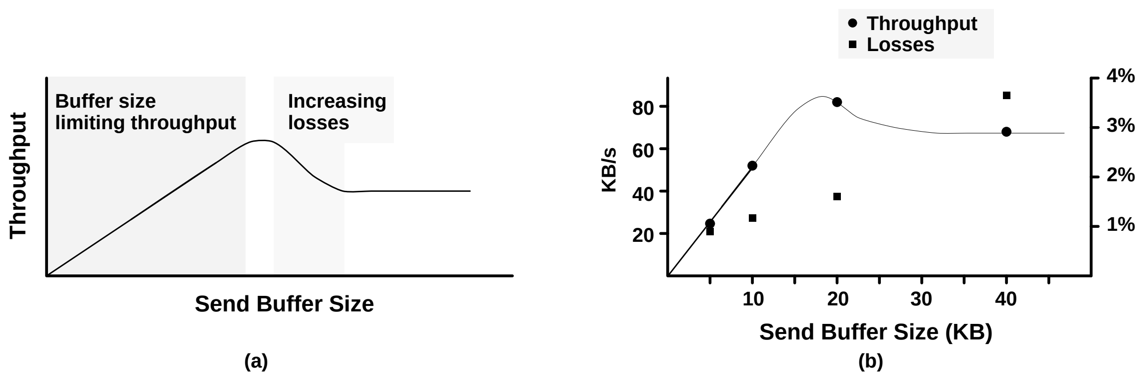


Figure 6.13: Throughput vs. Send Buffer Size.

6.2.4 Insider Knowledge

When doing experiments involving TCP transfers, it has been a common practice to set TCP's send buffer size based on the bandwidth delay product of the connection. After all, why use more buffering that is really needed? The size of the send buffer limits the maximal throughput a connection, with a given RTT, can achieve. The result of having this upper limit below the available bandwidth is obvious: the achieved throughput is not necessarily a reflection of the protocol or the network, but is an artificially imposed limit.

As the send buffer size increases, there is a corresponding increase in the throughput achieved by the TCP transfer. This relationship continues until a the size of the send buffer is slightly larger than the average available bandwidth-delay product—using a few buffers at the bottleneck router allows the connection to take advantage of transient increases in the available bandwidth. What happens after that point depends on the TCP implementation. In the BSD implementations of TCP (i.e. TCP Reno), the throughput starts to decrease, and does so until it reaches a point where additional increases of the send buffer size have no effect on throughput (see Figure 6.13a).

The decrease in the throughput as the buffer size increases is the result of the bandwidth probing mechanism in the BSD-based implementations of TCP. These implementations increase the window size continuously, by about one packet per

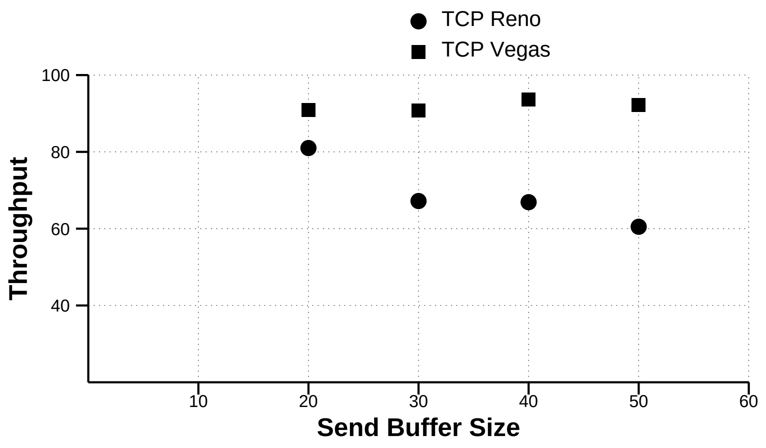


Figure 6.14: Throughput vs. Send Buffer Size.

RTT, until the buffer size limit is reached or packets are lost. As the window size increases from its optimal point, which only uses a few "extra" buffers at the bottleneck, the TCP connection uses more and more "extra" buffers at the bottleneck. The larger the number of these "extra" buffers, the higher the likelihood that the router will not be able to absorb a packet burst. Since some of these bursts will be by the connection itself, the larger number of "extra" buffers used by a connection increase the likelihood that the connection will have losses. When the losses are detected, the congestion control mechanism in TCP reduces the window size by half, reducing the throughput.

Figure 6.13b shows the average throughput achieved by TCP Reno as a function of the send buffer size during a series of transfers over the Internet between two fixed hosts. Notice how the losses increase as the send buffer size increases. Given all this, using a send buffer size which limits the throughput leads to misleading comparisons between protocols, as they both perform equally under this condition. Similarly, setting the send buffer to its optimal size also leads to misleading comparisons, as real applications will not be using this optimal send buffer size—it is not easy to know it as different connections will have different optimal sizes, and this optimal size will change with time.

This is another example where varying the parameters of the experiment leads to useful insights and comparisons. For example, a set of experiments comparing TCP Reno with TCP Vegas (see Section 5.2.3) showed that TCP Vegas was not sensitive to the send buffer size—as long as it is big enough so that it is not the limiting factor. Figure 6.14 shows the average throughput for a set of simulations with different send buffer sizes. Note that TCP Vegas was not affected by the buffer size, whereas TCP Reno's throughput decreases by more than 30%.

The basic principle to remember is that simulation results are constrained by the values of the input parameters. Any bias in the choice of these parameters will affect the results.

6.2.5 Byte versus Time Transfers

Throughput is generally measured by recording the time required to send a specific amount of data; e.g., if it takes 10 seconds to send 1 MByte of data, then the throughput is 100 KBytes/sec. When measuring the difference in throughput between two protocols, one of which is generally faster than the other, there is the possibility of erroneous measurements in some cases.

For example, if the experiment includes background traffic, there may be a systematic tendency for the level of traffic to either increase or decrease during the duration of the transfer we are timing. If the level increases during the transfer, then the slower protocol will be unfairly penalized because it had to compete with higher levels of traffic at the end of the test, whereas the faster protocol did not. Alternatively, if the level of the background traffic tends to decrease during the duration of the test, then the slower protocol will be measured as being faster—relative to the faster protocol—than it really is.

Instead, another way of measuring the throughput is to transfer as much data as possible for a certain amount of time. This way both protocols will encounter the same background traffic for the duration of the transfer. The differences in the measured throughput between the two methods can be considerable, depending on the characteristics of the traffic. For example, in one series of experiments resulted in a 56% difference in the throughput between two protocols when sending 1 MByte of data but only a 28% difference when sending for 30 seconds. Clearly, the preferred method should be to use timed transfers rather than length transfers.

6.2.6 Unwanted Synchronicity

It is very easy for simulations to lose a lot of the variability (randomness) that is inherent in the real world. This can be either a design flaw with the simulator or the result of badly chosen simulation parameters. An example of synchronicity in a simulation is starting all transfers at the same time. This is very easy to do in a simulator, but it would generally be an uncommon occurrence outside of a simulator.

Another example of lack of variability can occur with protocols, like the BSD implementation of TCP, where coarse timers are used to perform special processing. In this implementation of TCP, two coarse timers are used, one firing every 200ms and the other every 500ms. Most of the implementations of TCP delay ACKs, which means that when they receive a packet they do not send an acknowledgment immediately, but instead wait for more packets to arrive so they can acknowledge more than one packet at a time. The BSD implementations of TCP wait for one more packet, thereby acknowledging two packets at a time. However, since another packet may not arrive, TCP uses the 200ms timer to send any acknowledgments that were delayed. The 500ms timer is used to check if packets need to be retransmitted because they were lost.

If one is not careful in the implementation, it is easy to have these timers fire at the same time on all hosts in the simulated network. This results in periodic bursts of data as all hosts send the delayed acknowledgments or start retransmitting lost packets at the same time. Since the retransmissions occurring as a result of the 500ms timers involve the slow-start mechanism, which is a restart of the connection involving exponential growth of the sending rate, the effect is equivalent to starting multiple transfers at the same time.

A similar burstiness can occur even when the timers for each node do not all fire at the same time. The problem happens when using realistic traffic sources involving multiple TCP connections, but having only a few traffic sources, each of which is responsible for a large percent of the traffic. Then, when one of the timers fires at one of the sources, a large number of connections may send delayed ACKs or restart their transmissions. The results are similar to having a large number of traffic sources, with a group of sources firing their timers at the same time.

A tool that is very useful at finding unwanted uniform behavior is Fourier analysis [29]. This type of analysis uncovers periodic behavior which may point to possible problems and to the cause of these problems. Figure 6.15(a) shows the queue size at the router in Figure 6.7 when the only source of *tcplib* traffic is node 4. It is clear from the graph that there is periodic behavior which occurs 5 times per second (every 200ms). The fact that the spikes occur every 200ms points to the fast timer in TCP as the cause of these spikes. Note that this behavior does not appear in the graph of the average queue size, it only appears in the maximum queue size graph.

One way to eliminate these bursts is to have more traffic sources so the bursts will be distributed more evenly—assuming the traffic sources do not fire their timers at the same time. Figure 6.15(b) shows the queue behavior when there are three traffic sources. Periodic behavior is not as evident in this case, although Fourier analysis shows that it still exists. This periodicity can be decreased even further by increasing the number of sources.

Another way to eliminate some of the periodicity is to introduce node processing delays. Figure 6.15(c) shows the queue behavior when a delay, with a uniform distribution between 0 and 50μs, is introduced before a packet is sent by the node. The periodicity is not as evident as in the top graph, but it still exists and with very high peaks in some instances.

The basic principle is to put a distribution on everything that needs it. The problem is that when dealing with complex simulators, it is not always clear what things need to have distributions in them.

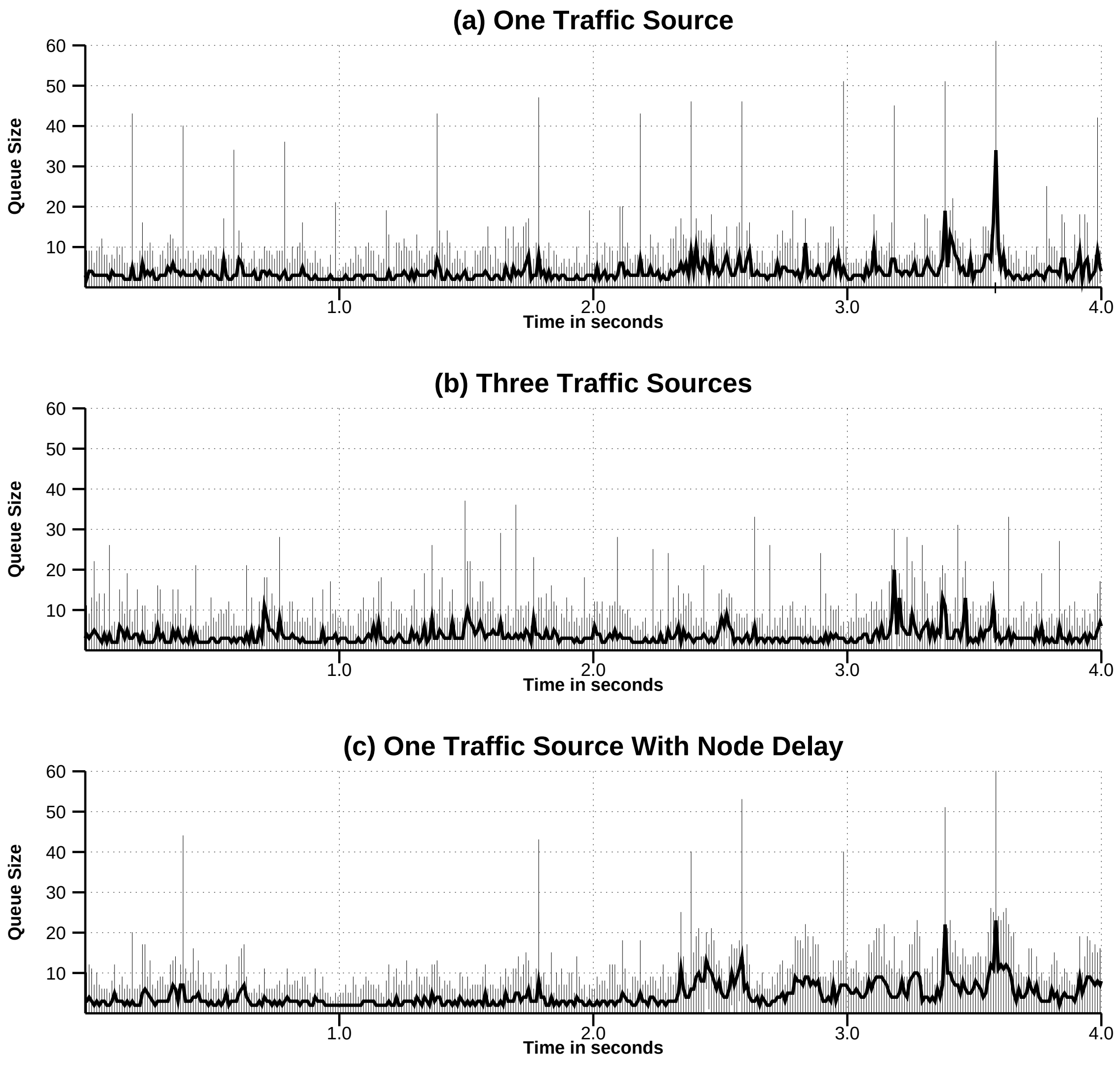


Figure 6.15: Graphs showing artificial periodic effects in simulation.

CHAPTER 7

Conclusion

7.1 Summary

This dissertation has presented the design and implementation of both a new congestion avoidance mechanism for wide area networks and a wide area network simulator. The primary objective of the congestion avoidance mechanism was to increase performance in currently available wide area networks by making better utilization of network resources. The network simulator (x-Sim) was developed to aid in the analysis and testing of this congestion avoidance mechanism. A description of x-Sim was given in Chapter 2. Its main goals were to: (1) be a realistic network simulator, (2) run x-kernel protocols without modification, (3) include good analysis tools.

The reasons for the first and third point are obvious. The reason for the second point, the requirement that x-Sim run x-kernel protocols, is that it allowed us to test the same protocol implementation in both the simulator and the Internet. This eliminates the danger associated with having two implementations of the same protocol, one for the simulator and another for the Internet—the danger that the two implementations would not exhibit the same exact behavior, leading to faulty results. The consequences of this requirement are twofold. First, any protocols already implemented in the x-kernel can be used, saving valuable development time. Second, x-Sim is closer to its goal of being a realistic network simulator as a result of using common protocol implementations.

One major weakness of earlier work with end-to-end congestion avoidance mechanisms is that the mechanisms are never tested in either realistic simulations or real wide area networks. The examination of some of these earlier mechanisms indicates that they fail under realistic network conditions. Since one of the goals of this work was that it result in measurable benefits when used over the Internet, and since

TCP is the most widely used protocol, it was decided that the congestion avoidance mechanism should be introduced into TCP. This was done by modifying TCP Reno, the most common implementation of TCP in use today, without requiring any changes to the TCP specification. As a result, this new implementation of TCP, named TCP Vegas, interoperates with any other valid implementation of TCP—in fact, all the changes are confined to the sending side. TCP and its congestion control mechanisms were described in Chapter 3.

Chapter 4 examined the behavior of TCP Reno's congestion control mechanisms, provided some insights into congestion avoidance, examined previous congestion control mechanisms, and described the congestion avoidance mechanisms in TCP Vegas. The main mechanism consists of two fundamental ideas. First, congestion can be avoided by limiting the number of extra buffers used by each connection in the network. Second, the number of buffers used by a connection can be estimated by comparing its *actual* throughput to an expected *optimal* throughput—the throughput if there were no congestion in the network.

The behavior and performance of TCP Vegas was examined in Chapter 5 through experiments on both real and simulated networks. Experiments on the Internet allowed us to measure the performance of TCP Vegas by comparing the throughput, number of retransmissions and network response times achieved by TCP Vegas and TCP Reno. Although important, performance is not the only issue that needs to be considered when measuring the value of a new network mechanism. Other relevant issues are the mechanism's effects on existing traffic, its fairness, and its stability. These and other issues were explored through simulations with x-Sim.

The value of x-Sim is further demonstrated in Chapter 6 were it is used to analyze TCP Lite, the TCP implementation in the latest release of the Berkeley Software Distribution (BSD) of Unix. This analysis uncovered a number of problems in TCP Lite which seriously affected its performance. The causes of these problems were determined through the simulator's analysis tools. Finally, solutions to these problems are implemented and tested with the simulator. Chapter 6 concludes with a set of guidelines for network simulation.

7.2 Contributions and Limitations

This dissertation provides two major contributions to the area of computer networking. The first contribution consists of TCP Vegas and its congestion avoidance mechanisms. Experiments on real and simulated networks show that TCP Vegas achieves between 37 and 71% higher throughput on the Internet, with one-fifth to one-half the losses, as compared to TCP Reno. Moreover, this improvement in throughput is not achieved by an aggressive strategy that steals bandwidth away from TCP connections that use the current algorithms. Rather, it is achieved by a more efficient use of the available bandwidth. Further experiments with simulated networks also show that TCP Vegas is stable and at least as fair as TCP Reno. Independent experiments by Peter Danzig at USC confirm that TCP Vegas yields higher network throughput and transfers bytes more efficiently than TCP Reno [1].

Although TCP Vegas has been shown to decrease congestion in the network, the fact that it is an end-to-end mechanism imposes certain limitations. For example, its view of the network state is delayed by one round trip time. Therefore it can not always react quickly enough to avoid congestion; congestion and losses may have already happened by the time it gets enough information to detect the early stages of congestion.

Insider mechanisms—those in which routers are actively involved—could be more effective at avoiding congestion than end-to-end mechanisms since they have more information available to them. This does not necessarily imply that they would always be able to prevent losses; that would depend on the particular details of each mechanism.

One of the reasons why insider mechanisms have not been deployed over the Internet is the high cost of modifying or replacing the routers. Another reason is the lack of universal agreement on what is the best insider mechanism, and just as important, whether the current best is good enough to justify the costs involved. Insider mechanisms will likely be used in the future, the only question is how much time their deployment will take.

Whether end-to-end congestion avoidance mechanisms have a place in the future of the Internet will depend on the details of the chosen insider mechanism, or mechanisms.

The second contribution of this dissertation consists of x-Sim and its analysis tools, together with a set of guidelines for doing network simulation. x-Sim was designed to be more realistic than existing network simulators. It achieves this goal through two techniques: by executing real protocol code—from the commonly used implementations of the protocol—and by producing realistic network traffic. The usefulness of x-Sim can be seen throughout this dissertation. It appears first in the work dealing with TCP Vegas, and then in the work analyzing TCP Lite. Through the simulator and its tools we were able not only discover the existence of performance problems, but more importantly, to uncover the causes of these problems and to test solutions to these problems. These solutions resulted in throughput improvements of more than 20%.

The realism of x-Sim comes at a cost: detailed and accurate simulations can take a long time to execute and can use a large number of resources such as memory. Both of this factors limit the complexity and length of network simulations that can be executed in x-Sim.

7.3 Future Directions

There are a number of related areas we plan to explore in the future. For example, we plan to enhance the simulator and investigate congestion avoidance techniques that require new mechanisms in the routers. As mentioned earlier, the realism of the simulator can limit the complexity of the simulations it can run. We plan to solve this problem by having more than one implementation of the major modules of the simulator. Each implementation of a module will simulate the behavior of a particular network component at a different level of realism.

This approach has two advantages. First, it allows us to quantify the effect on the simulation of using a lower realism module by comparing the behavior of the simulator to a version of the simulator that contains the highest realism modules.

This contrasts the current practice of simplifying simulators during their design: as a result, it is hard to quantify the effect the simplifications have on the simulations. The second advantage is that we can increase execution speed of the simulator by using the lowest realism modules which are acceptable for the level of detail we are interested in.

There have been some proposed techniques to avoid network congestion through new router mechanisms. A weakness of these works has been the limited realism of the simulations used to measure the effectiveness of each approach. Analyzing these mechanisms with x-Sim should either uncover weaknesses or increase the confidence level in these mechanisms.

REFERENCES

[1] J.-S. Ahn, P. B. Danzig, Z. Liu, and L. Yan. Experience with TCP Vegas: Emulation and Experiment. In *Proceedings of the SIGCOMM '95 Symposium*, pages 185–195, Aug. 1995.

[2] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. Request for Comments 1633, Sept. 1994.

[3] R. Brown. Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM*, 31(10):1220–1227, Oct. 1988.

[4] D. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.

[5] K. Chung, J. Sang, and V. Rego. A Performance Comparison of Event Calendar Algorithms: an Empirical Approach. *Software—Practice and Experience*, 23(10):1107–1138, Oct. 1993.

[6] J. C. Comfort. The Simulation of a Microprocessor Based Event Set Processor. *14th Annual Simulation Symposium*, pages 17–34, 1981.

[7] P. Danzig and S. Jamin. tcplib: A Library of TCP Internetwork Traffic Characteristics. Technical Report CS-SYS-91-495, Computer Science Department, USC, 1991.

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the SIGCOMM '89 Symposium*, pages 1–12, Sept. 1989.

[9] S. Floyd. TCP and Successive Fast Retransmits. Technical report, Lawrence Berkeley Laboratory, 1994. Available from anonymous ftp from ftp.ee.lbl.gov:papers/fastretrans.ps.

[10] A. Heybey. The network simulator. Technical report, MIT, Sept. 1990.

[11] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[12] N. C. Hutchinson, L. L. Peterson, and S. W. O'Malley. x-Kernel programmer's manual: Version 3.0. Technical Report 89-29, Department of Computer Science, University of Arizona, Nov. 1989.

[13] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of the SIGCOMM '88 Symposium*, pages 314–32, Aug. 1988.

[14] V. Jacobson and R. Braden. TCP Extensions for Long-Delay Paths. Request for Comments 1072, Oct. 1988.

[15] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. Request for Comments 1323, May 1992.

[16] R. Jain. A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. *ACM Computer Communication Review*, 19(5):56–71, Oct. 1989.

[17] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for ExperimentalDesign, Measurement, Simulation and Modeling.* John Wiley and Sons, Inc., New York, 1991.

[18] D. W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300–311, Apr. 1986.

[19] S. Keshav. REAL: A Network Simulator. Technical Report 88/472, Department of Computer Science, UC Berkeley, 1988.

[20] S. Keshav. A Control-Theoretic Approach to Flow Control. In *Proceedings of the SIGCOMM '91 Symposium*, pages 3–15, Sept. 1991.

[21] A. M. Law and W. D. Kelton. *Simulation Modelling and Analysis.* McGraw-Hill, New York, 1990.

[22] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the Self-Similar Nature of Ethernet Traffic. In *Proceedings of the SIGCOMM '93 Conference*, pages 183–193, Oct. 1993.

[23] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. In *Proceedings of the SIGCOMM '94 Conference*, pages 257–268, Aug. 1994.

[24] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach.* Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1996.

[25] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley Publishing Co., New York, 1994.

[26] USC. Transmission control protocol. Request for Comments 793, USC Information Sciences Institute, Marina del Ray, Calif., Sept. 1981.

[27] Z. Wang and J. Crowcroft. A New Congestion Control Scheme: Slow Start and Search (Tri-S). *ACM Computer Communication Review*, 21(1):32–43, Jan. 1991.

[28] Z. Wang and J. Crowcroft. Eliminating Periodic Packet Losses in 4.3-Tahoe BSD TCP Congestion Control Algorithm. *ACM Computer Communication Review*, 22(2):9–16, Apr. 1992.

[29] H. J. Weaver. *Applications of Discrete and Continuous Fourier Analysis.* John Wiley and Sons, Inc., New York, 1983.

[30] L. Zhang, S. Shenker, and D. D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proceedings of the SIGCOMM '91 Symposium*, pages 133–147, Sept. 1991.