MODELING AND EVALUATION OF PROTOCOL BOOSTERS

by

Aravindha Ganesh Ramakrishnan

Thesis submitted to the Faculty of the Graduate School of the University of Maryland, College Park in partial fulfillment of the requirements for the degree of Master of Science
2000

Advisory Committee:

Dr. John S. Baras, Chairman/Advisor
Dr. Raymond Miller
Dr. Leandros Tassiulas

©Copyright by

Aravindha Ganesh Ramakrishnan

2000

ABSTRACT

Title of Thesis: MODELING AND EVALUATION OF PROTOCOL BOOSTERS

Degree Candidate: Aravindha Ganesh Ramakrishnan

Degree and year: Master of Science, 2000

Thesis directed by: Professor John S. Baras

Department of Electrical Engineering

Protocol Boosters are software and/or hardware modules that transparently enhance the performance of existing protocols over heterogeneous networks. In this thesis we have presented a novel design for an ACK Reconstruction Protocol Booster to improve the performance of TCP over asymmetric satellite channels. The asymmetric nature of broadcast satellite channels has been known to have an adverse impact on the performance of TCP. Due to the channel asymmetry, many TCP ACKs may build up in a queue at the ground station router, increasing the round trip time and decreasing the throughput of a given connection. The ACK Reconstruction Protocol Booster has been designed to enhance the performance of TCP over such asymmetric links, transparently, without changing the syntax,

nor affecting the semantics of TCP. Through extensive simulations using OPNET, we have been able to establish that this booster yields

significant improvement in TCP throughput.

Another contribution of this thesis is the modeling of the Forward EraZure Correction (FZC) booster in OPNET and its incorporation within the ARL testbed. The FZC booster is a multi element protocol booster, which reduces the effective packet loss rate on noisy links such as terrestrial and satellite wireless networks. Through simulations, we have evaluated the performance of this booster under varying network conditions.

Other contributions of this thesis include a detailed examination of the protocol booster methodology, together with a critical analysis of the various implementation choices available. Also we have proposed a number of new ideas for future booster implementations

ACKNOWLEDGEMENTS

I am highly indebted to my advisor, Dr. John S. Baras, for introducing me to this wonderful area of research and for his guidance, support and encouragement throughout the course of this work. I would also like to thank Dr. Ray Miller and Dr. Leandros Tassiulas, for kindly consenting to be on the thesis committee. I also wish to thank Bill Marcus of Bellcore, Morristown, NJ for giving me the opportunity to work on the Linux kernel implementation of protocol boosters and for his comments on the design of the ACK reconstruction booster. Finally I wish to thank my Dad – Ramakrishnan Senior, Mom, Sister, Grandma and Aunties for their love and support.

TABLE OF CONTENTS

List of Figures iv

1 Introduction
1

2 Protocol Boosters
5

2.1 Introduction…..……………………………………………………………5
2.2 Motivation……………………………………………………………….....6
2.3 Protocol booster Properties…..…………………………………………...8
2.4 Comparison of boosters with other approaches…..……………………...13
2.5 Protocol Boosters as a solution to current networking problems………..16
2.6 Summary……………………………………………………………….....19

3 Protocol Booster Implementation
20

3.1 Introduction……………………………………………………………….20
3.2 Implementation………..…………………………………………………...20
3.3 Implementation choices and strategy…………………….....……………21
3.4 Kernel-level implementation in Linux…………………….……….....25
3.5 Summary…………………………………………………………………..31

4 The ACK Reconstruction Protocol Booster
32

4.1 Introduction…………………………………………………………….…32
4.2 The Transmission Control Protocol…………………………………….33
4.3 Effects of Asymmetry on TCP performance…………………..………..38
4.4 The ACK reconstruction algorithm…………………….…………....41
4.5 The OPNET model..…………………………...……………..……45
4.6 Results……………………………………………………….…………52

5 The FZC Booster
73

5.1 Introduction……………………………………………………………..…73
5.2 The *modus operandi*……...……………………..…………………...74
5.3 Implementation in the Linux kernel……………………………..…...…75
5.4 The OPNET model……………………………..…………………...76
5.5 Results……………………………………………………….…………84

6 Conclusion **99**

List of Figures

2.1 Trade-off between efficiency and heterogeneity……………………………...7

2.2 Snow Chain Analogy…………………………………………………………9

2.3 Traffic limiting booster………………………………………………….....11

3.1 Insertion of Protocol Boosters in a Layered Protocol………………...………21

3.2 Embedding and selecting boosters in theFreeBSD IP Stack…………………...23

3.3 Kernel Implementation of Protocol Boosters…………………………..……27

4.1 OPNET model of the client node (ground station)…………………...………46

4.2 OPNET model of the server node (satellite)…………………….………..47

4.3 IP process model state transition diagram…...…………………………..….48

4.4 Ground station booster element state transition diagram……………………....50

4.5 State transition diagram of second booster element…………………...……..51

4.6 TCP throughput (number of bits transferred to the application layer) with and without the booster for asymmetry ratio 1:1000………………………..……53

4.7 TCP throughput (number of bits transferred to the application layer) with and without the booster for asymmetry ratio 1:500..…………………...…………54

4.8 TCP throughput (number of bits transferred to the application layer) with and without the booster for asymmetry ratio 1:100..………………...……………55

4.9 TCP throughput (number of bits transferred to the application layer) with and without the booster for asymmetry ratio 1:10………………………………..56

4.10 TCP throughput (number of bits transferred to the application layer) with and without the booster for asymmetry ratio 1:1…..……………………..………57

4.11 TCP throughput (number of bits transferred to the application layer) with and without the booster at a bit rate ratio of 9600:960000……………………..…..59
4.12TCP throughput (number of bits transferred to the application layer) with and without the booster at a bit rate ratio of 56000:5600000……………………....60

4.13TCP throughput (total number of bits forwarded to the application layer) with and without booster for FTP-high load………………………………….62

4.14TCP throughput (total number of bits forwarded to the application layer) with and without booster for FTP-medium load……………………………...63

4.15TCP throughput (total number of bits forwarded to the application layer) with and without booster for FTP-low load…..………………………………64

4.16TCP throughput (total number of bits forwarded to the application layer) with and without booster for rlogin……….……………………………………65

4.17TCP throughput (total number of bits forwarded to the application layer) with and without booster for BER: 1E-7……………………………………...67

4.18TCP throughput (total number of bits forwarded to the application layer) with and without booster for BER: 1E-6……………………………………..68

4.19TCP throughput (total number of bits forwarded to the application layer) with and without booster for BER: 1E-5……………………………………...69

4.20Congestion window growth when there is channel asymmetry……………….71

4.21Congestion window growth when there is no channel asymmetry……………72

5.1 Communications over a wireless network with a FZC booster……………….74

5.2 Radio link transceiver pipeline stages…………………………………………79

5.3 Comparison of normal and boosted network at noise level 75000…………...86

5.4 Comparison of normal and boosted network at noise level 50000…………...87

5.5 Comparison of normal and boosted network at noise level 25000…………...88

5.6 Comparison of normal and boosted network at noise level 9500..…………...89

5.7 Comparison of normal and boosted network at noise level 6000…..………...90

5.8 Comparison of normal and boosted network at noise level 75000 and packet size 16000 bytes……………..…………………………………….92

5.9 Comparison of normal and boosted network at noise level 75000 and packet size 4096 bytes…………….…………………………………………93

5.10 Comparison of normal and boosted network at noise level 75000 and packet size 1024 bytes……..…….……………………………………….94

5.11 Comparison of normal and boosted network at noise level 75000 and packet size 512 bytes…………….………………………………………….95

5.12 Comparison of normal and boosted network at noise level 75000 and packet size 512 bytes with overcode 4%……………….…………………97

5.13 Comparison of normal and boosted network at noise level 75000 and packet size 512 bytes with overcode 30%…………….…………………98

Chapter 1

Introduction

Protocol Boosters [10] are robust protocol adapters designed to dynamically improve (boost) the performance of protocols in heterogeneous distributed computing systems. Protocol boosters allow (1) dynamic protocol customization to heterogeneous environments and (2) rapid protocol evolution.

Unlike alternative adaptation approaches, such as link layer, conversion and termination protocols, protocol boosters are both robust (end to end protocol messages are not modified) and maximize efficiency (does not replicate the functionality of the end to end protocol).

In this thesis we have presented a novel design for an ACK Reconstruction Protocol Booster to improve the performance of TCP over satellite channels. The asymmetric nature of satellite channels has been known to have an adverse impact on the performance of TCP [1,5,9,13]. Due to the channel asymmetry, many TCP ACKs may build up in a queue at the ground station router, increasing the round trip time and decreasing the throughput of a given connection.

Several solutions to this problem have been proposed. Link layer solutions such as ACK filtering attempt to solve the problem by dropping ACKs at the ingress to the bottle neck link. These methods however, suffer from a serious setback in that they ignore the fact that TCP uses ACKs for purposes other than mere acknowledgement of data received.

Protocol termination schemes such as TCP spoofing attempt to enhance performance of TCP by terminating the TCP connection at the gateway – resulting in the loss of valuable end-to-end properties. For instance, receipt of an acknowledgement does not mean that the destination has received the data.

Protocol conversion schemes also suffer from a serious setback in that the message syntax has to be changed- leading to additional processing overhead.

Our ACK Reconstruction Protocol Booster offers a viable solution to this problem. It is a multi element protocol booster, which has been designed to enhance the performance of TCP transparently without changing the syntax or semantics of TCP. Unlike link layer methods, our booster preserves the original ACK stream and hence, does not jeopardize the normal functionality of TCP. Unlike protocol termination, the end-to-end properties of TCP are preserved. Also, since it does not convert the protocol, it does not involve too much processing overhead. It is our claim, which has been substantiated through extensive

simulations using OPNET, that this booster is a definite boon to TCP communication over satellite links yielding significant improvement in TCP throughput.

Another contribution of this thesis is the modeling and evaluation of the Forward Erazure Correction Booster and its incorporation within the ARL simulation testbed. The Forward EraZure Correction (FZC) booster is a multi-element protocol booster, which reduces the effective packet loss rate on noisy links such as terrestrial and satellite wireless networks [16].

While the developers of this booster were able to demonstrate through simple proof-of-concept tests that this booster enhanced performance, there was no reliable model to analyze the performance of this booster within an actual hybrid wireless network. We have attempted to fill this void, by modeling the FZC booster in OPNET and incorporating it within the ARL testbed. Simulation results have proved to be illuminating in that we are now able to determine the degree of usefulness of this booster given the networking environment.

Other contributions of this thesis include a detailed examination of the protocol booster methodology together with a critical analysis of the various implementation choices available. Also we have proposed a number of new ideas for future booster implementations.

The rest of this document is organized as follows. Chapter 2 provides a detailed overview of the protocol booster methodology. Chapter 3 is an in depth analysis of the implementation of boosters and the choices faced by a designer. In chapter 4 we present the design of the ACK

Reconstruction Protocol Booster in detail together with the OPNET model and simulation results. Chapter 5, describes the FZC Booster followed by the OPNET model and simulation results. Chapter 6 concludes this thesis with an overview of the contributions, followed by suggestions for future work.

Chapter 2

PROTOCOL BOOSTERS

2.1 Introduction

Protocol Boosters [10] are robust protocol adapters designed to dynamically improve (boost) the performance of protocols in heterogeneous distributed computing systems. Protocol boosters allow: (1) dynamic protocol customization to heterogeneous environments and (2) rapid protocol evolution.

Unlike alternative adaptation approaches, such as link layer, conversion and termination protocols, protocol boosters are both robust (end to end protocol messages are not modified) and maximize efficiency (does not replicate the functionality of the end to end protocol). In this chapter we present a detailed description of protocol boosters. In section [2.2] we present the motivation for the development of protocol boosters. In section [2.3] we provide a formal definition of protocol boosters, together with several examples. Section [2.4] compares boosters with other protocol architecture alternatives. In section [2.5] we illustrate how protocol boosters can be a solution to current networking problems.

2.2 Motivation

In order to appreciate the concept of boosters let us first examine the current networking protocols and understand the problems they face. Most of the protocols in use today fall into two major categories: General-purpose protocols and specialized protocols.

General purpose protocols are generally complex and designed to meet a wide variety of application requirements and network environments ; however one must note that this often leads to an inefficient use of network resources.

On the other hand, Specialized protocols take full advantage of the application requirements (e.g., loss and latency tolerance) or network environment (e.g., dedicated and high speed links) to leave as much as possible of the communication resources for the application. Unfortunately, developing a robust specialized protocol is expensive and time consuming. Figure 2.1 shows an abstract example of the efficiency-generality trade-off: with IP Internet protocols being general purpose but inefficient and parallel processing protocols being efficient but requiring low error rate, low delay networks. Ideally, a protocol would adapt to provide the best possible performance given the path of the data. For instance, when on a LAN , the protocol would adjust itself to give performance similar to that of a specialized LAN protocol.

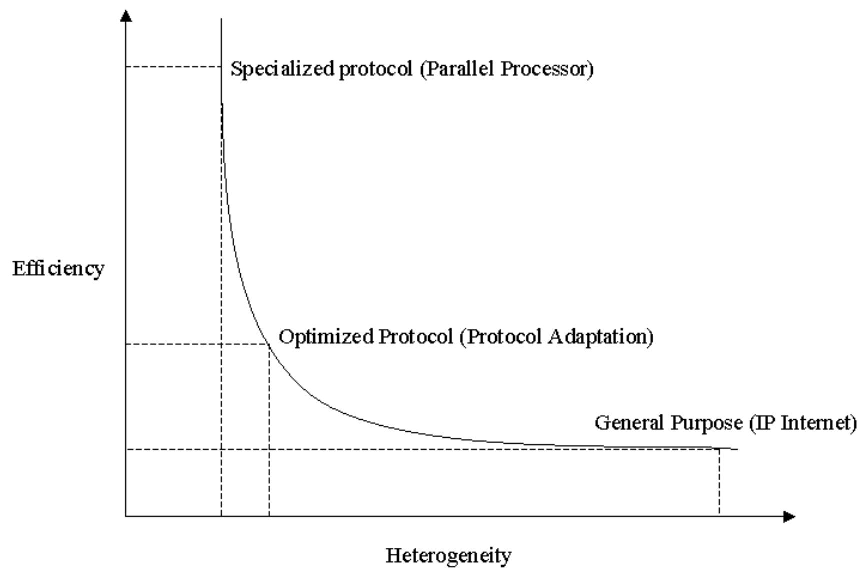


Figure 2.1 Trade-off between efficiency and heterogeneity

Another motivation for protocol boosters is the slow pace at which protocols evolve. The problem is not with the specific protocol or specific standardization method ; but with the need to have many people agree on the standard .

Thus we clearly need to have :

(1) a protocol adaptation technique which can dynamically improve the performance of protocols in heterogeneous systems.

(2) a protocol adaptation technique, which is transparent to the base protocol i.e., the protocol that is being adapted. This transparency would eliminate the need for standardization.

2.3 Protocol Booster Properties

A Protocol Booster is a software or hardware module that transparently improves protocol performance. The booster can reside anywhere in the network or end systems, and may operate independently (one-element booster) or in cooperation with other protocol boosters (multi-element booster). Protocol boosters provide an architectural alternative to existing protocol adaptation techniques, such as protocol conversion and protocol termination.

A protocol booster is a supporting agent that by itself is not a protocol. It may add, delete or delay protocol messages, but never originates, terminates, or converts that protocol. A multi-element protocol booster may define new protocol messages to exchange among themselves, but these protocols are originated and terminated by protocol booster elements and are not visible or meaningful external to the booster.

Protocol boosters are parasitic, in that they use whatever information is available from other protocols or boosters. Protocol boosters are transparent in that they can be added anywhere in the network or end system without altering the boosted protocol.

Snow Chain Analogy

A simple analogy may help explain the parasitic and transparent behavior of boosters. If a protocol is analogous to an automobile tire, then a protocol booster is analogous to snow chains (Figure 2.2). A car with regular tires can add snow chains on snowy roads, just as

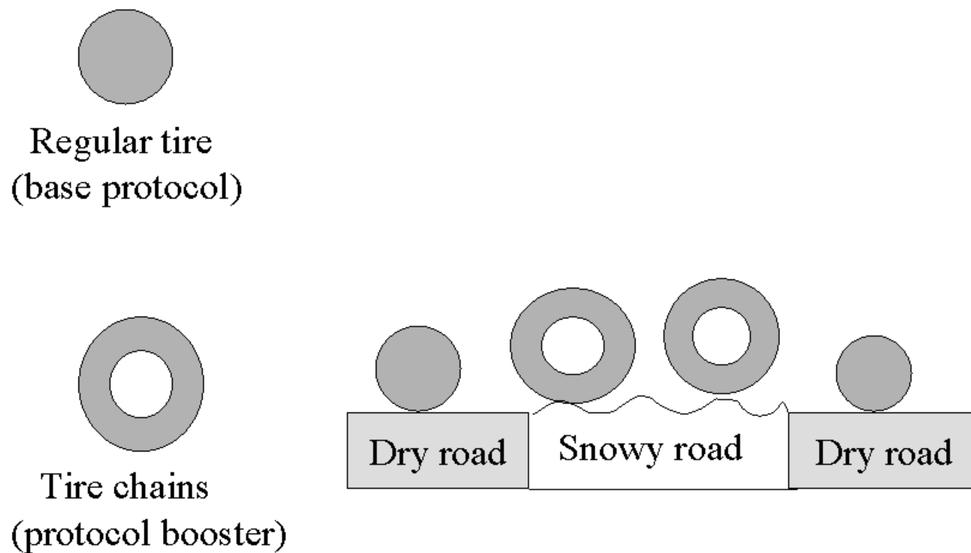


Figure 2.2 Snow Chain Analogy

a protocol designed for a wireline network may use boosters for transmission across a wireless network. The snow chains and boosters are parasitic adapters since a car cannot drive on snow chains alone, just as communication is not possible with protocol boosters alone.

Also, the snow chains and boosters are transparent, since the car tires are not modified by the addition of snow chains, just as the protocol is not terminated or converted by a protocol booster.

Booster Policy

Boosters can be added and deleted dynamically as additional network functionality is needed. A policy for this decision is needed in addition to the specific booster mechanism for adding functionality. Since boosters vary widely in their functions, it is impossible to have a completely general policy; policies must be associated with specific booster functionality. Policies may be based on a wide variety of factors, such as observed network behavior, packet source and destination, or time of day.

One-Element Protocol Booster Examples

(i) One-Element Error detection Booster for UDP

UDP has an optional 16-bit checksum field in the header. If it contains the value zero, it means the checksum was not computed by the source. Computing this checksum may be wasteful on a reliable LAN; however, if errors are possible, the checksum greatly improves data integrity. A transmitter sending data does not compute a checksum for either local or remote destinations. For reliable local communication this saves the checksum computation (at the source and destination). For wide-area communication, the single-element error detection

booster computes the checksum and puts it into the UDP header. The booster could be located either in the source host (below the level of UDP) or in a gateway machine.

(ii) One-Element channel traffic limiting booster for TCP

Often in many systems, it may be necessary to limit the traffic on a channel to the bare minimum. This may be due to limited channel capacity (as on the return channel of satellite systems) or due to the lossy nature of the channel (as in wireless systems).

Consider the system shown in Figure 2.3.

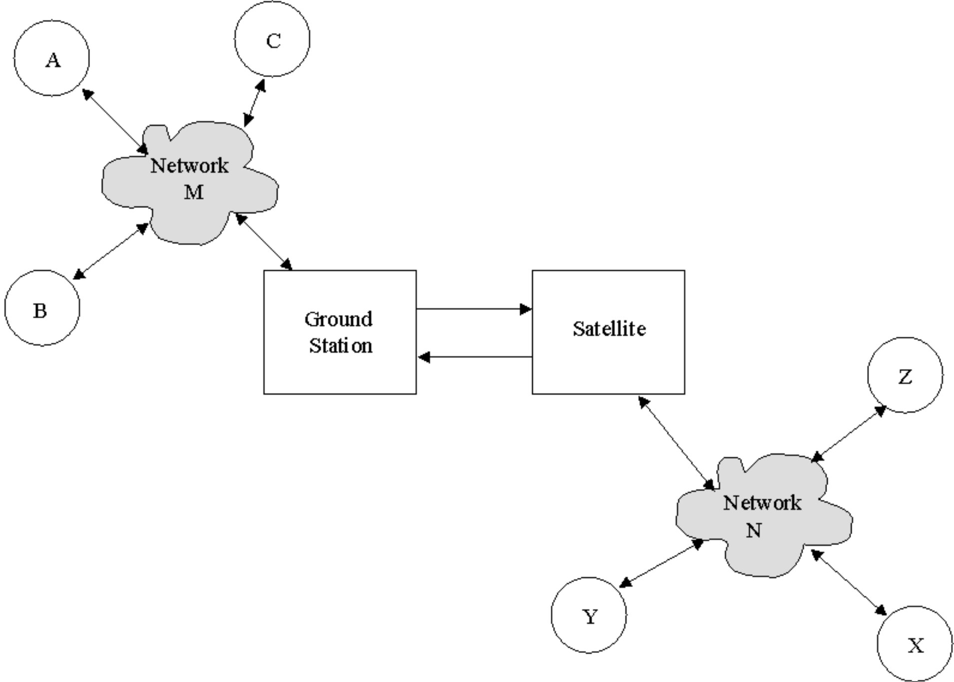


Figure 2.3 Traffic limiting booster

A booster can be installed at the ground station to limit the traffic on the reverse channel. Assuming that three connections A-X, B-Y, C-Z are active, the booster will do the following:

(a) Cache packets from X;

(b) If a packet is lost somewhere in network M and A sends a duplicate ACK, then the booster will drop the duplicate ACK and retransmit the lost packet from its cache.

(c) If an ACK sent by X is lost in network M and A retransmits the packet after timing out, then the booster will drop the packet from A and retransmit the ACK from X.

Two-Element Protocol Booster Examples

(a) Two-Element FZC Booster for IP or TCP

For many real-time and multicast applications, Forward Error Correction coding is desirable[18]. The two-element Forward EraZure Correction booster uses a packet Forward Error Correction code and erasure decoding [2]. The FZC booster at the transmitter side of the network adds parity packets. The FZC Booster at the receiver side removes the parity packets and regenerates missing data packets. The FZC booster can be applied between any two points in a network (including the end systems).

(b) Two-Element Jitter Control Booster for IP

For real time communication, we may be interested in bounding the amount of jitter that occurs in a network . A jitter control booster can be used to reduce jitter at the expense of increased latency. At the first booster element, time-stamps are generated for each data

message that passes. These time-stamps are transmitted to the second booster element, which delays messages and attempts to reproduce the inter-message interval that was measured by the first booster element.

2.4 Comparison of Boosters with Other Approaches

Protocol boosters provide faster evolution and increased efficiency compared to the use of standard general-purpose end-to-end protocols. This section compares protocol boosters with other protocol architecture alternatives, noting that only boosters take advantage of higher layer information (unlike link layer adaptation), while not altering the message syntax (protocol conversion) or semantics (protocol termination).

Link Layer Adaptation

A drawback of independent link layer protocols is that they treat all data streams identically. If there are applications with divergent needs passing over the same link, as there frequently are, then it is unlikely that a single link layer protocol can provide the most efficient service. When data is sent over a noisy link, for example, some data streams, such as those carrying audio, may desire low delay, even if some errors occur. Other data streams may desire a low residual error rate, even at the expense of increased delay. No single link layer protocol could satisfy both requirements. In contrast, a protocol booster could be designed to ignore all UDP connections or boost only certain specific applications, as determined by TCP/UDP port

numbers. Moreover, the booster could be migrated into the end system where it is closer to, and under the control of, the application.

Another drawback of independent link layer protocols is that they can duplicate functionality. If the transport layer protocol provides error control (e.g., TCP), then there is redundant functionality if an independent link layer also provide similar error control. If the transport layer does not provide error control (e.g., UDP), then the end-to-end robustness property is lost. Notice that the protocol booster can treat streams differently and does not duplicate functionality, and thus does not violate the end-to-end argument[21].

Protocol Conversion

The main distinction between conversion protocols[11] and booster protocols is that conversion changes the syntax of the base protocol. Thus, a disadvantage of protocol conversion is that it requires processing to change message syntax. While a protocol booster simply can observe most messages from the base protocol, the protocol converter must modify every message that arrives. Another disadvantage of protocol conversion is that it is not robust to failures. If one end of a protocol conversion fails, then no communication is possible; with boosters, even if one booster element fails, communication is still possible using the base protocol. While protocol boosters and link protocols offer soft degradation, protocol conversion and protocol termination offer hard degradation.

Protocol Termination

Protocol termination[3] uses different protocols at different points along the path from the transmitter to the receiver, with no single end-to-end protocol. Just as with protocol boosters, protocol termination allows choice of protocol characteristics appropriate to the network environments along the communication path and avoids duplication in functionality.

A drawback of protocol termination is that it loses desirable end-to-end properties. For example, if TCP is terminated in the network, then receiving a TCP acknowledgment does not mean that information arrived at the destination. Another drawback of protocol termination is that it provides additional points of failure. Failure of a network termination point causes all messages to be unintelligible (hard degradation) , even if alternative routes are available. If a booster fails, communication is still possible using the base protocol as long as another communication route is available. Performance degrades , but protocol operation continues (soft degradation). Soft degradation is helpful in hostile environments in which failures are expected, such as battlefield situations.

2.5 Protocol Boosters as a Solution to Current Networking Problems

In this section we discus how protocol boosters overcome the slow evolution and reduced efficiency of standard general-purpose protocols.

Fast Evolution of Protocol Boosters

With no need for standardization, we can design and implement protocol boosters with minimal resources. Also, because boosters are transparent, we can replace boosters as often as desired without the knowledge of those using them. Thus a simple, quick booster implementation can be installed quickly. As experience is gained, improved boosters can be created and installed. Because of the fast feedback that can be obtained on booster behavior and performance, boosters can evolve extremely quickly. Also many different boosters can evolve independently in parallel. All of this is because the transparent nature of boosters eliminates the need for standardization.

Protocol boosters are a free market approach to protocol and network design. Booster designs compete economically in the market place, rather than politically in a standards committee. In economics, it is generally believed that companies that compete economically in a free-market system are more efficient than those companies that operate by government fiat. Companies that are successful in the market efficiently produce goods and services desired by consumers. Similarly, the free-market competition among boosters assures that efficient and effective boosters will proliferate and that poor booster designs will become obsolete. Standardization of protocols is expensive because of the need to attend standards meetings,

and only established companies can afford to be involved in standards. Boosters need not be standardized, and they can be quickly designed and built by a small number of people at low cost.

Protocols that require standardization are subject to what economists call "network externality". Network externality is the concept that the value of something depends on the number of people who already use it. Examples of network externality can be seen in VCRs and computer operating systems. VHS tapes are the most available because most people have VHS VCRs. Once one specific example of something that fills a niche becomes dominant, it is difficult to displace it even with a technically superior product. Standardized protocols are subject to network externality. We use TCP/IP because we want to communicate with other people, and most of them use TCP/IP. Network externality dampens competition, because even if a better protocol is designed, it is unlikely to displace existing protocols. Boosters are immune from the feedback caused by network externality because they are transparent and need not be standardized. Consequently, existing boosters can and will be displaced easily and quickly by boosters with better performance.

Another advantage of boosters is that their design and use can be proprietary. With standardized protocols, proprietary market advantage is not possible because you can only communicate with those systems that are running the standardized protocol. Boosters are transparent, and thus, there is no need to disclose the internal design of a booster to those using the booster. The ability to gain proprietary advantage using boosters means that there is

increased market incentive to invest in new booster designs. Care must be taken, however, that the proliferation of protocol boosters does not result in poor performance because of unexpected interactions among proprietary boosters. To reduce this lack of interoperability, successful proprietary booster protocols could eventually become standardized, at which time the developer gives up proprietary claims in return for the wider market for standardized solutions.

Efficiency of Protocol Boosters in Heterogeneous Networks

It is difficult to maintain efficient protocol operation over a wide range of network environments. Thus, as shown in Figure 2.1, protocols generally exchange efficiency for heterogeneity. Reducing the heterogeneity in a networking environment often can increase protocol efficiency. Boosters can be used to increase protocol efficiency without reducing heterogeneity because boosters are a means of hiding the heterogeneity of the networking environment. Instead of optimizing the performance of a protocol over a wide range of network environments, a protocol can be optimized for the network environment between the end host and a booster or between boosters. For example, if network congestion is not an issue on a local network, the booster that adds congestion control can be disabled to reduce unnecessary delay. Thus, boosters allow dynamic customization of a protocol to a heterogeneous environment. It is not necessary to make pessimistic worst-case assumptions about network conditions and application requirements, as is necessary for general-purpose protocols.

The network environment seen on an end-to-end communication path depends on the route taken by the data through the communication network. When boosters are placed in the network, boosters are knowledgeable about the network environment in which they operate. Network boosters also operate only on data that passes through them , so data is only processed at the level necessary for communication locally. Any non-local communication is automatically enhanced by boosters. Thus boosters provide the highest possible performance given the route of the data.

2.6 Summary

This chapter provided a detailed overview of protocol boosters. The motivation for the development of protocol boosters was presented. We then went on to provide a definition for a protocol booster and an illustration of its properties. A few booster examples were presented. We discussed the need for having a booster policy, and finally we finally illustrated how boosters can provide a solution to current networking problems.

In the next chapter we shall explore the intricacies involved in the implementation of protocol boosters.

Chapter 3

Protocol Booster Implementation

3.1 Introduction

In this chapter we shall consider the issues involved in implementing protocol boosters in current operating systems. Section 3.2 presents the general idea of protocol booster implementation. A simplified illustration is provided. Section 3.3 motivates particular design choices reflected in the implementation. In section 3.4 we describe the details of a Unix kernel-level implementation of protocol boosters.

3.2 Implementation

Implementation of boosters requires dynamic insertion of protocol elements into a protocol graph [15]. In practice, protocol graphs are implemented as executable modules that cooperate via messages or shared state. Booster support requires inserting and removing the booster's function from the execution path followed for a group of packets handled by the protocol. A simplified illustration of one style of booster is shown in Figure 3.1.

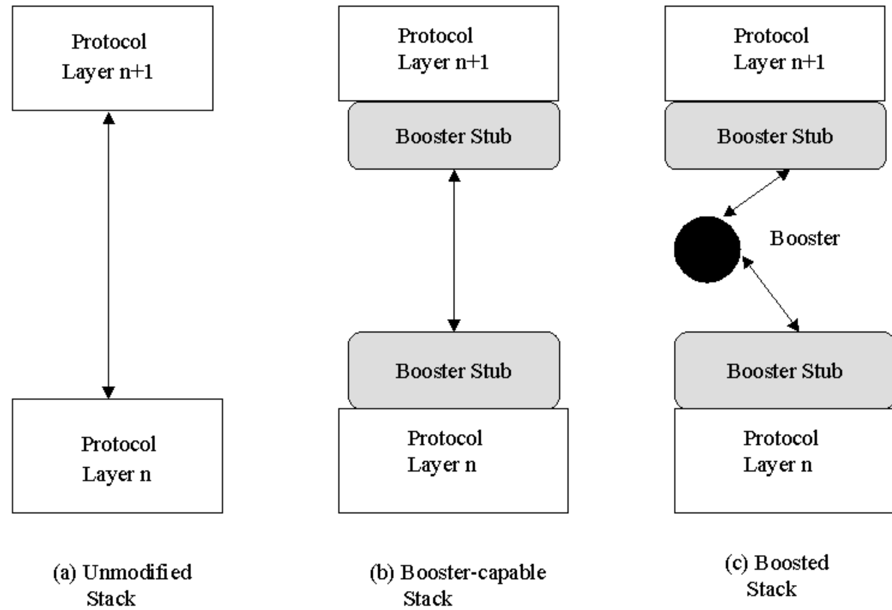


Figure 3.1 Insertion of Protocol Boosters in a Layered Protocol

3.3 Implementation choices and strategy

As Figure 3.1 shows, due to its generality and simplicity, the booster abstraction can be used in many protocol architectures. There is a wide range of implementation alternatives.

Kernel vs. User level

The initial design choice that confronts us is whether to run boosters inside the kernel protection domain, or to operate in user-space. Boosters running in user space are much easier to debug, as well as easier to adapt to other operating systems.

Running boosters as kernel modules can increase performance, because of context-switching and other overheads, as well as availability of control and information about arriving packets. As many boosters commit layer violations, such information can be very important. Since one role of boosters is as performance-enhancers interoperating with existing network protocols, boosters have currently been implemented as kernel modules. Section 3.3 discusses the kernel-level implementation in detail.

Platform choice

Protocol booster support was added to FreeBSD, a free Unix clone for the Intel x86 processor architecture. The placement of this implementation in the IP stack is shown in figure 3.2. Also, as mentioned earlier, kernel-level protocol booster support was added to the Linux operating system for the i386 (Intel) architecture. In the current prototype, the policy decision for boosting has been simplified: all packets destined to (or sourced from) a specific IP address are boosted or de-boosted as necessary.

This choice allows the investigation of OS performance independent of policy research and development. This is accomplished by a demultiplexing algorithm as illustrated in Figure 3.2, which examines the IP address and based on a table lookup, either invokes an appropriate booster or reinserts the packet in the normal execution path. Insertion or deletion of booster functionality is thus controlled by choice of IP address.

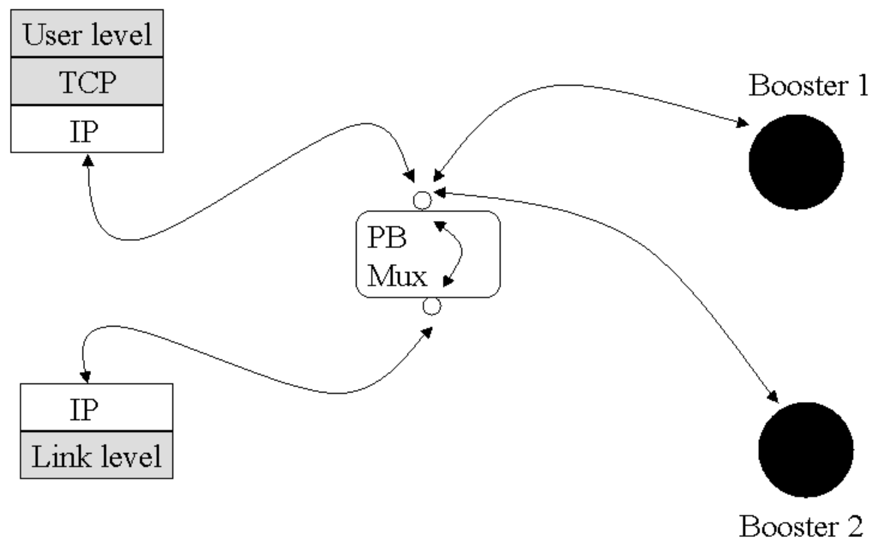


Figure 3.2 Embedding and selecting boosters in theFreeBSD IP Stack

Protocol Layer and implications

The choice of protocol layer is critical to the effectiveness of a protocol booster. At present, protocol boosters have been implemented at the IP layer. However, choice of protocol layer should depend on the particular booster and its functionality. Consider for instance, an ACK Compression protocol booster. On a system with asymmetric channel speeds such as broadcast satellite, the forward (data) channel may be considerably faster than the return (ACK) channel. On such a system, many TCP ACKs may build up in a queue, increasing round trip time and thus reducing the transmission rate for a given TCP window size. The nature of TCP's cumulative ACKs means that any ACK acknowledges at least as many bytes

as any earlier ACK. Consequently, if several ACKs are in a queue, it is necessary to keep only the ACK that has arrived most recently. An ACK compression booster can be designed to ensure that only a single ACK exists in the queue for each TCP connection. Critical to the functioning of this booster is the queuing up ACKs. If there is no queuing up of ACKs at the ground station router, there is no point in invoking the booster at all. Studies have indicated that queuing up of ACKs, occurs at the modem rather than at the IP layer. In view of this fact, it is obviously impractical to implement the ACK Compression protocol booster (or the ACK Reconstruction Booster to be discussed later) at the IP layer. Implementation of these boosters should be done in modem firmware, rather than at the kernel level IP implementation. However, other boosters such as the FZC booster or the Error detection booster are suitable for implementation in the IP layer.

Implementation of protocol booster at the IP layer introduces other limitations. These are related to *packet fragmentation and reassembly* and *multipath routin*g, and are a direct consequence of operating at the IP layer.

Packet fragmentation and reassembly is performed by IP at hosts to avoid the performance cost of repeatedly carrying it out as packets traverse an internetwork. A Maximum Transfer Unit (MTU) is determined for an IP route, which has the property that it requires no fragmentation and reassembly. Where a link has a smaller MTU than the packet size, the packet is fragmented into pieces of MTU size or smaller. The debooster receives the original boosted packet as two (or more) packet fragments. This presents a problem where the

booster functionality requires the entire original packet. Since this requirement is booster-dependent, the prototype OS implementation supplies the MTU of the outgoing interface to the booster so it can act appropriately.

Multipath routing occurs since Internet packets are not guaranteed to be delivered, take a particular route, or arrive in-order. TCP addresses the first and third problems as an IP overlay. This IP behavior can present a problem for boosters, where appropriate deboosters or state necessary to deboost the boosted packet are not present. It also complicates inserting and deleting boosters at necessary locations in an IP internetwork. While routes rarely change, as shown by Claffy[8] in her studies of Internet traffic, such routing dynamics can be addressed by future protocol boosters.

3.4 Kernel-Level Implementation in Linux

Kernel -level protocol booster support was added to the Linux operating system for the i386(Intel) architecture PC[17]. Kernel implementation offers *efficiency* and *transparency.*

Individual protocol boosters are implemented in the form of Linux loadable kernel modules(lkm). This allows each protocol booster to be inserted/removed *on-the-flight run time.*

Because booster functionality can be instantiated via multiple processes, uniquely naming all the system components is required. The system components consist of *booster loadable kernel modules*, *booster instances, booster sequences* and *booster traps.*

A booster sequence is a concatenated chain of booster modules. One can view the booster modules as definitions, and their association with a given booster sequence as a declaration, or booster instance. Each instance of a booster can behave differently based on the arguments that are passed to the booster when it is invoked or arguments passed via ioctl system calls. Arguments are encoded as character strings because Linux does not implement a general *scanf* routine within the kernel. Finally a mechanism to direct or *trap* packets to a booster sequence was incorporated using a modified version of the existing Linux firewalling filter.

Booster Architecture

Boosters must be placed strategically within the operating system to interoperate and work transparently with the IP protocol stack. Figure 3.3 illustrates the components of the booster architecture as they relate to the IP protocol stack.

The boosters themselves are positioned at a relatively low level within the IP stack so that they can operate on packets during the input, output or forwarding functions. The Boost interface allows for out-of-band, inter-booster module communications. The figure also shows

the policy manager which manages the boosters and the user level booster daemon. The raw packet interface (User D) is not forced through the booster interface.

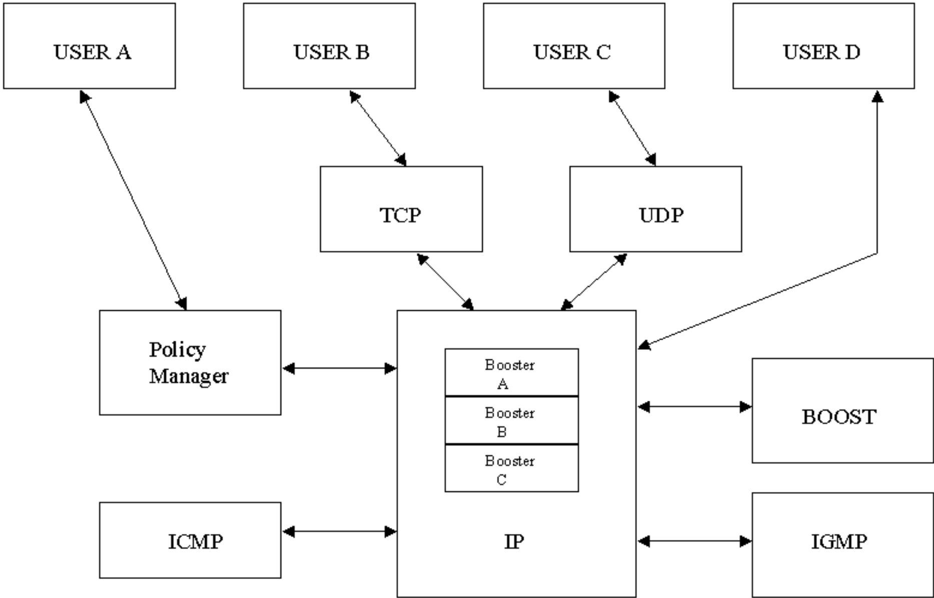


Figure 3.3 Kernel Implementation of Protocol Boosters

This was done to insure that there was one mechanism to transmit and receive packets that was unaffected by the presence of boosters.

Packets are fed to each booster in sequence in the order that they are assembled for output and in the reverse order on input. Boosters can influence the packets that are passed to them. The control mechanism that sequences boosters examines the return value from each booster. A booster can indicate that one of several actions can be taken:

- Retain a copy of the packet and pass the packet to the next booster.

- Do not retain a copy of the packet and pass the packet to the next booster.
- Break the sequence and discard the packet.

The need to retain packets does not stall either transmission or reception since packet cloning (packet use counts) is used to eliminate unnecessary copies in physical memory and to prevent packets from being released until the appropriate time. As part of this discipline, boosters must obey one cardinal rule. They must not force the transmission of the packet passed to them, i.e., the packet will be transmitted after all of the boosters in the sequence have seen the packet. Boosters can generate additional packets but frequently it is important that the packets generated follow the packet that the booster is currently examining . Any new packets can be queued to a special queue associated with the booster sequence so that they are transmitted *after* regular packets are processed by the booster sequence. This can be important when the transmission of a booster-generated packet before an old packet would cause unnecessary processing at a downstream node. There is, however, nothing to prevent a booster from transmitting a new packet immediately.

Structure of a Booster Module

There are six main interfaces that a booster loadable module must implement. The first of these is called the instance manager interface used to create and release any private data for the booster and maintain appropriate use counts. The count mechanism that keeps device drivers loaded is subverted so that each *instance* of a booster increments the use count to prevent

modules from being unloaded prematurely while a sequence is still active. The *output interface* examines IP packets being output by the system, the *input interface* examines packets as they are input to a system while the *forwarding interface* examines packets as they are output when forwarding packets. Many boosters implement the forwarding interface as a wrapper around the booster output interface. The *Boost interface* is devoted to handling the reception of booster protocol packets, which are used for transparent, inter-booster module communications. Finally an *ioctl interface* is provided to allow user processes to modify parameters associated with the instance manager interface.

The Kernel Interface

The booster management routines are delegated to the resident kernel for efficiency and during the software interrupts used to process network traffic. These routines are exposed for loadable modules to use via the standard Linux module address resolution mechanism. A substantial part of the interface that manages the construction of booster sequences resides in the *policy manager* which is itself a loadable module. The policy manager is also a character device driver (/dev/policy) and exposes its interfaces via a series of ioctl calls. This interface combines elements of the IP firewalling interface to expose the needed functions to construct boosters and traps. A Java class is provided to allow for a pleasant programming interface and to hide the implementation details.

Knowledge about the resources used by boosters is found in the */proc* file system. The directory */proc/net/boosters* contains all of the relevant information about protocol boosters. The policy manager exposes the *booster types* file to show what boosters have been loaded into the kernel. Similarly, a *nettraps* file exposes the traps that have been set and which booster sequences they are connected to.

Finally, each booster has a standard entry, which provides information on each instance of that booster, however, each booster can expose as many files as necessary to document the attributes of the booster.

The Policy Manager

The name policy manager is actually a misnomer since it does not implement any policy of its own. It provides services to user level programs to assemble boosters, create traps, etc. Via a series of ioctl calls similar to the interfaces that most UNIX systems use to provide the interface to the ARP and route subsystems.

The policy manager was implemented as a loadable kernel module because it was envisioned that a series of loadable kernel level policy managers could be built which automatically load and unload boosters based on an internal policy that they implemented or were configured to use via a language or a via messages that they received.

3.5 Summary

This chapter discussed how protocol boosters can be implemented in current operating system environments. A generic illustration of the implementation was followed by a detailed discussion of the design choices that have to be made in implementing boosters. The issues involved in kernel and user level implementation; platform choice and the protocol layer of implementation were discussed. We finally described in detail, the kernel level booster implementation in Linux.

Chapter 4

The ACK Reconstruction Protocol Booster

4.1 Introduction

The asymmetric nature of satellite channels has been known to have an adverse impact on the performance of TCP[1,5,9,13]. Due to the channel asymmetry, many TCP ACKs may build up in a queue at the ground station router, increasing the round trip time and decreasing the throughput of a given connection.

This chapter presents the design of an ACK Reconstruction Protocol Booster to enhance the performance of TCP over satellite channels. Section 4.2 provides a brief overview of TCP. In section 4.3 we illustrate how channel asymmetry affects the performance of TCP and in section 4.4 we present the ACK Reconstruction Algorithm. Section 4.5 discusses the OPNET model and in section 4.6 we discuss the simulation results.

4.2 The Transmission Control Protocol (TCP)

While the original formal specification of TCP is in RFC 793 [20], numerous variants of it have been developed over the past several years, such as Tahoe, Reno, Vegas, etc. [22,6]. This section discusses the Reno variant of TCP, which is the predominantly deployed version today.

Cumulative Acknowledgments

TCP is an ARQ- based reliable transport protocol that uses cumulative ACKs and byte-based sequence numbers for reliability. TCP provides a *fully reliable, in-order, byte-stream* delivery abstraction to the higher-layer application, which typically uses a socket interface [23] to interface with the transport layer. The basic unit of transmission is called a *segment*, which is a contiguous sequence of bytes identified by its 32-bit long start and end sequence numbers. The transmitted segments are smaller than or equal to the connection's maximum segment size (MSS), which is negotiated at the start of the connection.

A cumulative ACK from a receiver for byte k implies that all bytes less than k have been successfully received, and that a segment beginning at byte k has not yet arrived. During normal loss-free operation, the cumulative ACK sequence steadily increases as segments arrive in sequence. In what follows, we describe how TCP performs loss recovery, congestion control, and connection management, and highlight weaknesses in its congestion control scheme.

Loss Recovery

When the TCP sender discovers that data has been lost in the network, it recovers from it by retransmitting the missing data segments. TCP has two mechanisms for discovering and recovering from losses: *timer-driven retransmissions* and *data-driven retransmissions.*

Timer-driven recovery: When the TCP sender does not receive a positive cumulative ACK for a segment within a certain timeout interval, it retransmits the missing data. To determine the timeout interval, it maintains a running estimate of the connection's round-trip time using an exponential weighted moving average (EWMA) formula, $srtt = a * rtt + (1-a) * srtt$, where $srtt$ is the smoothed round-trip time average, rtt the current round-trip sample, and a the EWMA constant set to 0.125 in the TCP specification. It also estimates the mean linear deviation, $rttvar$, using a similar EWMA filter, with a set to 0.25. A timeout occurs if the sender does not receive an ACK for a segment within $srtt + 4*rttvar$ since the arrival of the last new cumulative ACK. Furthermore, the retransmission timer is exponentially backed off after each unsuccessful retransmission. The details of the round-trip time calculations and timer management can be found in [12,22].

Data-driven recovery

TCP's data-driven retransmission mechanism uses a technique called Fast Retransmission. It relies on the information conveyed by cumulative ACKs and takes advantage of the receipt of later data segments after a lost one. Because ACKs are cumulative, all segments after a missing one generate duplicate cumulative ACKs that are sent to the TCP sender. The sender uses these duplicate ACKs to deduce that a segment is missing and retransmits it.

However, the sender must not retransmit a segment upon the arrival of the very first duplicate ACK. This is because the Internet service model does not preclude the reordering of packets in the network; such reordering causes later segments to be received ahead of earlier ones, and triggers duplicate ACKs in the same way that losses do. Furthermore, the degree of packet reordering on the Internet seems to be increasing, according to statistics by Paxson [19]. Thus, to avoid prematurely retransmitting segments, the sender waits for three duplicate ACKs, the current standard fast retransmit threshold.

This is followed by the *fast recovery* phase, where additional packets are transmitted after the sender is sure that at least half the current window has reached the receiver, based on a count of the number of received duplicate ACKs. Fast recovery ensures that a fast retransmission is followed by congestion avoidance and not by slow start. Since the arrival of duplicate ACKs signals to the sender that data is indeed flowing between the two ends, there is no reason to suddenly throttle the sender by invoking slow start.

Fast retransmissions are often not sufficient to recover from multiple losses in a single window, because all the cumulative ACKs arrive for the first loss in the window. This usually results in a coarse-grained timeout before the packet is retransmitted.

TCP currently makes the tacit assumption that all losses occur because of network congestion. Thus, coupled with either of these modes of retransmission is congestion control that reduces

the sender's retransmission rate. The next section discusses how TCP manages congestion by probing for sustainable bandwidth and reacting to congestion.

Congestion Avoidance and Control

TCP's congestion management is based largely on Jacobson's seminal paper[12] and on the principles of multiplicative-decrease/additive-increase expounded by Chiu and Jain[7]. TCP uses a window-based algorithm to manage congestion, where the window is an estimate of the number of bytes currently unacknowledged and outstanding in the network.

The TCP sender performs *flow control* by ensuring that the transmission window does not exceed the receiver's advertised window size. It performs congestion control by using a window-based scheme, where the sender regulates the amount of transmitted data using a congestion window. When a connection starts or resumes after an idle period of time, *slow start* is performed. Here, the congestion window is initialized to one segment and every new ACK increases the window by one MSS. After a certain threshold (called the slow start threshold, *ssthresh*) is reached, the connection moves into the *congestion avoidance* phase, in which the congestion window effectively increases by one segment for each successfully transmitted window. In response to a packet loss, the sender halves its congestion window; if a timeout occurs, the congestion window is set to one segment and the connection goes through slow start once again.

In addition, *ssthresh* is set to half the value of *cwnd* at the time of loss, and at each stage slow start is performed until *cwnd* reaches *ssthresh.* At any point in time, the TCP sender ensures that the number of unacknowledged bytes is not larger than the smaller of the receiver's advertised flow control window and *cwnd.* In the steady state of the connection, the sender transmits a window's worth of data every round-trip, at a long-term rate equal to the bottleneck bandwidth. Since the ratio of window to round-trip delay equals the bottleneck bandwidth, TCP's congestion window size tends towards the connection's bandwidth-delay product. This is the number of outstanding bytes in transit in the steady state. Thus it is clear that a low-bandwidth connection implies small transmission windows. In turn, this affects the fast retransmission mechanism because not enough duplicate ACKs arrive, leading to expensive timeouts.

Data transmissions are triggered and clocked by ACKs that arrive for previous segments; every time an ACK arrives, the transmission window shifts by an amount equal to the sum of the number of bytes acknowledged and the increase in *cwnd,* subject to the constraint imposed by the flow control window. This elegant notion of transmitting data based on incoming ACKs, called *ACK-clocking,* makes TCP a tightly-coupled feedback system and frees the sender from maintaining explicit software timers for transmission. However, it also leads to significant performance degradation when ACK feedback is imperfect or variable.

Connection Management

TCP has a sophisticated connection initiation mechanism using a three-way handshake, where a SYN exchange occurs before the connection is established and data can flow. A connection

is uniquely identified by the source and destination IP address and port numbers. Details of the connection setup, teardown and state transitions are orthogonal to our work(see [20,22] for details).

4.3 Effects of Asymmetry on TCP Performance

A network is said to exhibit asymmetry with respect to TCP performance, if the throughput achieved is not solely a function of the link and traffic characteristics of the forward direction, but also depends significantly on those of the reverse direction[4]. Here, forward, refers to the direction from the sender to the receiver and reverse, the opposite.

Fundamentally, network asymmetry affects the performance of reliable transport protocols such as TCP because these protocols rely on feedback in the form of ACKs from the receiver to ensure reliability and smooth transmissions. Because TCP transmissions are ACK-clocked, any disruption in the feedback process can impair the performance of the forward data transfer. For example, a low bandwidth ACK path could significantly impede the growth of the TCP sender window during slow start, independent of the link bandwidth in the direction of data transfer.

Depending on the characteristics of the reverse path, two types of situations arise for unidirectional traffic over asymmetric-bandwidth networks: when the reverse bottleneck link

has sufficient queuing to prevent packet (ACK) losses, and when the reverse bottleneck link has a small buffer. We consider each situation in turn.

Deep Reverse Queues

If the reverse bottleneck link has deep queues so that ACKs do not get dropped on the reverse path, then performance is a strong function of the normalized bandwidth ratio, k, defined by Lakshman and Madhow [14]. k is the ratio of the raw bandwidths divided by the ratio of the packet sizes used in the two directions. For example, for a 10 Mbps forward channel and a 50 Kbps reverse channel, the raw bandwidth ratio is 200. With 1000-byte data packets and 40-byte ACKs, the ratio of the packet sizes is 25. This implies that $k = 200/25 = 8$. Thus, if the receiver acknowledges more frequently than one ACK every $k = 8$ data packets, the reverse bottleneck link will get saturated before the forward bottleneck link does, limiting the throughput in the forward direction.

If $k > 1$ and ACKs are not delayed or dropped, TCP ACK-clocking breaks down. Consider two data packets transmitted by the sender in quick succession. En route to the receiver, these packets get spaced apart according to the bottleneck link bandwidth in the forward direction. The principle of ACK clocking is that the ACKs generated in response to these packets preserve temporal spacing. However, the limited reverse bandwidth and queuing at the reverse bottleneck router alters the inter-ACK spacing observed at the sender. When ACKs arrive at the bottleneck link in the reverse direction at a faster rate than the link can support, they get queued behind one another. The spacing between them when they emerge from the

link is dilated with respect to their original spacing, and is a function of the reverse bottleneck bandwidth. Thus the sender clocks out new data at a slower rate than if there had been no queuing of ACKs. No longer is the performance of the connection dependent on the forward bottleneck link alone; instead, it is throttled by the rate of arriving ACKs. As a side-effect, the sender's rate of congestion window growth slows down too.

Shallow Reverse Queues

The second situation arises when the reverse bottleneck link has a relatively small amount of buffer space to accommodate ACKs. As the transmission window grows, this queue fills and ACKs are dropped. If the receiver acknowledges every packet, only one of every k ACKs get through to the sender, and the remaining $(k-1)$ are dropped due to buffer overflow at the reverse channel buffer (here k is the normalized bandwidth ratio). In this case, the reverse bottleneck link capacity and slow arrival are not directly responsible for any degraded performance. However, there are three important reasons for degraded performance in this case because ACKs are infrequent:

1. First, the sender transmits data in large bursts. If the sender receives only one ACK in k, it transmits in bursts of k (or more) segments because each ACK shifts the sliding window by at least k (acknowledged) segments. This increases the likelihood of data loss along the forward path especially when k is large, because routers do not handle large bursts of packets well.

2. Second, TCP sender implementations increase their congestion window by counting the number of ACKs they receive and not on how much data is actually acknowledged by each ACK. Thus fewer ACKs implies a slower rate of growth of the congestion window, which degrades performance over long-delay connections.

3. Third, the sender's fast retransmission and recovery algorithms are less effective when ACKs are lost. The sender may not receive the threshold number of duplicate ACKs even if the receiver transmits more than the required number. Furthermore, the sender may not receive enough duplicate ACKs to adequately inflate its window during fast recovery.

In summary, bandwidth asymmetric networks suffer from degraded performance due to slow or imperfect ACK feedback.

4.4 The ACK reconstruction algorithm

The cumulative nature of TCP ACKs imply that each ACK acknowledges at least as many bytes as the previous ACK. So if a number of ACKs are queued up at the ground station router, it makes sense, just to retain the latest ACK and discard the rest.

However, before we design an algorithm to send just the most recent ACK along the link, we must remember from our discussion in the previous sections, that TCP uses its ACKs for purposes other than the mere acknowledgment of data received. TCP assumes that a packet

is lost in the network if it receives three or more duplicate ACKs. TCP then enters what is known as a "fast retransmit", wherein it retransmits the missing segment even before the retransmit timer goes off.

Also, TCP 's flow control methodology (slow start and congestion avoidance) relies on the arrival of the ACKs to increase the window size. Our algorithm should ensure that it does not jeopardize the normal functioning of TCP.

Another point to be noted is that TCP estimates its RTT based on the timely arrival of ACKs. So we have to make sure that by dropping/delaying ACKs we do not affect these RTT estimates.

The ACK reconstruction booster has been designed after a thorough consideration of these various factors. The ACK reconstruction booster is a two element protocol booster, which uses the cumulative nature of TCP acknowledgments to reduce the number of ACKs flowing on the bandwidth constrained ground station- satellite link. While this discussion is with respect to satellite environment, this booster can be used in other environments such as in ADSL where a channel asymmetry is present.

The two booster elements are located at the two respective ends of the asymmetric channel. For the purposes of our discussion we will refer to the ground station end as the "sender" (of ACKs) and the satellite end as the "receiver".

The booster at the sender does the following.

- Whenever an ACK is about to be queued for transmission, the booster notes the number of packets already in the queue. If this number exceeds a certain threshold, then the booster is invoked. The booster notes the information contained in this ACK namely the ACK sequence number, window sequence number, the source IP address, the destination IP address, the source port and the destination port. A special packet called a “Boost” packet is constructed (possibly using Raw IP) and the above information is written onto it.
- Then the booster traverses the queue from head to tail. All plain ACKs (ACKs that contain no data) are dropped. A record is maintained about the number of ACKs that are dropped. This is again written onto the Boost packet. When the booster reaches a piggy-backed ACK (an ACK along with user data) , a slightly different approach is employed. Since we don’t want to lose the user data, we overwrite the ACK number and window sequence number with that of the latest ACK , recompute the checksum and allow the piggy-backed ACK to remain in the queue.
- The booster maintains a record of the rate at which the ACKs arrive at the ground station router. This information is again written onto the Boost packet.
- The Boost packet is transmitted.

However there are a few special cases which must be taken care of.

(1) We stated that the booster is invoked whenever an ACK is about to be queued for transmission. What happens if the ACK to be queued is a piggy-backed ACK? If we incorporate the data from the piggy-backed ACK into the Boost packet and transmit it, this may result in unnecessary reordering if any other piggy-backed ACKs are already present in the queue. So we have to impose the further condition (policy) that the booster be invoked, only for plain ACKs (ACKs that contain no user data).

(2) We must ensure fairness among the connections. We said that the booster on constructing the Boost packet immediately transmits it. If we implement this scheme as is, this could lead to unfairness. Consider the following scenario. Let us assume there are packets belonging to two different connections- say connection A and connection B are queued up for transmission. A packet belonging to connection B is currently at the head of the queue and is about to be transmitted. At this point of time, a plain ACK belonging to connection A arrives to be queued for transmission. The booster is promptly invoked, does all the processing, and constructs the Boost packet. Now, if we immediately transmit this Boost packet (for connection A), then we are unfair to connection B, because the next packet that was in line to be transmitted was that of connection B. In order to prevent the booster from causing such unfairness we have to add an additional detail to the booster implementation. Whenever the Boost packet has been constructed, it should not be immediately transmitted. Instead it should be queued up at the position of the first packet (from the head of the queue) belonging to that particular connection. Implementing this should be easy. When the booster traverses the queue from head to tail it can mark the

position of the first plain ACK by allowing it to remain in the queue and not dropping it. Later when the entire queue has been traversed, it can swap the plain ACK with the Boost packet.

(3) Implementing the above may lead to a situation where, the booster while traversing the queue encounters another Boost packet. In such a situation the data from the earlier Boost packet is also used in the construction of the new Boost packet. The old one is dropped.

At the receiving end (of ACKs) the booster, on receiving a Boost packet, uses the information contained in the Boost packet to reconstruct the ACK stream. It generates ACKs with increasing sequence numbers from the last ACK number to the ACK number contained in the Boost packet. The number of ACKs regenerated is equal to the number of ACKs that were dropped. Also, temporal spacing between the ACKs is maintained by regenerating ACKs at intervals of time equal to the average arrival rate of ACKs at the ground station router. These reconstructed ACKs are then passed on to TCP.

4.5 The OPNET Model

The ACK reconstruction booster has been implemented in OPNET. The objective was to develop a simple model, which would illustrate the concept and which would enable us to study the performance benefits and analyze how it would vary with different asymmetry ratios, different bit error rates, and the nature of the traffic flowing across the link. In the actual implementation, this booster would have to be implemented in the modem firmware since

queuing of ACKs takes place only at the modem. The OPNET model of this booster, however is implemented at the IP layer. Figure 4.1 shows the OPNET node model for the client (ground station) and Figure 4.2 the model for the server (satellite).

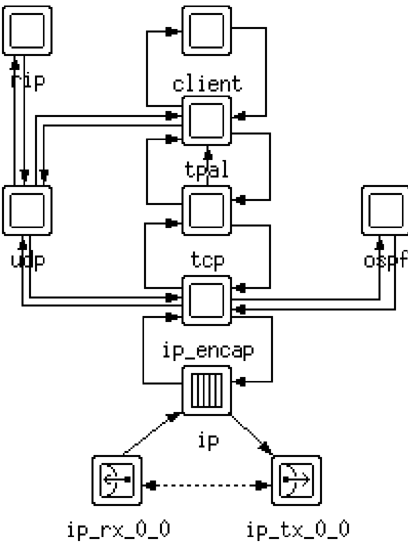


Figure 4.1 OPNET model for the client (ground station)

The client and server are connected via two point to point links – one from the client to the server and the other from the server to client. We shall refer to the link from the server to the client as the forward direction and the other as the reverse link. This model allows us to independently set the bit rates for the forward and reverse links.

This client – server model allows us to model typical real life applications such as ftp, rlogin besides providing an interface to model customized applications.

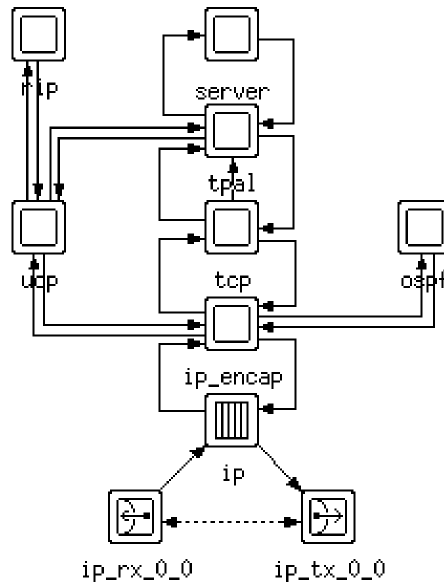


Figure 4.2 OPNET model of the server (satellite)

The ACK Reconstruction Protocol Booster has been modeled within the IP module of the client and in the TCP module of the server. Figure 4.3 illustrates the IP process model that was used to model the booster.

The *init* state performs all necessary initializations. An IP address is assigned for each interface. In the *wait* state, the process waits for all other processes to register their IP addresses in the global IP address table. The *init_too* state is used to broadcast a special IP datagram on all network interfaces to allow IP nodes to discover their neighbors and construct their interface tables. The *arrival* state handles the enqueing of arriving datagrams.

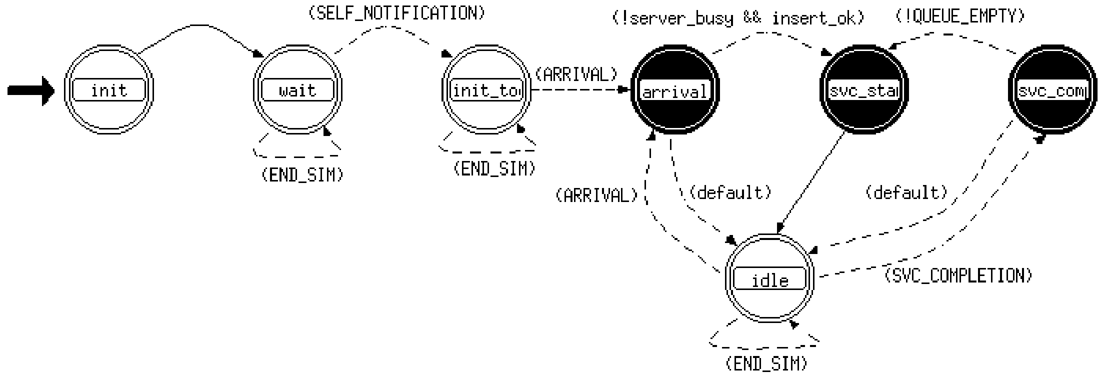


Figure 4.3 IP process model state transition diagram

In the *idle* state, the process waits either for a datagram arrival or a service completion. The *svc_start* state schedules a self interrupt to expire at the end of the service time. This indicates that the process is now busy servicing a datagram. The *svc_compl* state is used to route a datagram based on the destination IP address.

The SELF_NOTIFICATION transition occurs when the IP addresses have been assigned and registered in the global IP Address Table. The ARRIVAL transition occurs when an IP

datagram arrives either from the higher or lowere layer. The SVC_COMPLETION transition occurs when a datagram has completed service and is ready to be forwarded on the appropriate output interface. The *transition !server_busy && insert_ok* occurs if the process is not busy servicing another datagram, and the arrived datagram was successfully inserted into the queue. The !QUEUE_EMPTY transition occurs when the queue contains a datagram that needs to be serviced. The END_SIM runs final reports for link utilization if link utilization reporting is enabled.

OPNET allows us to configure the IP forwarding rate of nodes. This is the rate (measured in number of packets per second) at which the IP layer forwards the packets from its internal queue. In the OPNET model, we force the ACKs to queue up at the IP layer by setting the IP forwarding rate of the ground station to be a fraction of the IP forwarding rate of the satellite.

The ACK reconstruction booster is a two element protocol booster. The first element, resides at the ground station (client). It deals with the construction of the Boost packet and is implemented as a child process from the SVC_COMPL stage of the OPNET IP process model described above. The state transition diagram for the booster process at the ground station is shown in figure 4.4.

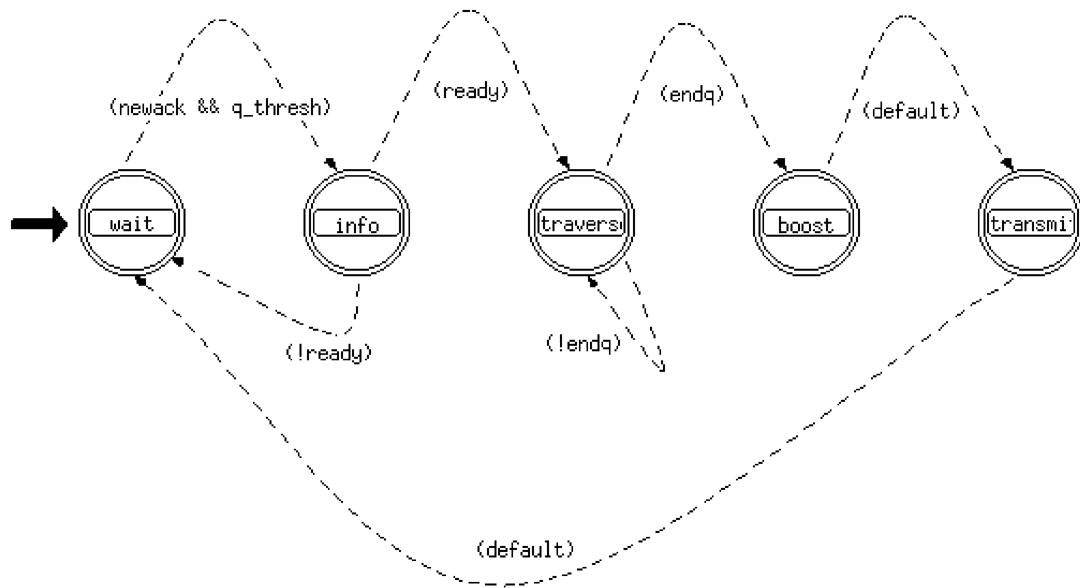


Figure 4.4 Ground station booster element state transition diagram

The *wait* state is an idle state, where the process awaits the arrival of a new acknowledgement. On the arrival of a new acknowledgement, if the length of the queue is greater than the threshold (*newack && q_thresh*), the process enters the *info* state. The *info* state collects the connection information and the latest acknowledgement and window sequence number. On passing an integrity check (*ready*) the process enters the *traverse* state. While the end of the queue is not reached (!*endq*) the process examines successive packets in the queue and performs the operations described in the algorithm. On reaching the end of the queue (*endq*), the process enters the *boost* state, wherein the Boost packet is constructed using the information collected. Upon completion of the construction of the Boost

packet, the process enters the *transmit* state, where the Boost packet is queued for transmission. The process then transits back to the *wait* state.

In order to simplify the OPNET model, we make the assumption that only plain non-duplicate ACKs are queued up at the ground station.

At the other end, the second booster element reconstructs the ACK stream using the information contained in the booster packet. Figure 4.5 illustrates the state transition diagram of the second booster element.

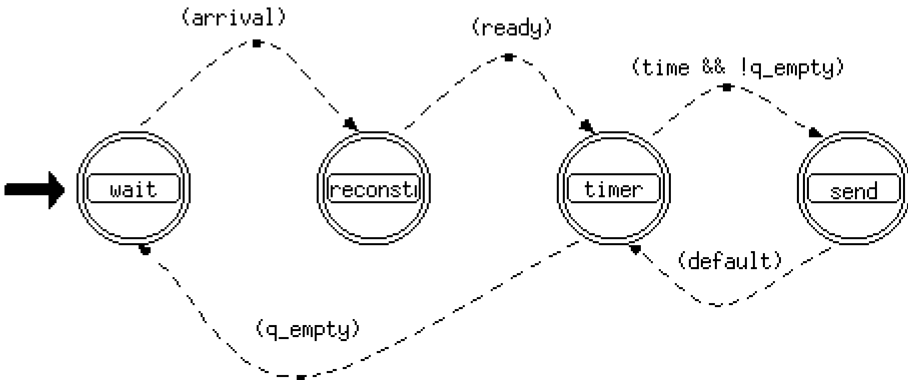


Figure 4.5 State transition diagram of second booster element

The second booster element resides at the TCP module of the OPNET model. In the *wait* state, the process awaits the arrival of the Boost packet. Upon *arrival* of a Boost packet, the

process transits to the *reconstruction* state where the ACK stream is regenerated based on the count of ACKs contained in the Boost packet. The process then transits to the timer state. Here based on the ACK inter arrival information, a timer is set to indicate the time of transmission of the next ACK. At that *time* the process transits to the *send* state, which delivers the ACK to the TCP module. This continues until the entire ACK stream has been regenerated *(!q_empty)* and sent to TCP. When the regeneration queue becomes empty (*q_empty*) the process transits back to the *wait* state.

4.6 Results

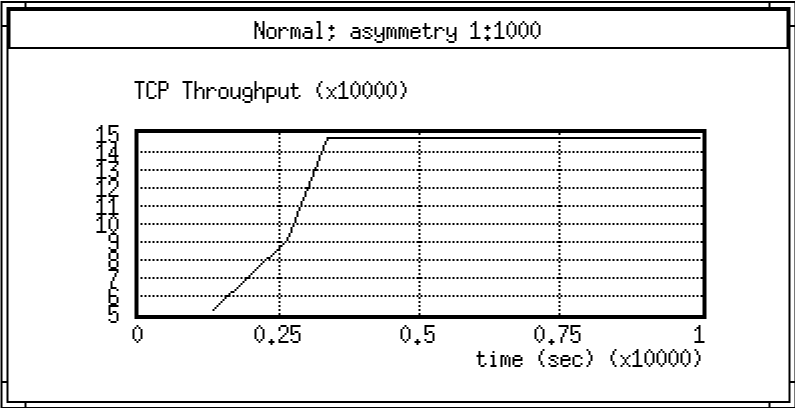
The simulations were performed on a simple network consisting of two nodes interconnected by two point-to-point links. The simulations were repeated for different asymmetry ratios, bit error rates and traffic loads.

Different asymmetry ratios

The objective of this set of simulations was to analyze the behavior of the booster at different ratios of bandwidth asymmetry. The forward transfer rate was set to 10Mbps and the simulations were done for varying degrees of asymmetry ranging from 1:1000 to 1:1. The results are presented in figures 4.6 to 4.10. It can be observed that, as the degree of asymmetry increases, the performance benefits gained from the booster correspondingly increases. At the largest asymmetry ratio of 1:1000 the booster yields maximum performance

gain (almost 83 %) while at an asymmetry ratio of 1:1 the performance with the booster is almost the same as that without the booster.

Thus it is apparent that the ACK reconstruction booster definitely enhances the throughput of TCP connections, whenever there is significant asymmetry in the channel.



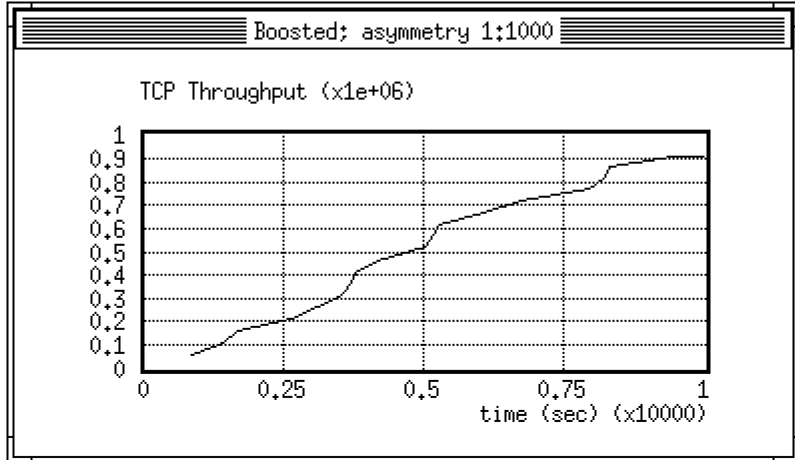
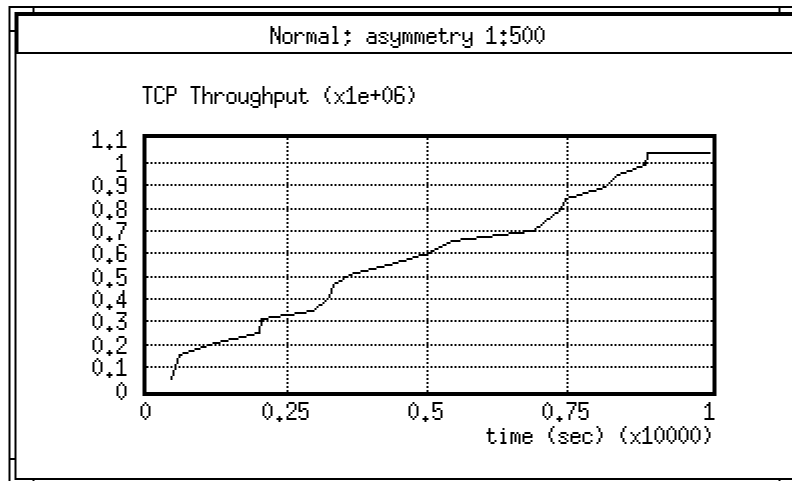


Figure 4.6: TCP throughput (number of bits transferred to application layer) with and without the booster for asymmetry ratio 1:1000



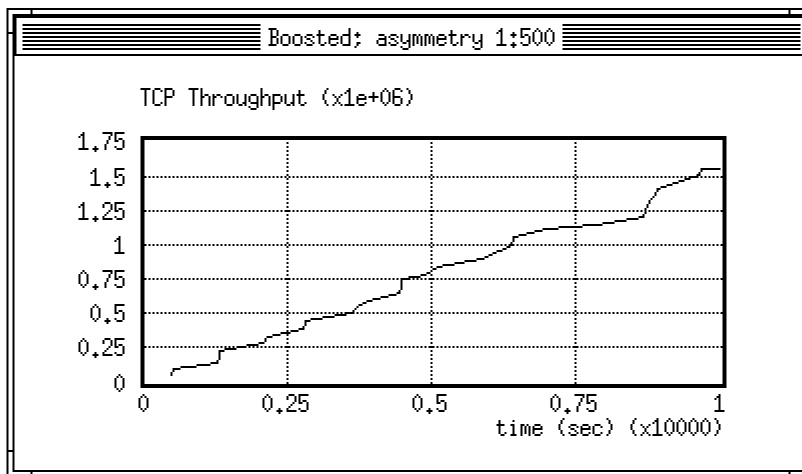
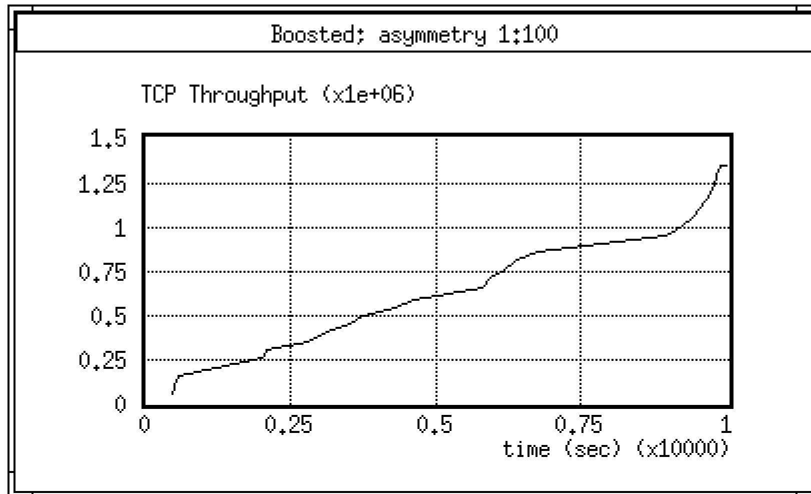


Figure 4.7: TCP throughput (number of bits transferred to application layer) with and without the booster for asymmetry ratio 1:500



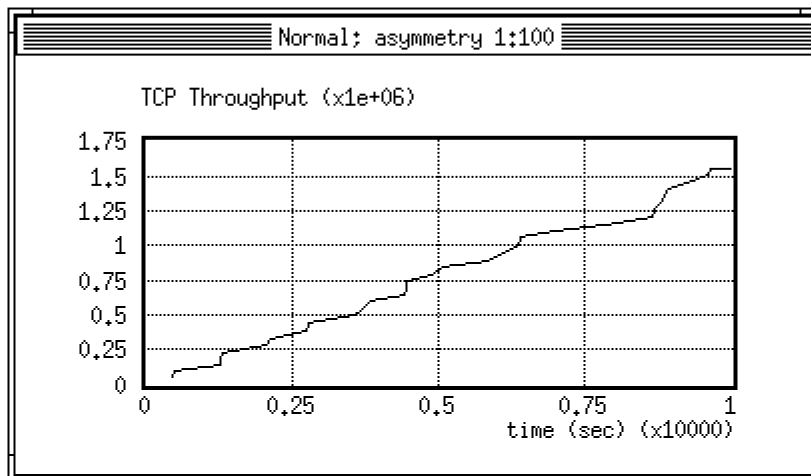
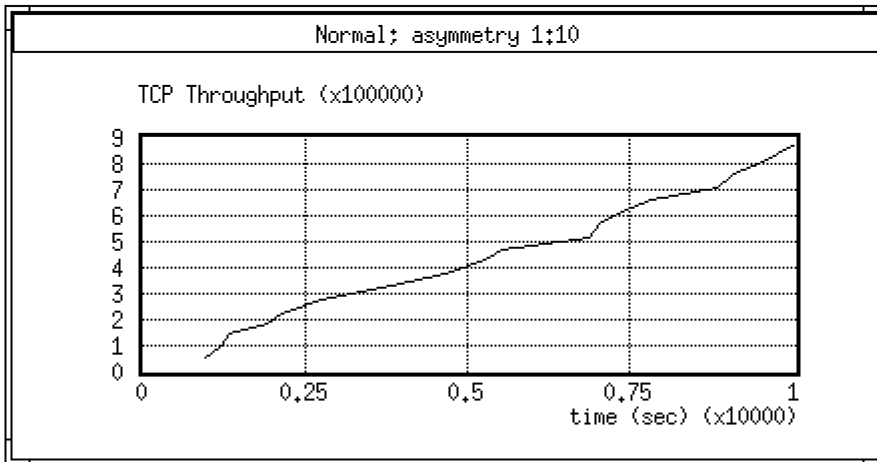


Figure 4.8: TCP throughput (number of bits transferred to the application layer) with and without the booster for asymmetry ratio 1:100


Normal; asymmetry 1:10
TCP Throughput (x100000)
9
8
7
6
5
4
3
2
1
0
0
0.25
0.5
0.75
1
time (sec) (x10000)

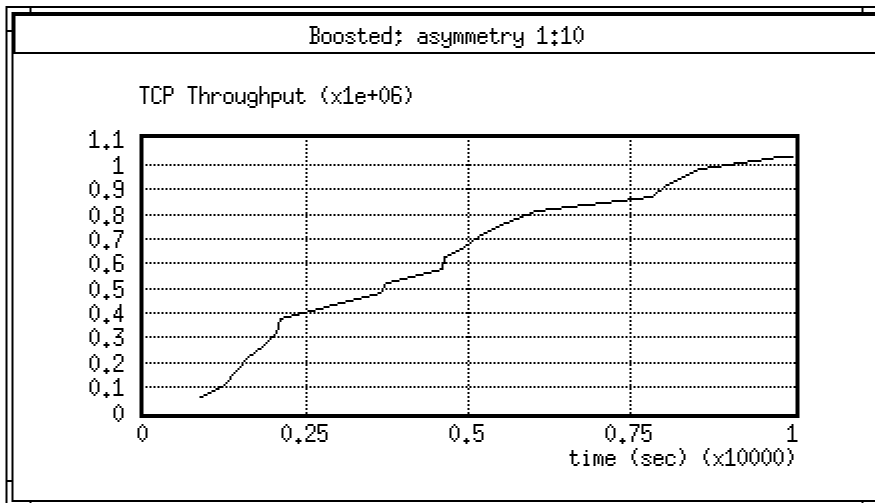
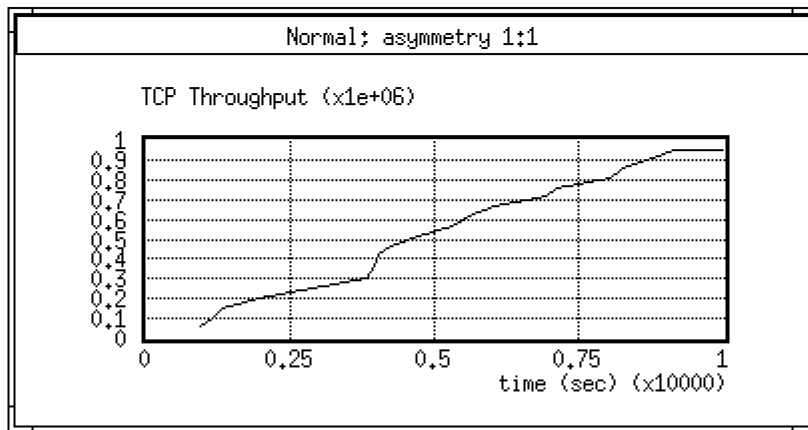

Boosted; asymmetry 1:10
TCP Throughput (x1e+06)
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.25
0.5
0.75
1
time (sec) (x10000)

Figure 4.9: TCP throughput (number of bits transferred to application layer) with and without the booster for asymmetry ratio 1:10



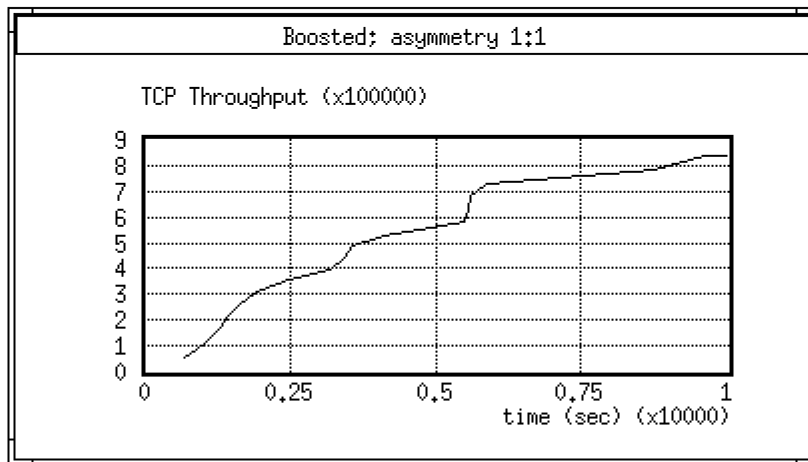
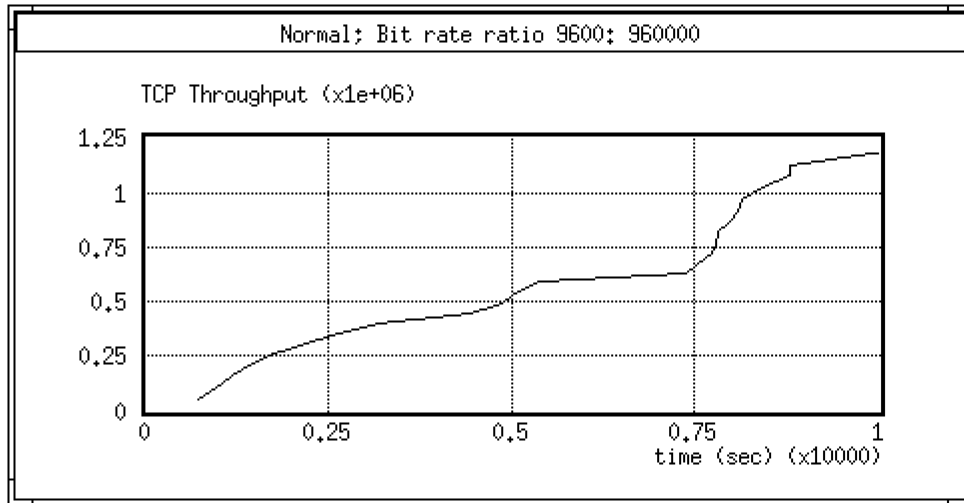


Figure 4.10: TCP throughput (number of bits transferred to application layer) with and without the booster for asymmetry ratio 1:1

Different reverse bandwidths

The objective of this set of simulations was to study the effect of higher line rate modems. Simulations were done for reverse bandwidths 9600 bps and 56000 bps. The asymmetry ratio was maintained at 1:100.

The results are presented in figures 4.11 and 4.12. It is observed that the booster continues to yield higher performance notwithstanding the higher reverse bandwidth. The utility of the booster is thus a function of the asymmetry ratio and not the reverse bandwidth.



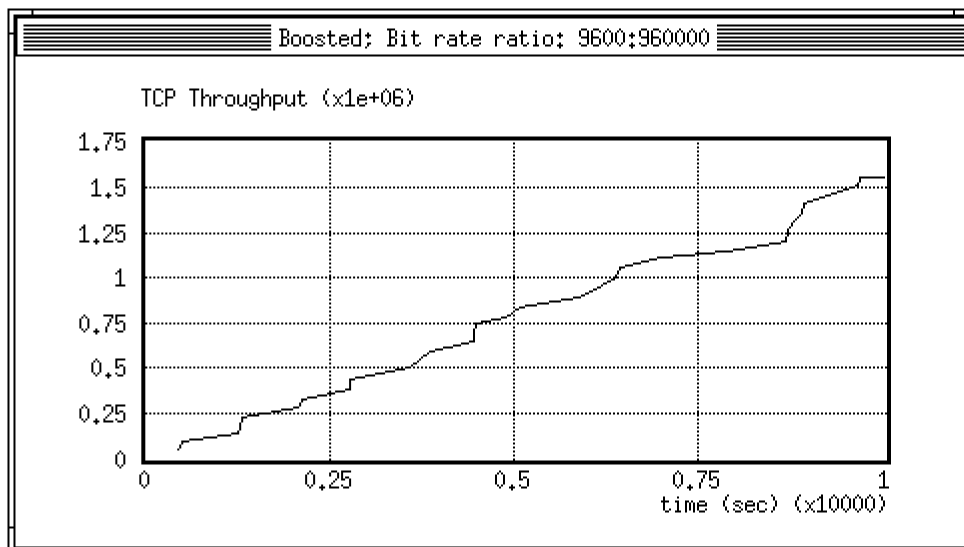
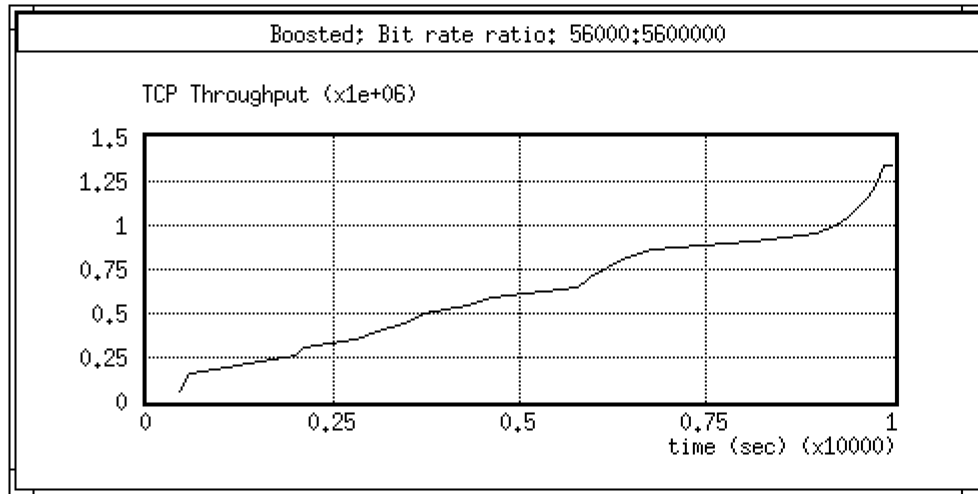


Figure 4.11 TCP throughput (number of bits transferred to application layer) with and without the booster at a bit rate ratio of 9600:960000



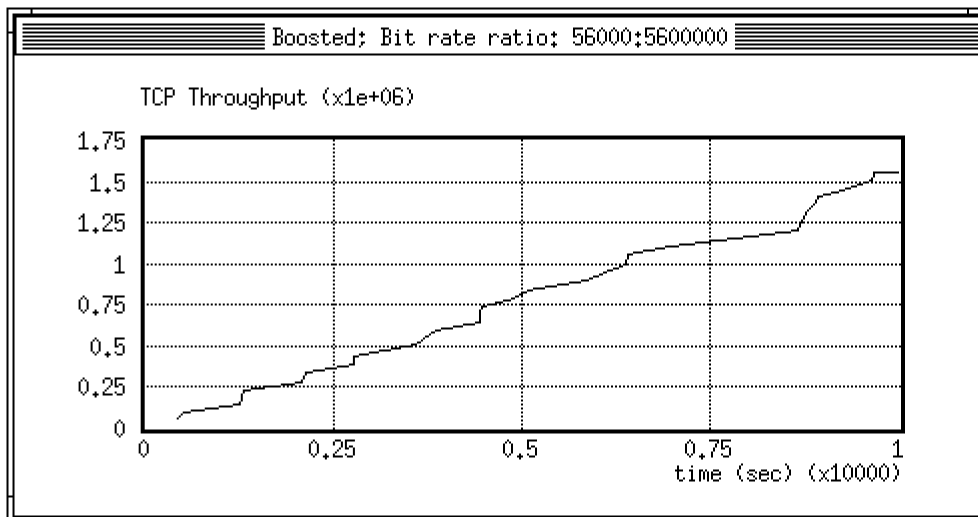
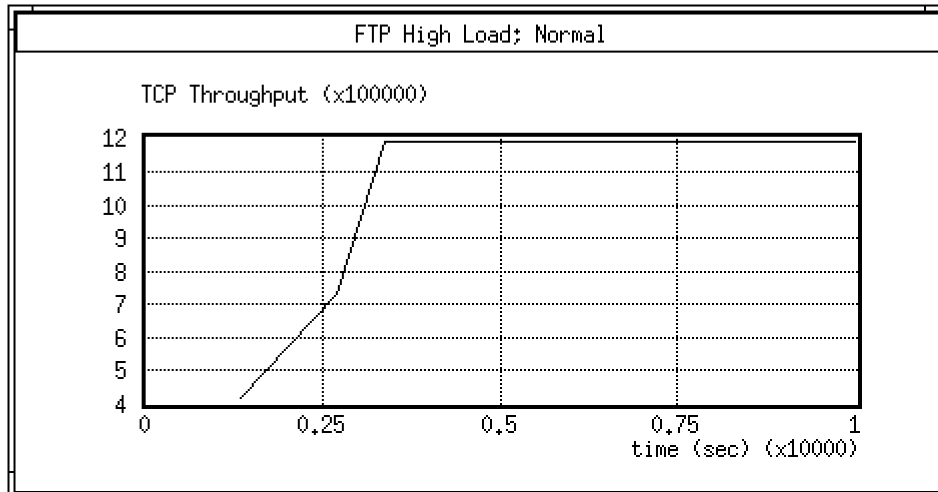


Figure 4.12: TCP throughput (number of bits transferred to application layer) with and without the booster at a bit rate ratio of 56000:5600000

Different traffic loads

The objective of this set of simulations was to study the performance of the booster under varying traffic loads and types. Simulations were done with varying FTP traffic loads – high, medium and low. Also, the performance of the booster under interactive applications like rlogin was studied. The results are shown in figures 4.13 to 4.16. It is observed that the booster yields the highest performance gain when there is heavy transfer of data in the forward direction and large number of ACKs traversing the reverse link (FTP high load). For interactive applications like rlogin, it is observed that the booster does not significantly improve performance. This can be attributed to the fact that there is no queuing up of ACKs. However for applications such as ftp which involve bulk data flow in the forward direction which in turn generate a correspondingly large number of ACKs, the booster definitely improves performance.



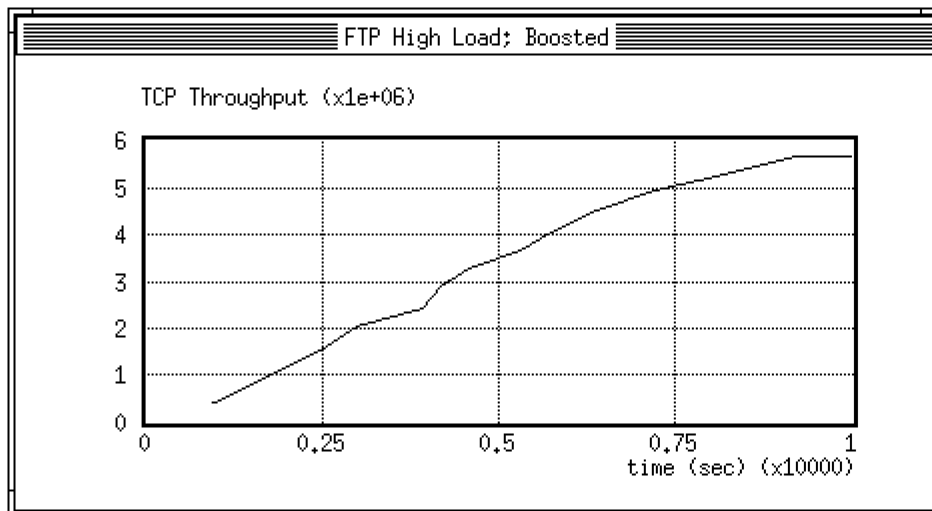
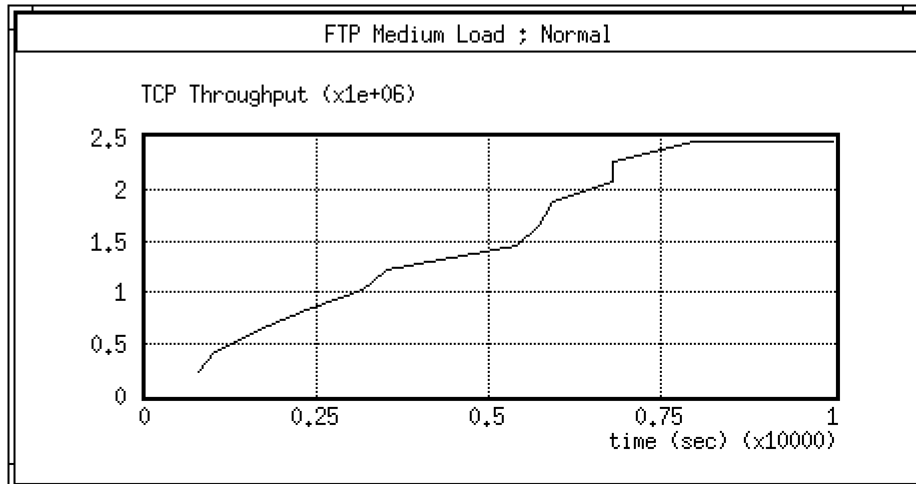


Figure 4.13: TCP throughput (total number of bits forwarded to the application layer) with and without booster for FTP-high load.


FTP Medium Load ; Normal
TCP Throughput (x1e+06)
2.5
2
1.5
1
0.5
0
0
0.25
0.5
0.75
1
time (sec) (x10000)

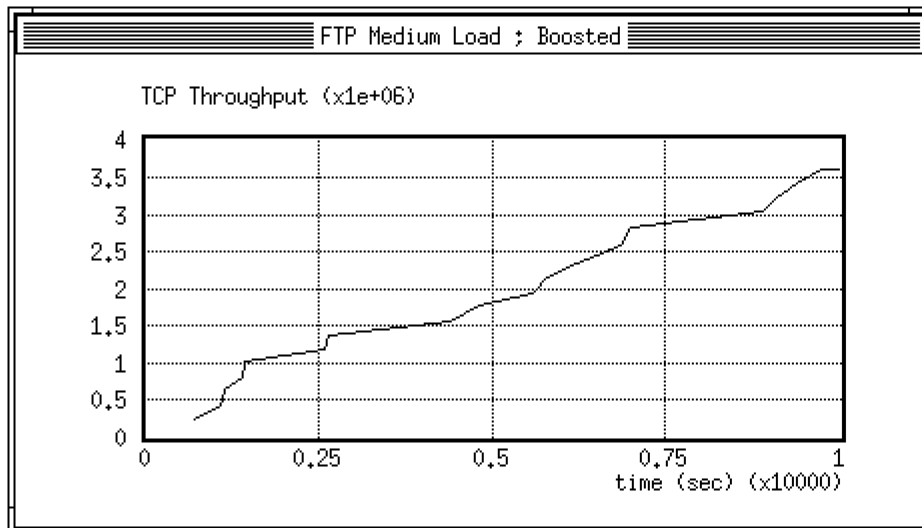
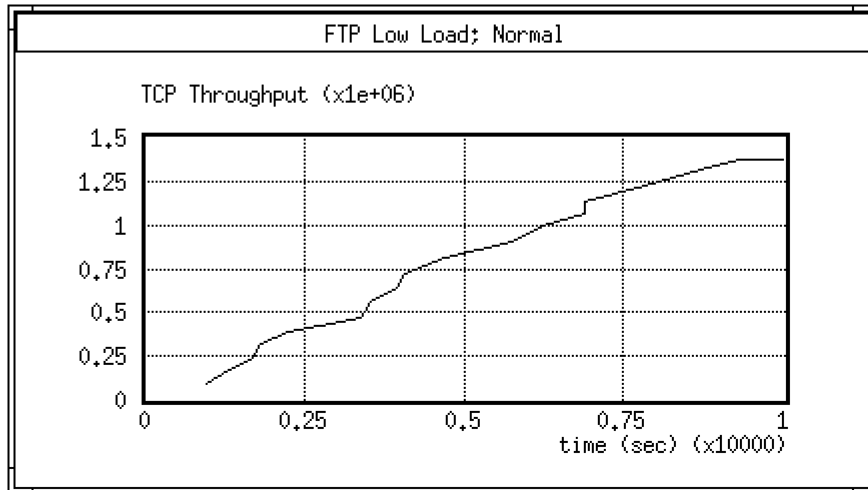

FTP Medium Load ; Boosted
TCP Throughput (x1e+06)
4
3.5
3
2.5
2
1.5
1
0.5
0
0
0.25
0.5
0.75
1
time (sec) (x10000)

Figure 4.14: TCP throughput (total number of bits forwarded to the application layer) with and without booster for FTP-medium load.



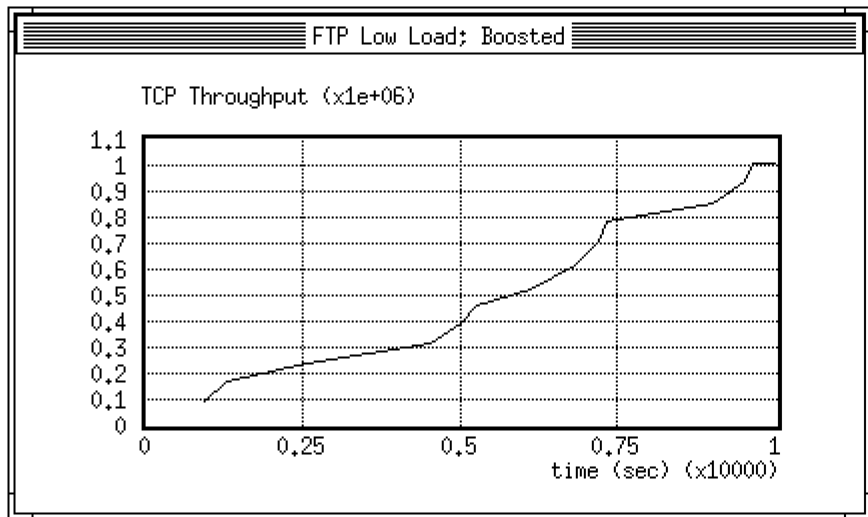
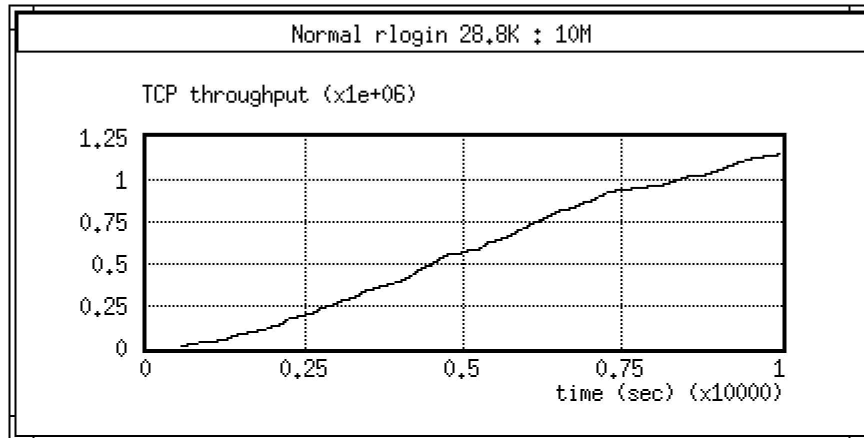


Figure 4.15: TCP throughput (total number of bits forwarded to the application layer) with and without booster for FTP-low load.



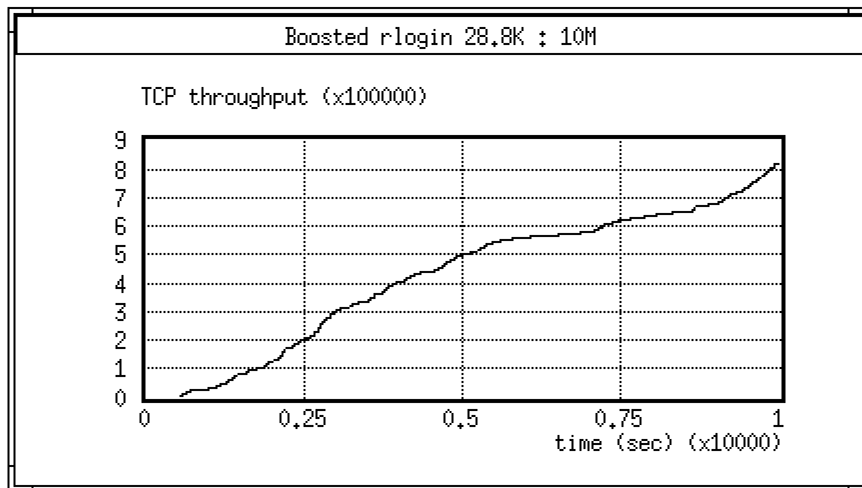
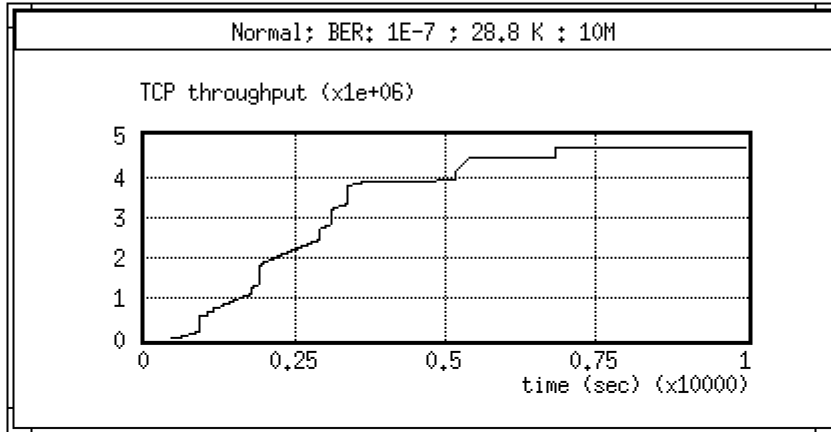


Figure 4.16: TCP throughput (total number of bits forwarded to the application layer) with and without booster for rlogin.

Different bit error rates

The objective of this set of simulations was to study the effect of increasing bit error rates on the performance of the booster. The simulations were done with a forward bandwidth of 10Mbps and reverse bandwidth of 28.8Kbps. The bit error rate of the link was varied from 1E-7 to 1E-5. The results are shown in figures 4.17 to 4.19. It is observed that as the bit error rate of the link increases, the throughput drops- but the booster still yields significant performance enhancement.


Normal; BER: 1E-7 ; 28.8 K : 10M
TCP throughput (x1e+06)
5
4
3
2
1
0
0
0.25
0.5
0.75
1
time (sec) (x10000)

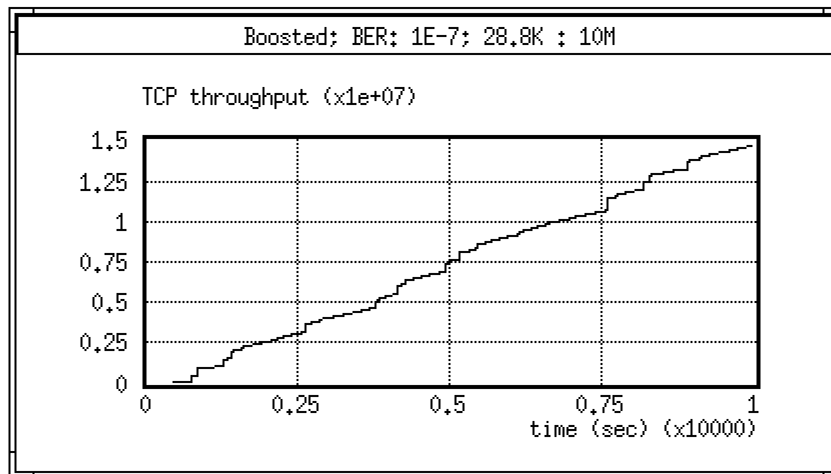

Boosted; BER: 1E-7; 28.8K : 10M
TCP throughput (x1e+07)
1.5
1.25
1
0.75
0.5
0.25
0
0
0.25
0.5
0.75
1
time (sec) (x10000)

Figure 4.17: TCP throughput (total number of bits forwarded to the application layer) with and without booster for BER: 1E-7.

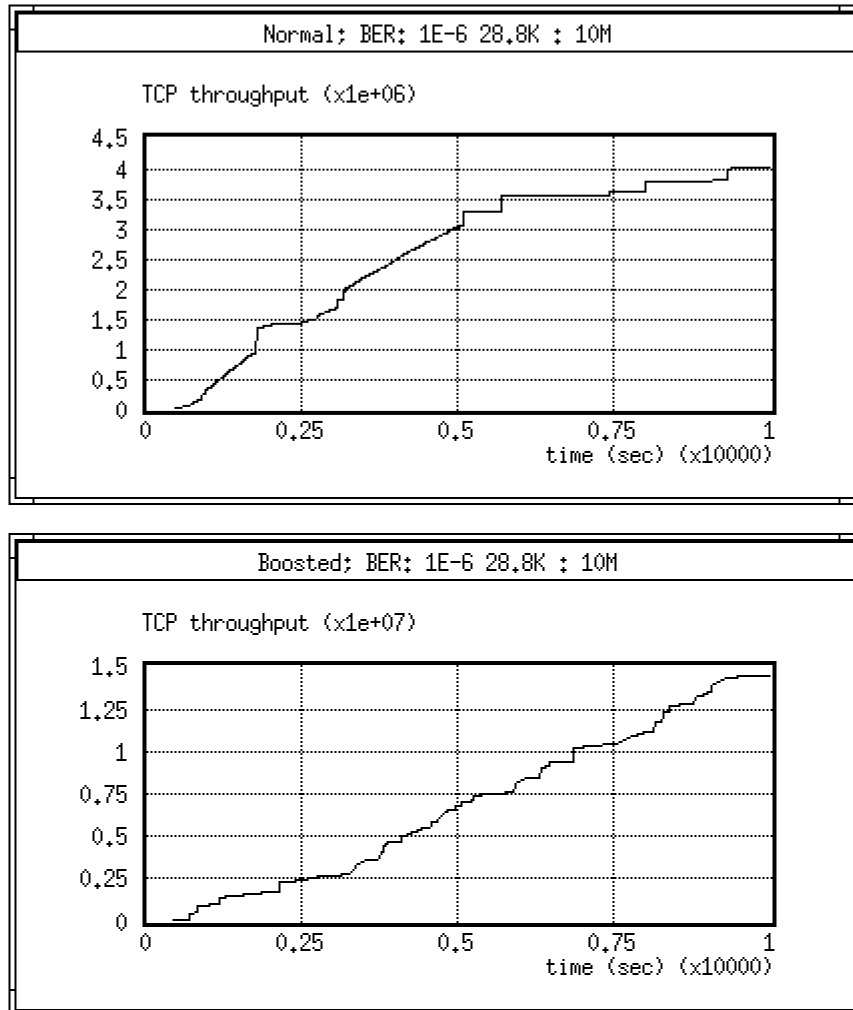
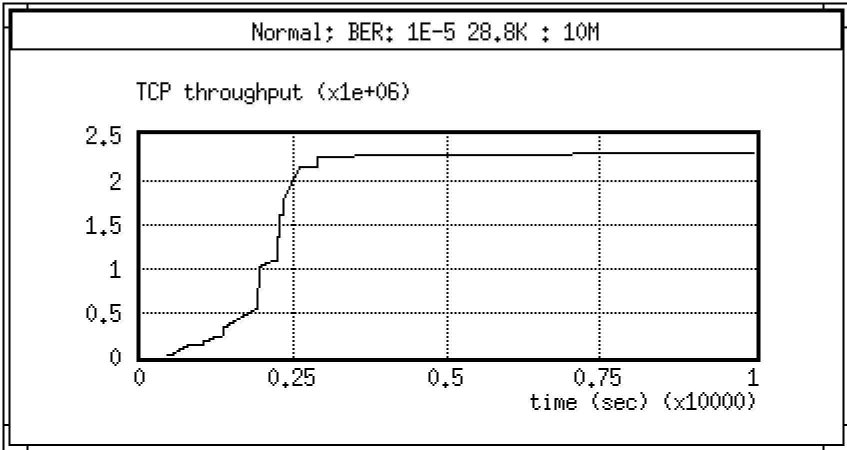


Figure 4.18: TCP throughput (total number of bits forwarded to the application layer) with and without booster for BER: 1E-6.


Normal; BER: 1E-5 28.8K : 10M
TCP throughput (x1e+06)
2.5
2
1.5
1
0.5
0
0
0.25
0.5
0.75
1
time (sec) (x10000)

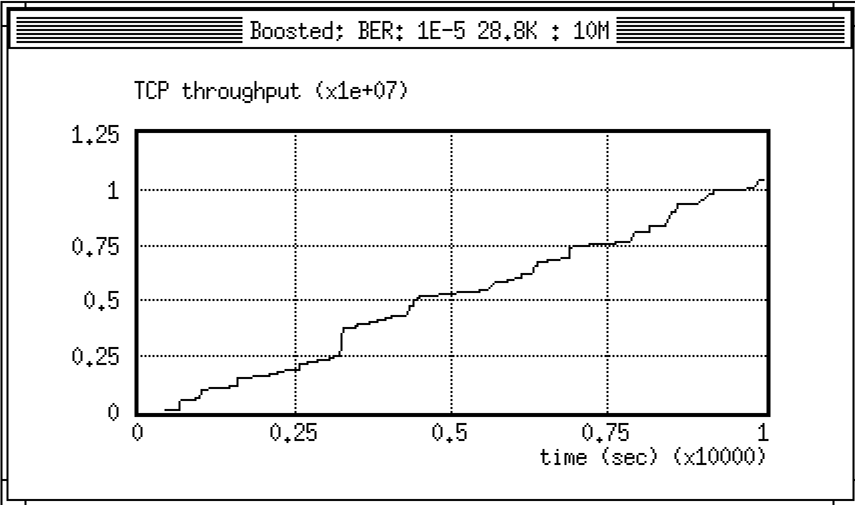
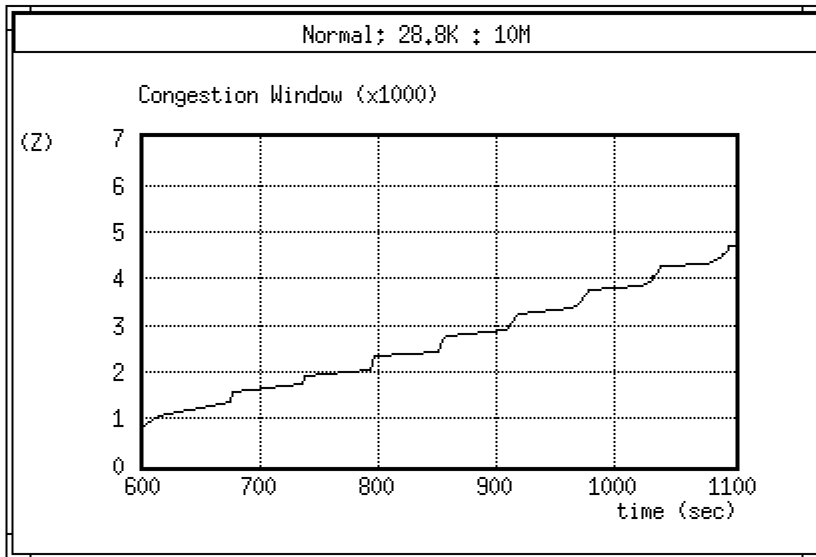

Boosted; BER: 1E-5 28.8K : 10M
TCP throughput (x1e+07)
1.25
1
0.75
0.5
0.25
0
0
0.25
0.5
0.75
1
time (sec) (x10000)

Figure 4.19: TCP throughput (total number of bits forwarded to the application layer) with and without booster for BER: 1E-5.

Congestion window

The growth of the congestion window was studied for the boosted and normal case. As shown in figures 4.20 and 4.21, the congestion window grows at a much faster rate for the boosted case when there is an asymmetry in the bandwidths of the forward and reverse links. When the link bandwidths are the same, the booster does not yield any significant improvement.


Normal; 28.8K : 10M
Congestion Window (x1000)
(Z)
7
6
5
4
3
2
1
0
600
700
800
900
1000
1100
time (sec)

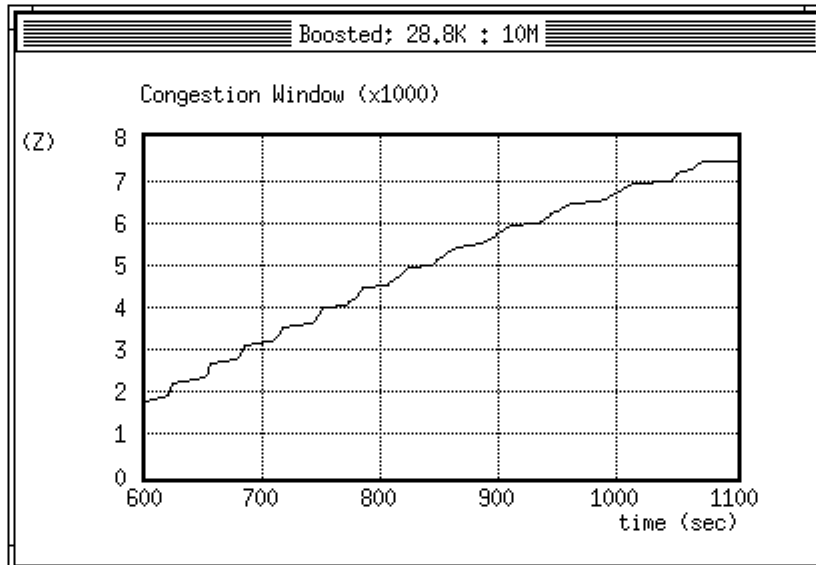
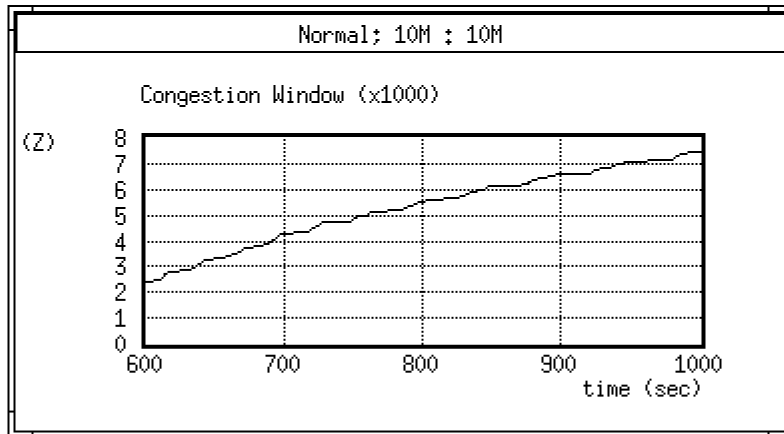

Boosted; 28.8K : 10M
Congestion Window (x1000)
(Z)
8
7
6
5
4
3
2
1
0
600
700
800
900
1000
1100
time (sec)

Figure 4.20: Congestion window (in bytes) growth, when there is channel bandwidth asymmetry for the boosted and normal case.


Normal; 10M : 10M
Congestion Window (x1000)
(Z)
8
7
6
5
4
3
2
1
0
600
700
800
900
1000
time (sec)

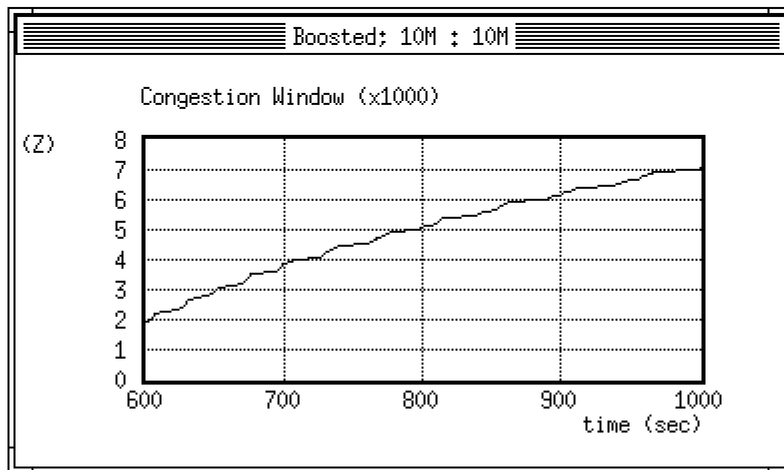

Boosted; 10M : 10M
Congestion Window (x1000)
(Z)
8
7
6
5
4
3
2
1
0
600
700
800
900
1000
time (sec)

Figure 4.21: Congestion window (in bytes) growth , when there is no channel bandwidth asymmetry for the boosted and normal case.

Chapter 5

The FZC Booster

5.1 Introduction

The Forward eraZure Correction (FZC) booster is a multi-element protocol booster, which reduces the effective packet loss rate on noisy links such as terrestrial and satellite wireless networks[16]. Although packet error correction is normally most efficiently and flexibly done by packet retransmission (automatic repeat request, ARQ), forward error correction (FEC) is desirable for some latency-constrained and multicast applications, or where the return channel is unavailable or slow and where the loss of a single packet causes other packets to need retransmission. Figure 5.1 highlights the operation of the FZC booster.

The FZC booster uses a packet FEC code with erasure decoding. This booster is a part of the TCP/IP booster family, which also includes ARQ boosters, a reorder booster, an error detection booster and an ACK reconstruction booster. Each booster is designed to provide a specific function and work harmoniously with the other boosters in the family. The FZC booster is not well suited for dealing with packet loss due to congestion; other members of the TCP/IP error control booster family handle this situation.

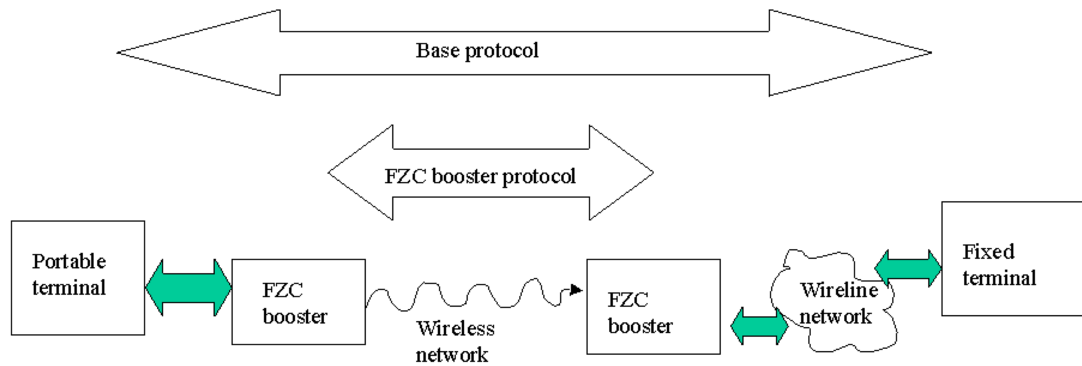


Figure 5.1 Communications over a wireless network with a FZC booster

5.2 The *Modus Operandi*

At the transmitter side of the wireless network, the FZC booster adds parity packets. The FZC booster at the receiver side of the wireless network removes the parity packets and regenerates missing data packets. This appears similar to link FEC; however, link FEC only corrects bits (or words), not IP packets or TCP segments, and cannot be applied between any two points in a network (including the end systems). Also, the FZC booster can be applied incrementally. In figure 5.1, for example, we could add an extra FZC booster at the fixed terminal. If this booster adds $h1$ parity packets and the second booster adds $h2$ parity packets, the portable terminal can recover from up to $h1+h2$ packet erasures. In the reverse direction, the second booster could reduce the number of redundant packets to reduce congestion in the wireline network.

5.3 Implementation in the Linux kernel

The FZC booster was implemented on the kernel infrastructure described in the previous chapter. This booster caches, then immediately forwards, each data packet it receives, whether the packet is from an upper layer protocol or the IP forwarder. The only modification to each data packet is that the FZC booster over writes the IP packet's 16-bit identification field with a sequence number, allowing the decoder to know the packet's position information. Practically speaking, this does not change the end-to-end UDP datagrams or TCP segments. However, if an application requires, IP options can be used for packet sequence information.

After receiving k packets (k is defined per channel) on a given channel, the cached packets are zero padded to the size of the largest packet in the cache. Also, each packet's size and protocol type are appended to the packet's tail. The transmitter performs an FZC matrix multiplication over the payload, padding, and appended tail of the k packets. The h overcode packet payloads produced by the FEC encoder are then prepended with an IP header and a booster header. This IP header contains a prototype field identifying it as a protocol booster packet and a sequence number in the 16-bit identification field. The booster header contains the type of booster (FZC booster), the value k, and the sequence number of the first of the k packets. The h packets are then transmitted towards the same destination as the data packets.

The FZC booster at the receive side caches incoming data packets and immediately forwards them either to an upper layer protocol or towards their eventual destination. Overcode packets are also cached, but are not forwarded. Packets are released from the cache when either:

1. An entire collection of k data packets is present.
2. The received data packets plus parity packets equal k.
3. The cache occupancy dictates cache content replacement.

Only in situation 2 are the matrix computations performed to generate the missing data packets.

To assess the effectiveness of this booster arrangement it was deployed in a simulated wireless environment running UDP. It was found that running the FZC booster using 4 percent $(k=50, h=2)$ and 30 percent $(k=20, h=6)$ overcode, respectively, reduces the throughput to approximately 7.7 Mbps and 3.9 Mbps from 9.6 Mbps, which is still very acceptable for most current access network technologies.

5.4 The OPNET Model

An OPNET model of the FZC booster has been developed. Since OPNET does not support modeling the actual data packets, the encoding and decoding operations of the FZC booster could not be modeled. However, using the data from the above tests, the effect of the FZC booster has been modeled in the transceiver pipeline stages of OPNET.

OPNET allows us to model the characteristics of three different types of communication links-point-to-point, bus and radio links. Each type of link provides a fundamentally different type of connectivity: point-to-point links connect a single source node to a single destination node; bus links connect a fixed set of nodes to each other; and radio links potentially allow all nodes in a model to communicate with each other, based on a dynamic evaluation. While the general type of connectivity provided by these links is predefined by OPNET, an open architecture is provided to allow developers to specify customized behavior for each individual link on a per-transmission basis. This architecture is referred to as the transceiver pipeline because it provides a conduit connecting a transmitter to one or more receivers.

The transceiver pipeline has a similar structure for each of the three supported link types. In each case, the simulation kernel manages the transfer of packets by implementing a series of computations, each of which models particular aspects of link behavior. The sequence of the computations and their interface are standardized for each type of link. However each computation, referred to as a pipeline stage, is performed outside the simulation kernel by a user-supplied procedure, called a pipeline procedure. In this manner, OPNET provides an open and modular architecture for implementing link behavior.

A link's underlying implementation can generally be thought of as a sequentially executed set of pipeline stages. The pipeline stage sequence of a link is executed once for each packet transmission that is submitted at the source of the link. In other words, when a packet is sent to a transmitter, the simulation kernel proceeds to call appropriate pipeline stages to process

the packet. Certain pipeline stages are executed when the packet is transmitted, and others are executed later due to the delay associated with the traversal of the link and transmission of the packet.

The principal objective of a transceiver pipeline is to determine whether or not a packet can be received at the link's destination. This determination is usually made in the final stages of the pipeline based on information computed during earlier stages. The vehicle used to convey information between the stages is the packet itself, which is provided as an argument to each pipeline procedure. The packet is also used by the pipeline stags to return information to the simulation kernel.

Figure 5.2 illustrates the radio link transceiver pipeline execution sequence. The FZC booster is modeled in the transmission delay pipeline stage and the error correction pipeline stage. The transmission delay stage is the first stage of the pipeline, and is specified by the *"txdel model"* attribute of the point-to-point link. It is invoked immediately upon beginning transmission of a packet, in order to calculate the amount of time required for the entire packet to complete transmission. This result can be thought of as the simulation time difference between the beginning of transmission of the first bit and the end of transmission of the last bit of the packet. The simulation kernel uses the result provided by this stage to schedule an end-of-transmission event for the transmitter

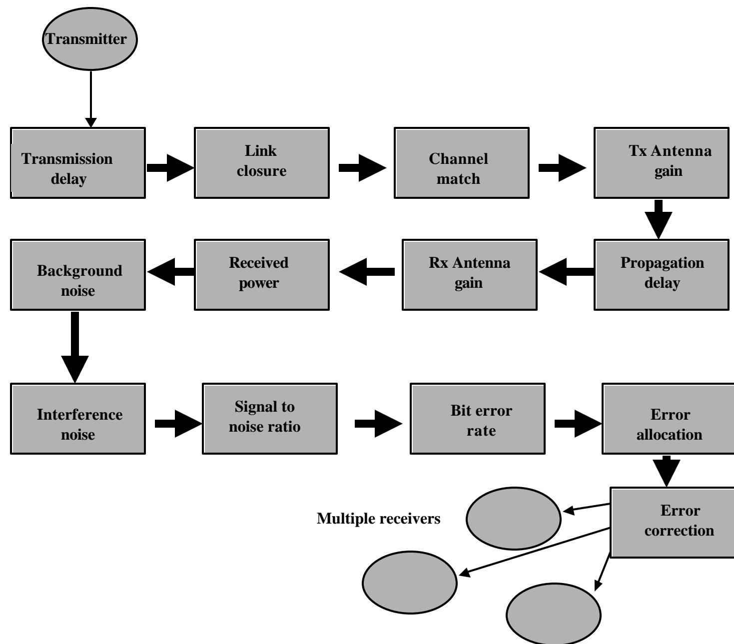


Figure 5.2: Radio Link Transceiver Pipeline Stages

channel that is used to send the packet. When this event occurs, the transmitter may begin transmission of the next packet in the channel's internal queue, if any are present; otherwise the transmitter channel becomes idle. In addition, the transmission delay result is used in conjunction with the result of the propagation delay stage to compute the time at which the packet completes reception at the link's destination (i.e., the time at which the last bit finishes arriving is the time at which it finishes transmitting added to the propagation delay on the link). The transmission delay is computed as *packet_length/data_rate* ; the *data_rate* being the data rate of the link. The performance cost (i.e., the reduction in throughput) that results from using the booster, can be modeled by reducing the *data_rate* appropriately, before it is used to compute the transmission delay. A variable *degradation* reflects the reduction in

throughput, as a consequence of the redundancy of the parity packets. This variable *degradation* is computed based on the *overcode* (the percentage of packet erasures sought to be corrected). This *overcode,* is a user configurable parameter, which can be set at run time. The relation between the *overcode* and the *degradation* is arrived at, using the data supplied from the real-life experiments.

Based on the available data, we have derived: *degradation* = 0.862 – 1.538 * *overcode.* For example, to correct 2 packets out of 50 packets (*4 percent overcode*), the *degradation* = *0.862 – 1.538 * 0.04 ~ 0.8* which is in agreement with the actual experimental data. So we multiply the *data_rate* by 0.8 to simulate the performance cost that results from using this booster.

The closure stage is the second stage of the pipeline. The purpose of this stage is to determine whether a particular receiver channel can be affected by a transmission. The ability of the transmission to reach the receiver is referred to as closure between the transmitter channel and the receiver channel, hence the name of the stage. Note that the goal of the closure stage is not to determine if a transmission is valid or appropriate for a particular channel, but only if the transmitted signal can physically attain the candidate receiver channel and affect it in any way; thus this stage applies to interfering transmissions as well as to desired ones.

The channel match stage is the third stage of the pipeline. The purpose of this stage is to classify the transmission with respect to the receiver channel. One of three possible categories must be assigned to the packet, as defined below:

- Valid. Packets in this category are considered compatible with the receiver channel and will possibly be accepted and forwarded to other modules in the receiving node, provided that they are not affected by an excessive number of errors
- Noise. This classification is used to identify packets whose data content cannot be received, but that have an impact on the receiver channel's performance by generating interference.
- Ignored. If a transmission is determined to have no effect whatsoever on a receiver channel's state or performance, then it should be identified using this classification.

The transmitter antenna gain is the fourth stage of the radio transceiver pipeline. The purpose of this stage is to compute the gain provided by the transmitter's associated antenna, based on the direction of the vector leading from the transmitter to the receiver.

The propagation delay stage is the fifth stage of the pipeline. The purpose of this stage is to calculate the amount of time required for the packet's signal to travel from the radio transmitter to the radio receiver. This result is generally dependent on the distance between the source and the destination.

The receiver antenna gain is the sixth stage of the pipeline. Its purpose is to compute the gain provided by the receiver's associated antenna, based on the vector leading from the receiver to the transmitter.

The seventh stage of the radio transciever pipeline is the Receiever Power stage. This stage computes the received power of the arriving packet's signal based on factors such as the power of the transmitter, the distance separating the transmitter and the receiver, the transmission frequency, and transmitter and receiver antenna gains.

The background noise stage is the eighth stage of the pipeline. The purpose of this stage is to represent the effect of all noise sources, except for other concurrently arriving transmissions, since these are accounted for by the interference noise stage. The expected result is the sum of the power of other noise sources, measured at the receiver's location and in the receiver channel's band. Typical background noise sources include thermal or galactic noise, emissions from neighbouring electronics, and otherwise unmodeled radio transmissions (e.g., commercial radio, amateur radio, television, depending on frequency). It takes as input, the Rx noise figure – an integer value which indicates the noise level associated with a particular receiver. In most of our simulations we evaluate the performance of the FZC booster by varying this Rx noise figure.

The ninth stage is the interference noise stage and its purpose is to account for the interactions between transmissions that arrive concurrently at the same receiver channel

The SNR stage is the tenth stage and it computes the current average power SNR result for the arriving packet. This calculation is usually based on values obtained during earlier stages, including received power, background noise and interference noise

The Bit error rate stage is the eleventh stage and its purpose is to derive the probability of bit errors during the past interval of constant SNR

The error allocation stage is the twelfth stage of the pipeline and its purpose is to estimate the number of bit errors in a packet segment where the bit error probabilityhas been calculated and is constant. Bit error count estimation is usually based on the bit error probability and the length of the affected segment.

The error correction effect of the FZC booster is modeled in the error correction pipeline stage. The error correction stage is the final stage of the pipeline and is specified by the *"ecc model"* attribute of the link. The purpose of this stage is to determine whether or not the arriving packet can be accepted and forwarded via the channel's corresponding output stream to one of the neighboring modules in the destination node. This is usually dependent upon the result computed in the error allocation stage and the ability of the receiver to correct the errors affecting the packet, hence the name of the stage. Based on the determination of this stage, the kernel will either destroy the packet, or allow it to proceed into the destination node. In addition, this result affects error and throughput statistics collected for the receiver channel. A

counter, *pkt_count* keeps track of the number of packets received. It is decremented on the arrival of a new packet. Another counter, *count_parity* keeps track of the number of packets that can be corrected. The initial value of *count_parity = pkt_count * overcode*. The value of *count_parity* is decremented by one, every time a packet has been corrected. When a packet arrives at the error correction transceiver pipeline stage, the number of errors in the packet is computed. Then, a test is made to determine whether the packet can be accepted based on the error correction threshold of the receiver. If the number of errors is lesser than or equal to the error correction threshold, then the packet is accepted, else it is rejected. A variable *accept* is assigned the value *OPC_TRUE* or *OPC_FALSE* , depending on whether the packet is accepted or rejected. When a packet has been rejected *(i.e., accept ==OPC_FALSE)*, if *count_parity* has a non zero value, the packet is accepted and the value of accept is changed to *OPC_TRUE*. The counter, *count_parity* is then decremented by one. When the counter *pkt_count* becomes zero, it is reinitialized and *count_parity* is reinitialized to (*pkt_count * overcode).*

5.5 Results

The above model for the FZC booster has been incorporated in the ARL simulation testbed consisting of a network of several mobile units communicating via wireless links.

Effect of level of noise on the performance gain

The noise level (the Rx noise figure – as discussed in the background noise pipeline stage) of the links were varied over a wide range, spanning several orders of magnitude and the throughput attained was recorded. For purposes of comparison, the same was done for the network, without the booster. The results, show that below a certain noise level, the throughput of the normal (i.e., "un-boosted") network was greater than the boosted network. But above that threshold, the performance of the boosted network improves and is several magnitudes greater than the normal network. This is because for lower bit error rates, the network has the overhead of the additional parity packets without any benefit. But for higher bit error rates, the error correction achieved, compensates for the overhead and the boosted network outscores the "un-boosted" network. Based on these results, it is suggested that the FZC booster be employed only in situations warranting its need i.e., in networks with links with high bit error rates. This can be done by having a policy module, which loads the booster only when the bit error rate exceeds a certain threshold.

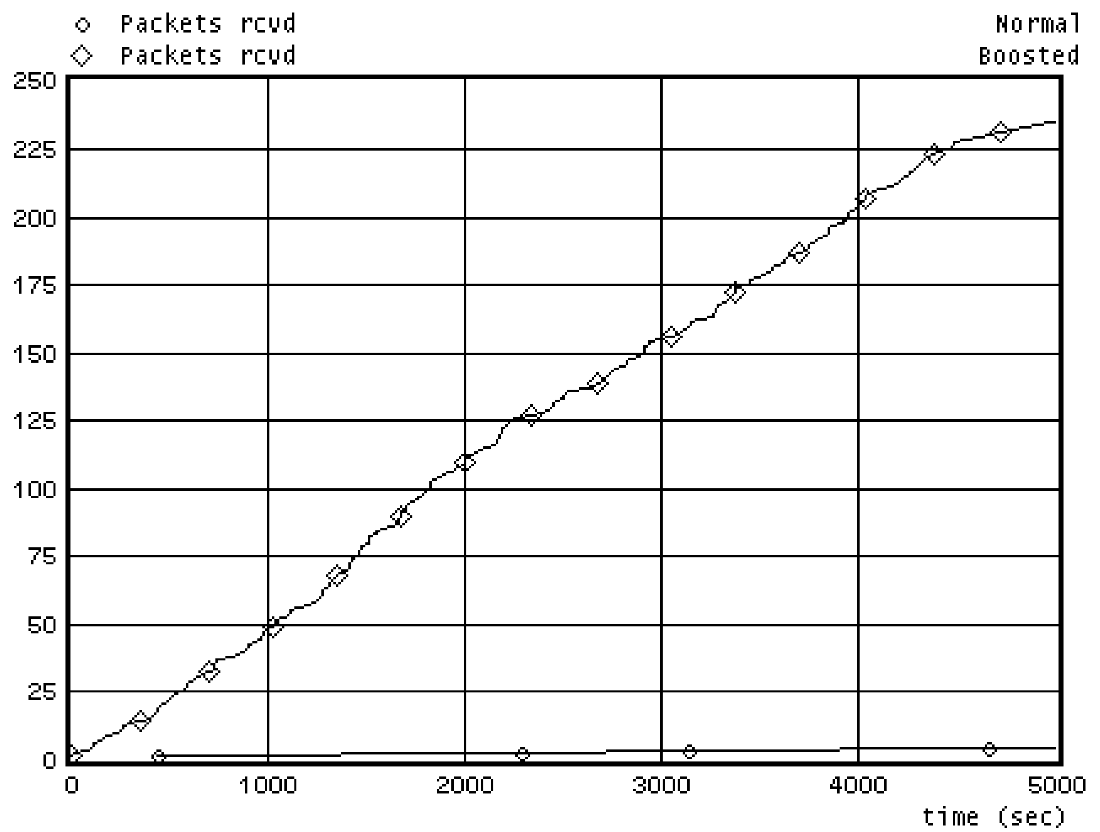


Figure 5.3: Comparison of normal and boosted network at noise level 75000

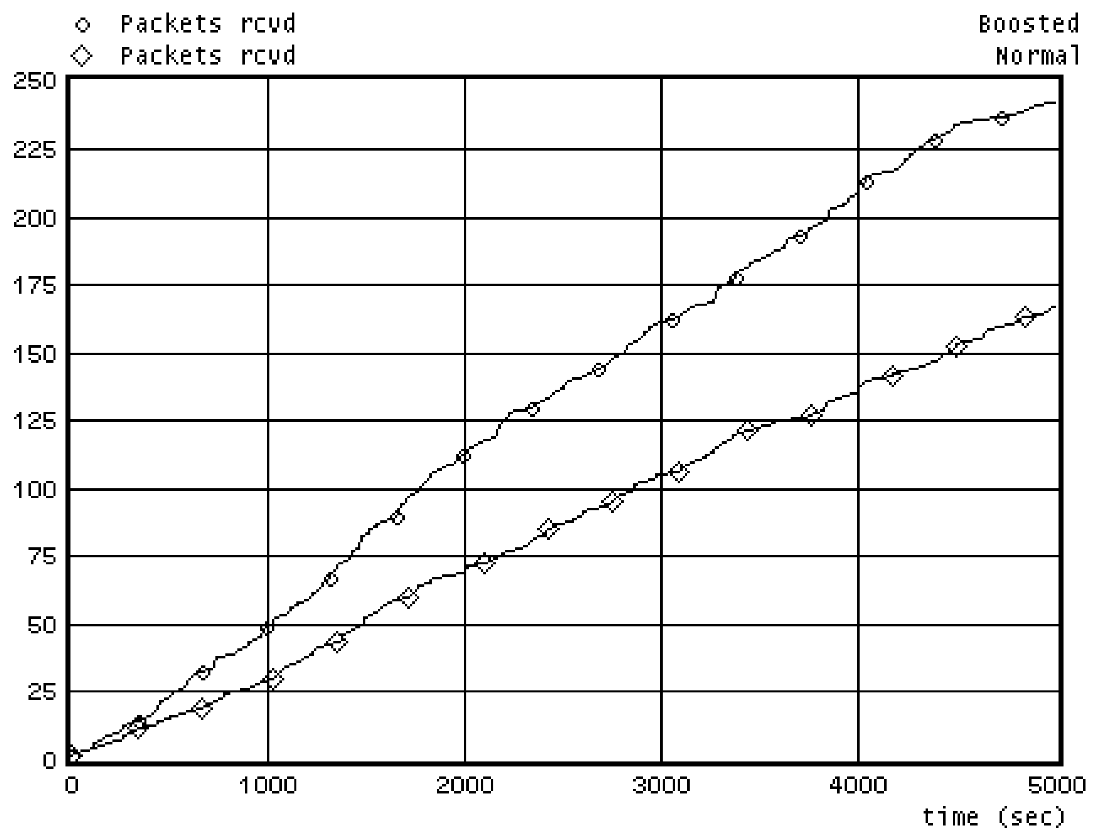


Figure 5.4: Comparison of normal and boosted network at noise level 50000

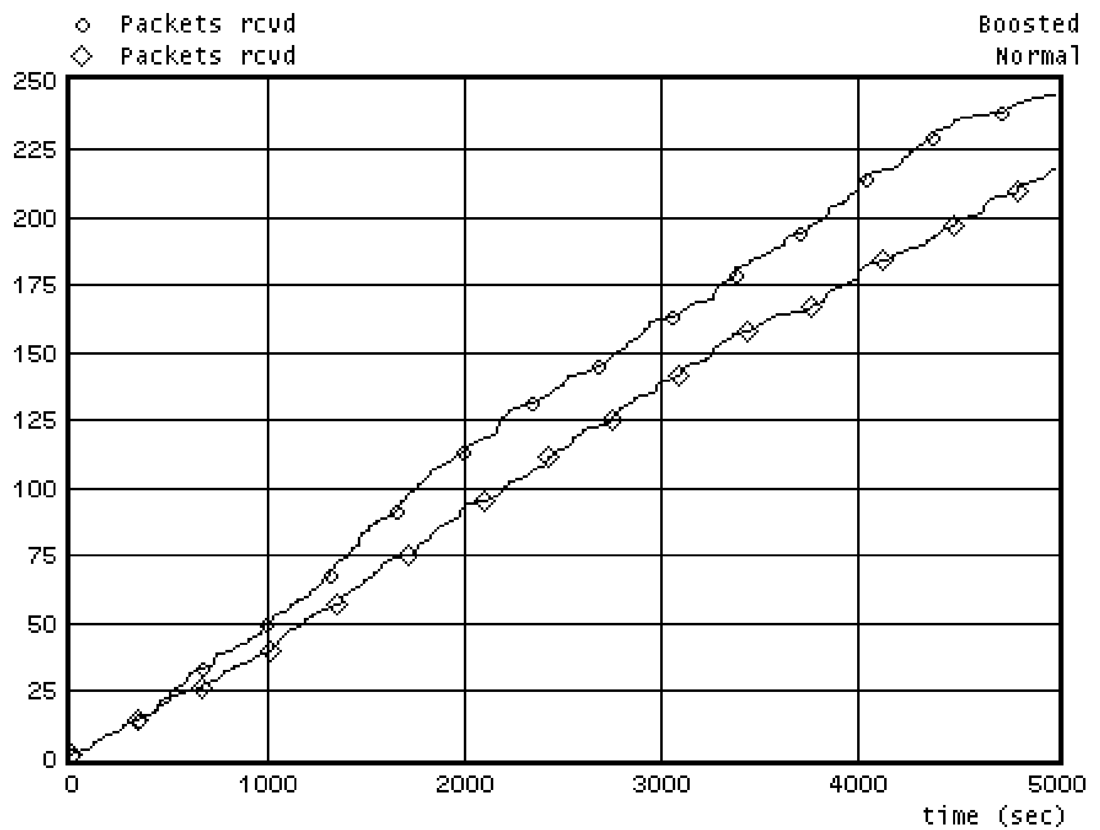


Figure 5.5: Comparison of normal and boosted network at noise level 25000

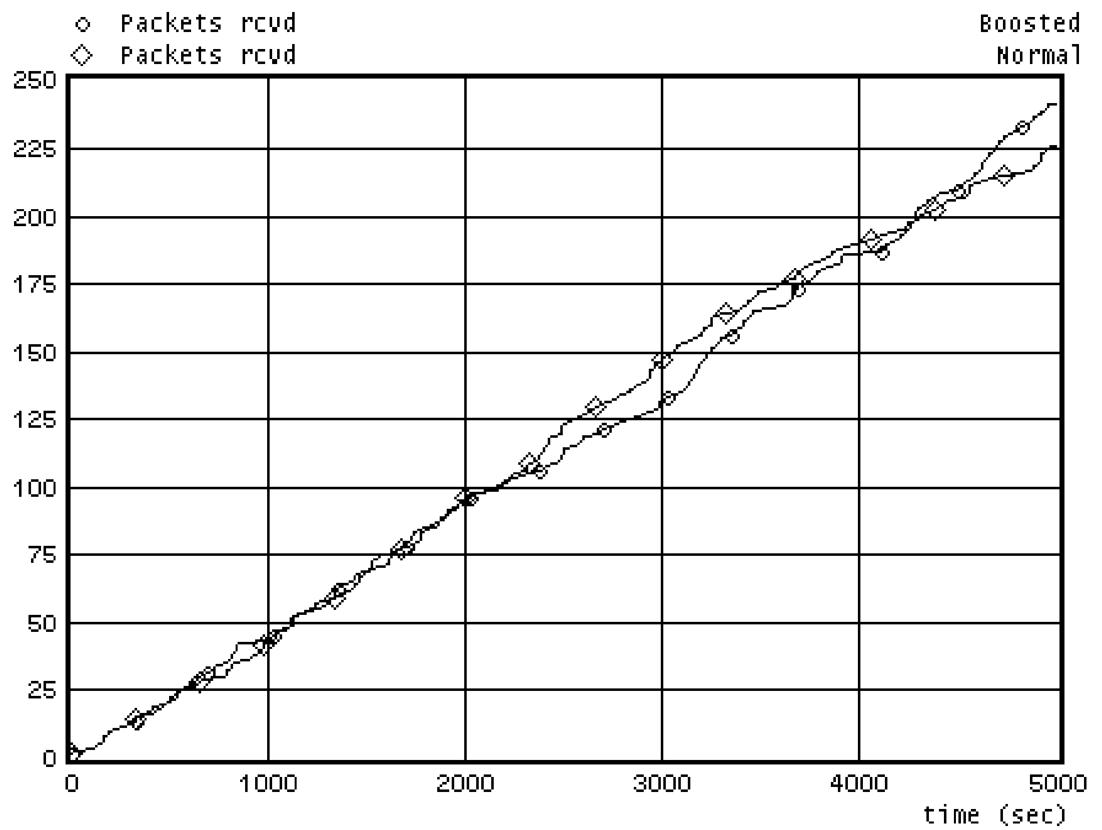


Figure 5.6: Comparison of normal and boosted network at noise level 9500

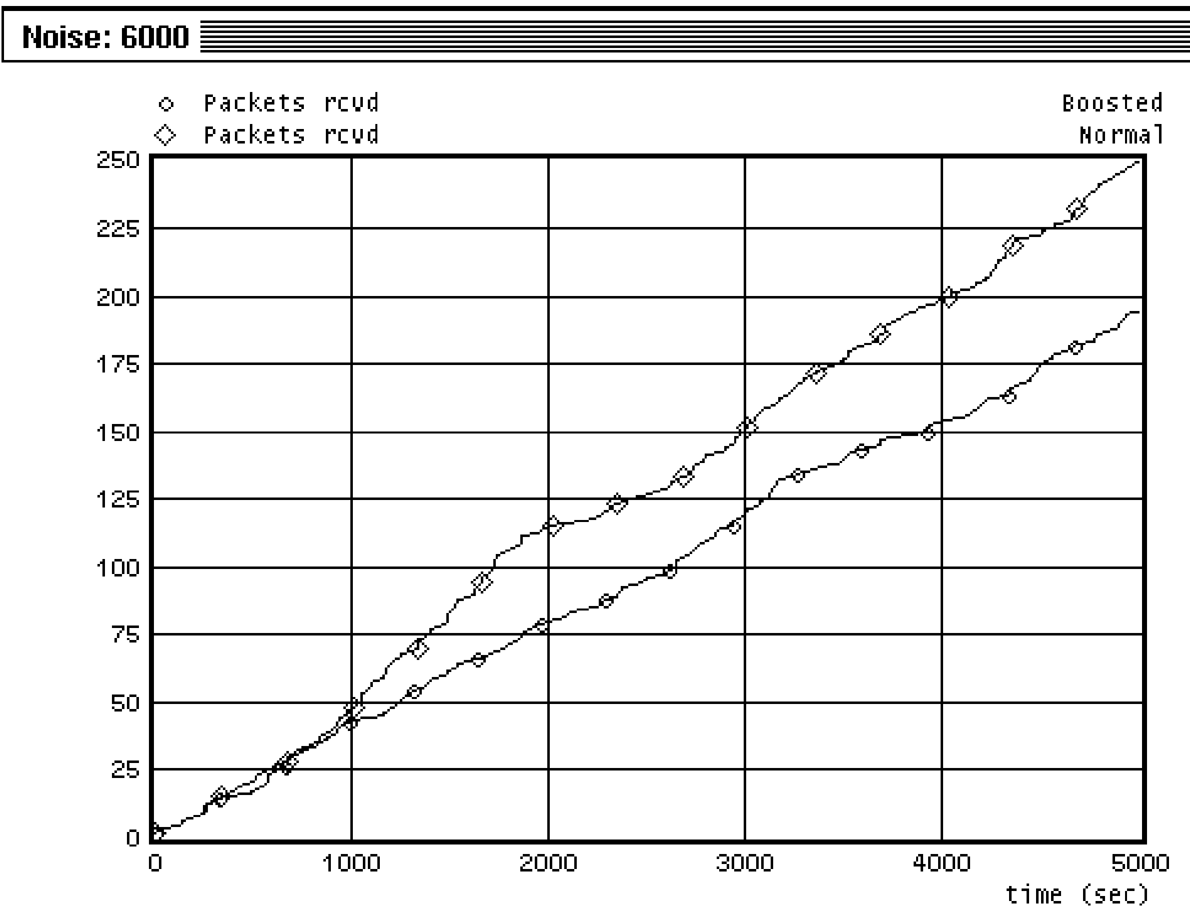


Figure 5.7: Comparison of normal and boosted network at noise level 6000

Effect of packet size on performance

In order to study the effect of packet size on the performance of the booster we ran simulations with packet sizes ranging from 512 bytes to 16000 bytes. It has been noticed that packet size has minimal effect on the booster performance. The booster continues to yield consistently high throughput for various packet sizes.

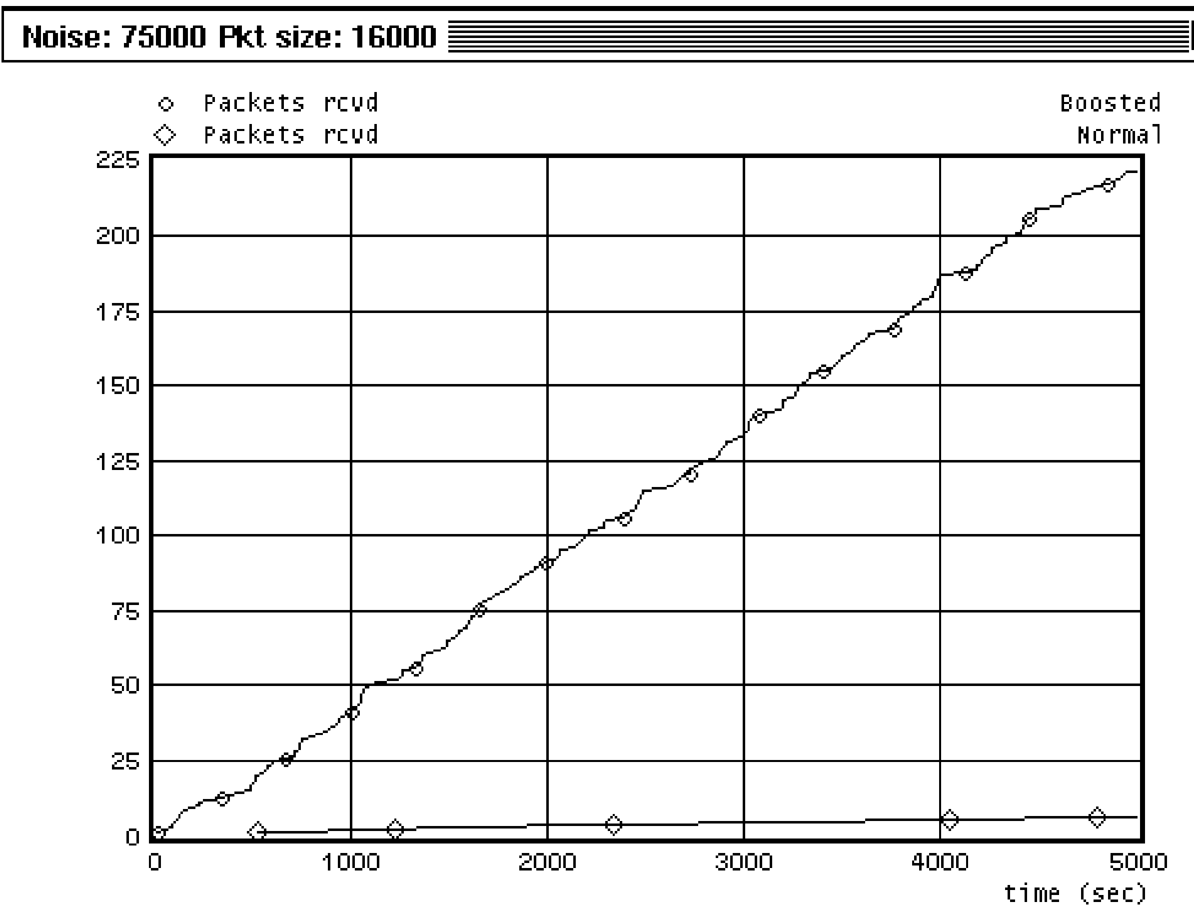


Figure 5.8: Comparison of normal and boosted network at noise level 75000 and packet size 16000 bytes

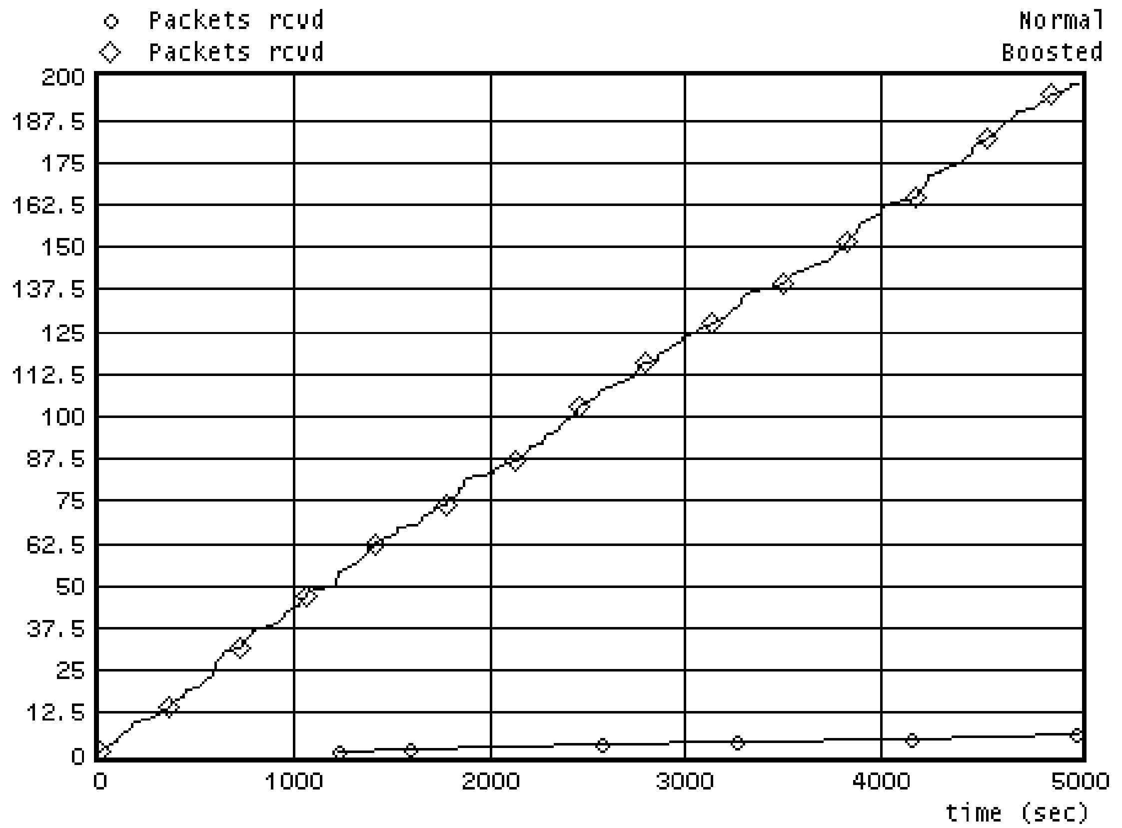


Figure 5.9: Comparison of normal and boosted network at noise level 75000 and packet size 4096 bytes

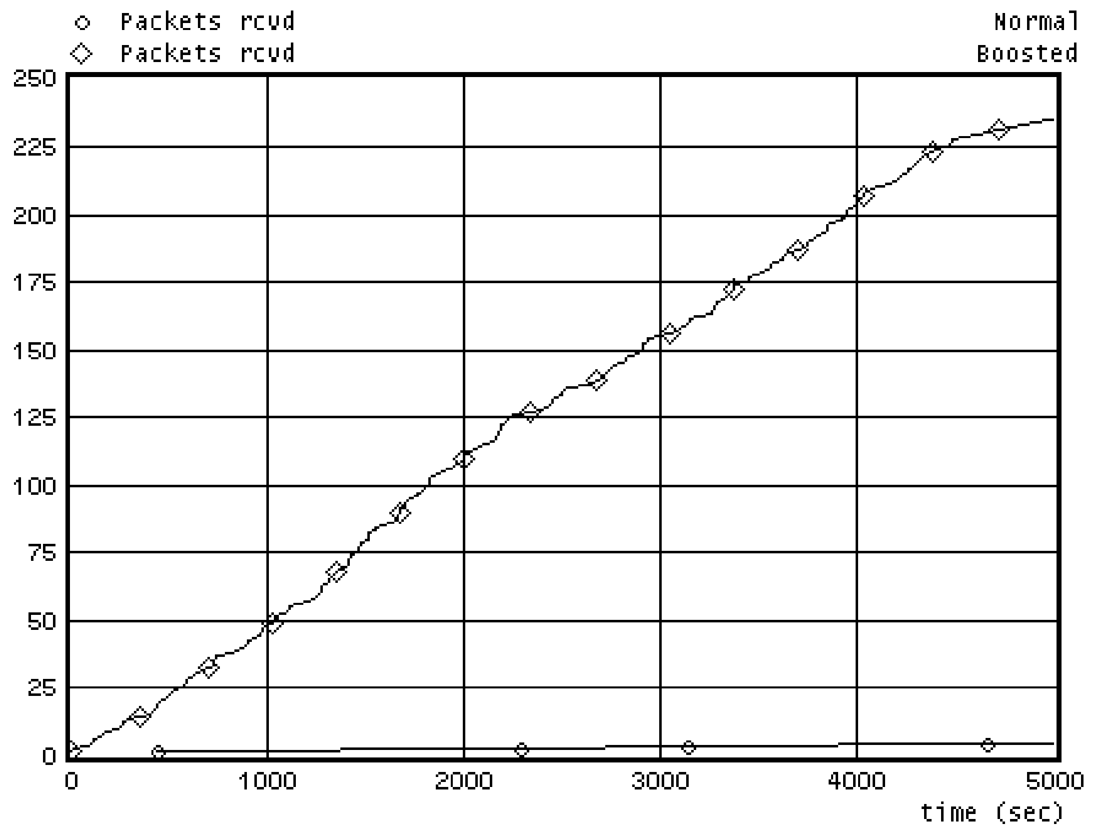


Figure 5.10: Comparison of normal and boosted network at noise level 75000 and packet size 1024 bytes

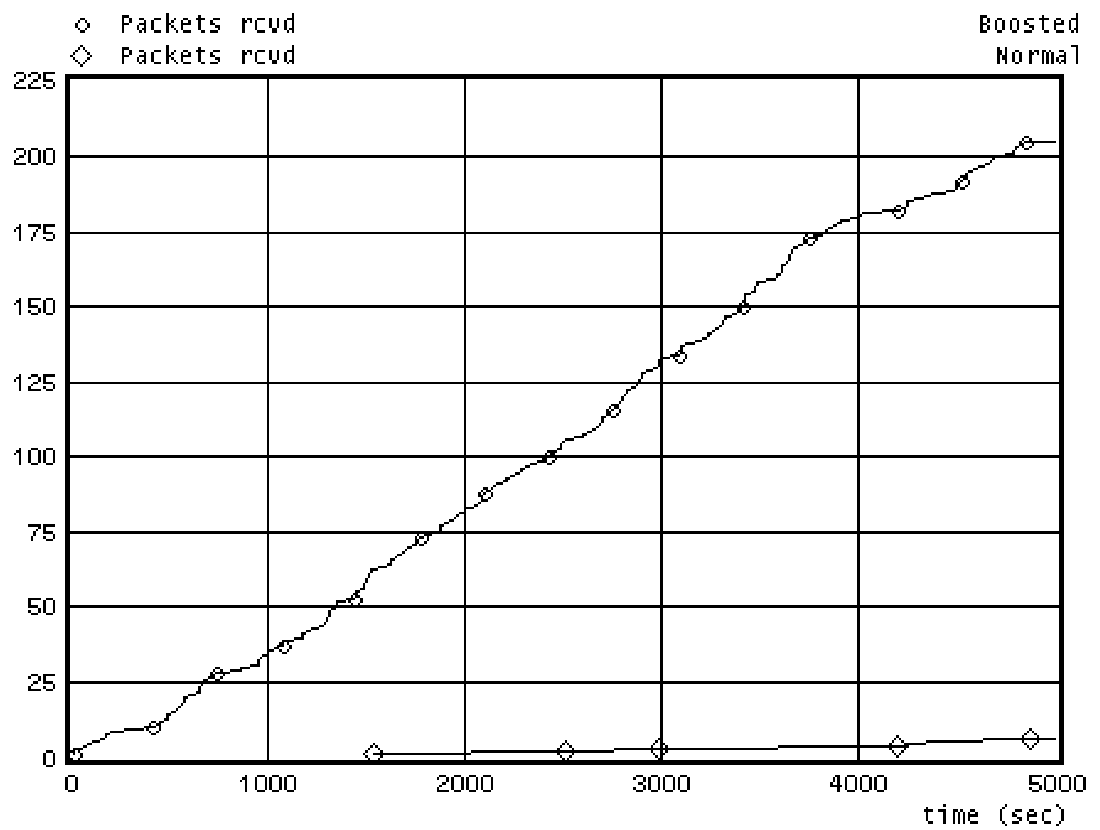


Figure 5.11: Comparison of normal and boosted network at noise level 75000 and packet size 512 bytes

Effect of different percentage overcodes

The same set of simulations were repeated with different overcode percentages. Figure 5.11 represents the comparison of boosted vs normal network at noise level 750000 and 4% overcode. Figures 5.12 is the simulation result for 30% overcode. It is observed that higher percentage overcode yields better performance over highly noisy links.

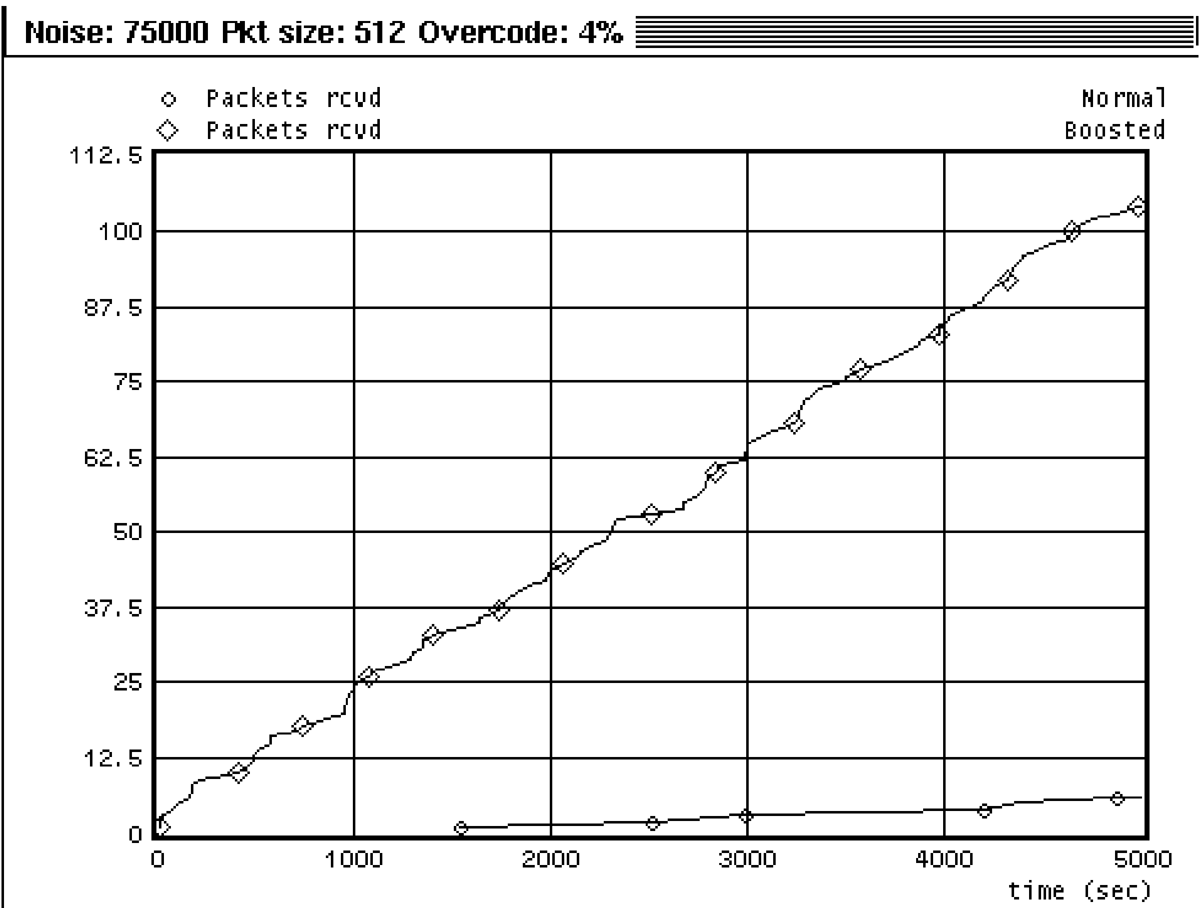


Figure 5.12: Comparison of normal and boosted network at noise level 75000 and packet size 512 bytes with overcode 4%

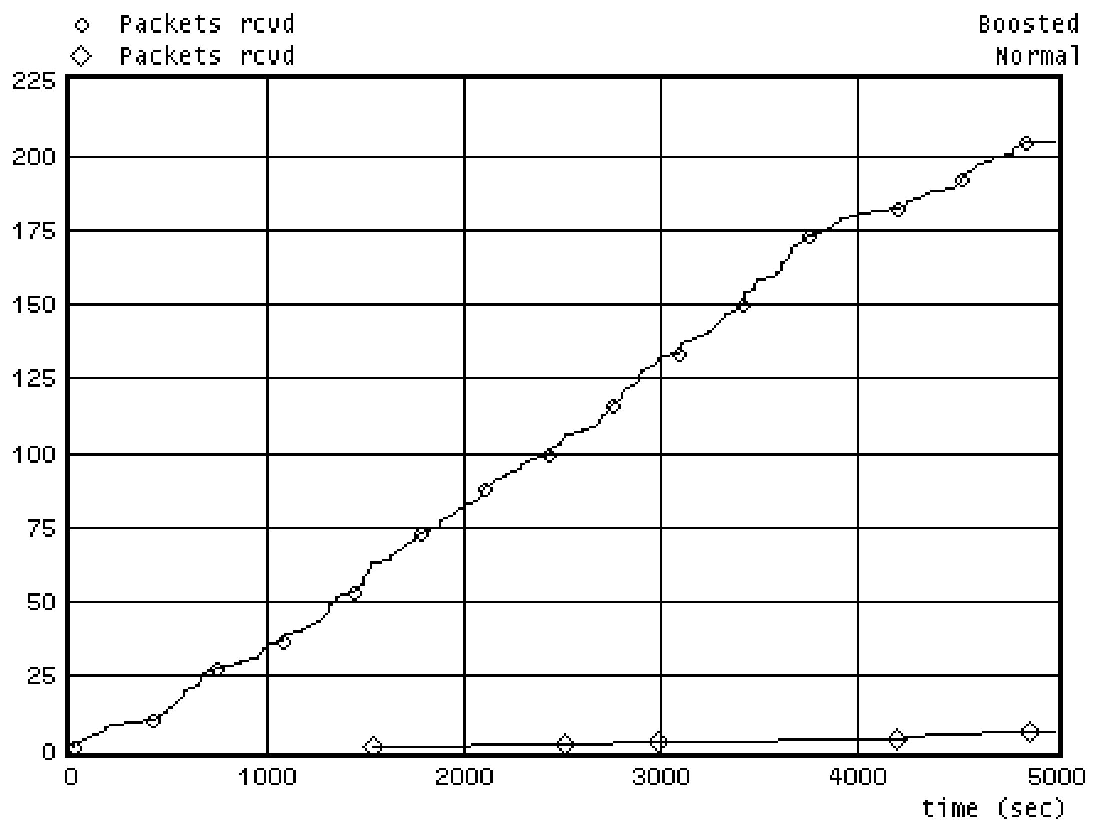


Figure 5.13: Comparison of normal and boosted network at noise level 75000 and packet size 512 bytes with overcode 30%

Chapter 6

Conclusion

Protocol boosters can be viewed as a step towards the fully programmable infrastructure proposed by a number of researchers under the rubric of "Active Networks"[24]. While many of the problems are the same (e.g., robust end-to-end behavior, security for systems into which boosters are loaded, etc.), a key advantage of boosters is that they can easily be injected into today's systems without a wholesale change in the network infrastructure. In that sense, they offer an early test of the promise of active networks.

In this thesis we have presented a novel design for an ACK Reconstruction Protocol Booster to improve the performance of TCP over satellite channels. Unlike link layer methods, our booster preserves the original ACK stream and hence, does not jeopardize the normal functionality of TCP. Unlike protocol termination, the end-to-end properties of TCP are preserved. Also, since it does not convert the protocol, it does not involve too much processing overhead. Through extensive simulations using OPNET we have shown that this booster is a definite boon to TCP communication over satellite links yielding significant improvement in TCP throughput.

Another contribution of this thesis was the modeling and evaluation of the Forward Erazure Correction Booster and its incorporation within the ARL simulation testbed. The Forward

EraZure Correction (FZC) booster is a multi-element protocol booster, which reduces the effective packet loss rate on noisy links such as terrestrial and satellite wireless networks.

While the developers of this booster were able to demonstrate through simple proof-of-concept tests that this booster enhanced performance, there was no reliable model to analyze the performance of this booster within an actual hybrid wireless network. We have attempted to fill this void, by modeling the FZC booster in OPNET and incorporating it within the ARL testbed. Simulation results have proved to be illuminating in that we are now able to determine the degree of usefulness of this booster given the networking environment.

Also , we have provided a detailed description of the protocol booster methodology and showed how protocol boosters can be an effective solution to the problems faced by current networking protocols. We discussed various design issues to be considered when we implement protocol boosters in the operating system. A detailed description of the kernel level implementation in Linux was provided.

There are a number of situations warranting the use of boosters some of which were cited earlier. Other boosters which can be developed in the future include:

Two-Element Encryption Booster

In the case of sensitive data traveling over an insecure subnet, an encryption booster can transparently increase the security of the network services provided. For sensitive data traveling between secure clouds, it may be less expensive to encrypt the data only over the insecure hop thereby reducing CPU cost on the end points. A policy module could detect the insecure hop by IP address or by other means.

One-Element Congestion Control Booster for TCP

Congestion control reduces buffer overflow loss by reducing transmission rate at the source when the network is congested. A TCP transmitter deduces information about network congestion by examining acknowledgments (ACKs) sent by the TCP receiver. If the transmitter sees several ACKs with the same sequence number, then it assumes that network congestion caused a loss of data messages. If congestion is noted in a subnet, then a congestion control booster could artificially produce duplicate ACKs. The TCP receiver would think that data messages have been lost because of congestion and would reduce its window size; thus, reducing the amount of data it injects into the network.

Two-Element Selective ARQ Booster for IP or TCP

For links with significant error rate using a selective ARQ protocol (with selective acknowledgment and selective retransmission) can significantly improve efficiency compared

to using TCP's ARQ (with cumulative acknowledgment and possibly go-back-N retransmission). The two element ARQ booster uses a selective ARQ booster to supplement TCP by : a) caching packets in the upstream booster; b) sending negative acknowledgments when gaps are detected in the downstream booster; c) selectively retransmitting the packets requested in the negative acknowledgments (if they are in the cache).

The protocol booster methodology offers some exciting possibilities for accelerating the evolution of protocols, for changing the economics of protocol development, and for creating useful "hybrid" protocols which have "just enough" support for the heterogeneity actually encountered. Protocol boosters can thus be viewed as an optimistic approach to protocol design. Applications implement application-application protocols assuming an ideal world. Protocol boosters add functions on an as-needed basis to provide this ideal world; when the real world is really ideal (e.g., homogeneous workstations on a LAN) no overhead is incurred.

BIBLIOGRAPHY

1. Mark Allman, Chris Hayes, Hans Kruse, and Shawn Ostermann "TCP Performance Over Satellite Links", Proceedings of the 5th International Conference on Telecommunication Systems, March 1997
2. D.Bakin, W.Marcus, A.McAuley, T.Raleigh, "An FEC Booster for UDP Application over Terrestrial and Satellite Wireless Networks", International Mobile Satellite Conference (IMSC 97), Pasadena, CA, June 1997.
3. A.Bakre and B.Badrinath, "I-TCP: Indirect TCP for Mobile Hosts", proceedings IEEE 15th Annual Conference of Distributed Computer Systems, Vancouver, Canada, May 1995.
4. Hari Balakrishnan, "Challenges to Reliable Data Transport over Heterogeneous wireless Networks", Ph.D. Thesis, U.C.Berkeley, August 1998.
5. Hari Balakrishnan, V.N.Padmanabhan, R.H.Katz, "The Effects of Asymmetry on TCP Performance", Proc. ACM Mobicom, September 1997.
6. L.S. Brakmo, S.W.O'Malley and L.L. Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance" , Proc. ACM SIGCOMM '94, August 1994.
7. D.M.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.
8. K.Claffy, "Internet Traffic Characterization", Ph. D. Thesis, UCSD, 1994.

9. Robert Durst, Gregory Miller, and Eric Travis, “TCP Extensions for Space Communications”, ACM MobiCom, November 1996.
10. D.Feldmeier et al., “Protocol Boosters”, IEEE JSAC, vol.16, no.3, Apr.1998.
11. P.Green, “Computer Network Architectures and Protocols”, New York: Plenum,1982
12. V.Jacobson, “Congestion Avoidance and Control”, Proc. ACM SIGCOMM 88, August 1988.
13. Hans Kruse, “Performance of Common Data Communications Protocols Over Long Delay Links: An Experimental Examination”, Proceedings, 3rd International Conference on Telecommunication Systems Modeling and Design, 1995
14. T.V.Lakshman et al., ”Window-based Error Recovery and Flow Control with a Slow Acknowledgment Channel:A study of TCP/IP Performance”, Proc. Infocom 97,April 1997.
15. A.Mallet, J.D.Chung, J.M.Smith, “Operating System Support for Protocol Boosters”, Technical Report, Distributed Systems Laboratory, University of Pennsylvania.
16. W.S.Marcus, I.Hadzic, A.J.McAuley, J.M.Smith, “Protocol boosters: Applying Programmability to Network Infrastructures”, IEEE Communications Magazine, Oct. 1998.
17. W.S.Marcus, A.J.McAuley and T.Raleigh, “Protocol Booster: A Kernel-Level Implementation”, to be published, Proc. GLOBECOM 98, Sydney, Australia, Nov.1998.
18. A.J.McAuley, “Reliable Broadband Communication Using a Burst Erasure Correcting Code”, Proceedings of ACM SIGCOMM, pp.287-306, September 1990.

19. V.Paxson, "Measurements and Analysis of End-to-End Internet Dynamics", Ph.D Thesis, U.C.Berkeley, May 1997.

20. J.B. Postel, "Transmission Control Protocol", Information Sciences Institute, Marina del Rey, CA, September 1981.

21. J.H.Saltzer, D.P.Reed, D.D.Clark, "End-to-end Arguments in System Design", Proceedings of the 2'nd IEEE International Conference on Distributed Computing Systems, pp.509-512, April 1981.

22. Richard Stevens, "TCP/IP Illustrated Volume I", Addison-Wesley, Reading, MA, Nov.1994.

23. Richard Stevens, "UNIX Network Programming ", Addison-Wesley, Reading, MA, 1992.

24. D.Tennenhouse, et al, "A survey of active networks research", IEEE Communications Magazine, 35(1): 80-86