CHAMELEON - A System for Adaptive QoS Provisioning

BY

Rajesh Krishna Balan

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE,

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2000

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

I would firstly like to thank my supervisor Professor A. L. Ananda for his invaluable guidance and help during the course of preparing this thesis. His patient guidance, valuable suggestions and overall friendly demeanour contributed immensely to the overall preparation of this thesis.

I would also like to thank the members of the TCP Trunk project group (Dr. Jacob, Dr. Seah, Renjish, Chun Meng, Yong Xiang, Lee Lee, Ann Kian) for their assistance and valuable input during the course of preparing this thesis. I would especially like to thank my colleague, Mr. Lee Boon Peng, for his help during the course of this work. It would have been impossible to finish this work without his invaluable help.

I would also like to thank the School of Computing and the National University of Singapore for allowing me to undertake this work and for providing financial assistance during the course of the work.

Finally, I would like to thank my family for putting up with my erratic schedule due to this work and for understanding that I needed to further my studies and not enter the workforce immediately after receiving my honours degree.

SUMMARY

There has been an increase in the number of applications over the Internet that require some form of bandwidth and/or round trip time delay guarantees to be effective. However, the basic Internet Protocol (IP) that is in use today, is unable to provide any sort of Quality of Service (QoS) guarantees beyond best effort. In this thesis, a system that has been developed to provide these guarantees in a simple yet effective manner is described. This system is called Chameleon and it is designed to be totally transparent to the end user and does not require any changes to the existing protocols and hardware in the network. It is designed to be deployed across border routers because these routers usually experience large amounts of congestion as they frequently interconnect networks with very different bandwidths. An experimental testbed was setup and the Chameleon was installed on the border routers in that setup. A series of tests were conducted to determine its effectiveness and they show that the Chameleon can adaptively provision QoS such that the throughput, round trip times and the packet loss percentage for different classes of traffic are substantially improved. The thesis also discusses the use of Chameleon as a reliable yet effective link layer mechanism for use over lossy links.

Keywords : *TCP Tunnels, Quality of Service, Chameleon, Lossy Links, TCP Protocol, Aggregation of Flows*

CHAPTER 1

1 Introduction

There has been a proliferation of multimedia traffic over the Internet in the last few years. More and more applications are using multimedia to enhance their presentations. These enhancements can take the form of pictures, movies and even audio clips embedded into the applications. However these multimedia enhancements tend to be fairly large in size and hence require a fairly large amount of network bandwidth if they are to be streamed over a network effectively. This bandwidth is necessary otherwise the quality of the multimedia application drops drastically. As such, more and more bandwidth on networks is being segregated for use by multimedia traffic and their corresponding applications. These applications include video conferencing, IP Telephony, Real Audio, and Net Casting.

This proliferation of multimedia applications involving voice and video is straining the capacity of the Internet. The evolution and deployment of new network technologies involving the provision of bandwidth has continually fed the insatiable end user's appetite for even more bandwidth. Whenever a leap in network technology for provisioning bandwidth occurs, the end user demands for bandwidth threatens to consume the available capacity and stress the limits of technology. In addition, the increasing use of TCP [22,29,51,53] unfriendly and unresponsive flows compounds the problem of tackling congestion on the Internet.

All these applications require some form of bandwidth and/or round trip time delay guarantees to be effective. However, the basic Internet Protocol (IP) that is in use today, is unable to provide any sort of Quality of Service (QoS) guarantees beyond best effort. That is because IP itself contains no mechanisms to guarantee delivery of packets based on upper layer defined parameters.

As such, there have been great efforts to provide QoS guarantees over IP using protocols like RSVP [27], and architectures like Differentiated Services [26,42]. There are also other methods that use virtual circuits to provide QoS, e.g., ATM and MPLS [12,39].

In this thesis, a complete system that can provide QoS for multiple classes of traffic is described. This solution is meant to be deployed between two border routers and eventually between IP telephony gateways (in this thesis, the words *gateway* and *router* are used interchangeably) as a way of providing QoS guarantees for IP telephony streams. A hybrid approach is employed that uses encapsulation, TCP tunnels, scheduling, and buffer management techniques. This system is called *Chameleon* as it automatically adapts itself to the environmental conditions and it is fully described in Chapter 4. The Chameleon is designed to be deployed at the border routers because that is where most of the congestion and packet drops usually take place. This congestion occurs because the bandwidth between the border routers is usually less than the bandwidth of the networks connecting to the border routers. This difference in bandwidth can sometimes be in the order of several magnitudes (e.g. a gigabit LAN connecting to another gigabit LAN via a kilobit MAN. The Chameleon is to be deployed at the border routers of the MAN in this case). As such, there is a need to optimise the flow of data between border routers and that is what the Chameleon aims to do. By improving the performance of traffic across the gateways, the end-to-end performance will also be improved as a major source of congestion has been removed. The Chameleon can provide QoS guarantees for various classes of traffic entering the Chameleon. It transfers data between the two border routers by encapsulating the data into TCP tunnels as shown in Fig 1.

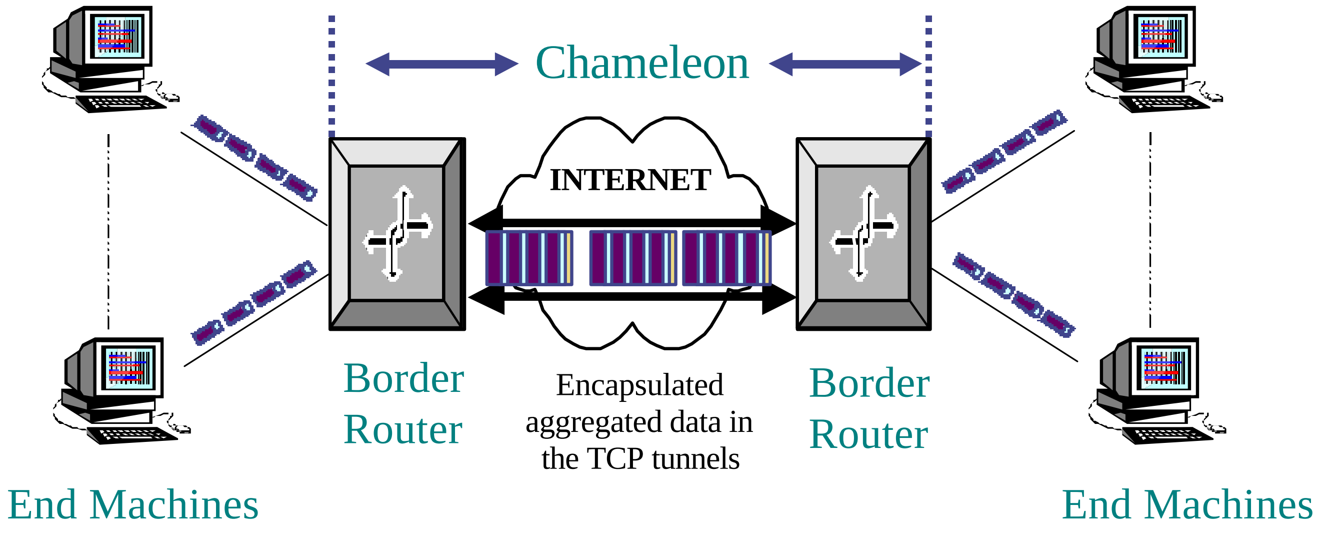


Figure 1 : Logical Representation of the Chameleon

One of the unique features of the Chameleon is that is uses TCP tunnels to encapsulate traffic travelling between the border routers (in this case, the border routers refer to the routers on which the Chameleon has been deployed). TCP tunnels are very similar to IP tunnels. However instead of encapsulating any kind of packet within IP, the Chameleon encapsulates any kind of packet within TCP. This has a number of benefits which are detailed in Chapter 3. However, one of the main benefits is that the TCP tunnels provide a TCP level flow control mechanism for encapsulated traffic which otherwise would not have any form of flow control. This includes UDP and RTP traffic. Another benefit is that TCP tunnels can help small TCP flows from getting unnecessarily killed by larger more aggressive TCP flows. This is because the Chameleon aggregates a number of TCP flows into one big TCP flow within the TCP tunnel. Thus even small TCP flows have now effectively become “large” TCP flows and hence become more resistant to slow starts caused by packet losses due to congestion and / or packet corruption in the network.

Another unique feature of the Chameleon is that it is adaptive. This means that the Chameleon can automatically adjust itself to changes in the traffic distribution on the network. This means that if the Chameleon was deployed to provide a high level of QoS for say, HTTP traffic (with for example a guaranteed minimum bandwidth of x Kbps for the HTTP traffic), if the amount of HTTP traffic entering the Chameleon drops below the minimum guaranteed levels, the excess bandwidth will be given to other classes of traffic. This access bandwidth can be distributed to other classes of traffic in whatever way (be it fair or otherwise) as desired by the administrator of the Chameleon.

The Chameleon can also act as a reliable yet effective data link layer for TCP transmissions over a lossy link. TCP performs very poorly over lossy links as it is unable to differentiate packet losses due to packet corruption from those losses due to congestion. As such, it treats packet corruption in lossy environment similarly to network congestion and thus unnecessarily reduces its sending rate in lossy environments. It is therefore necessary to use a modified version of TCP which is aware of packet corruption over lossy links. Such a version of TCP is HACK [40] which has been developed by Lee and Balan et. al. HACK was used as the TCP mechanism for the TCP tunnels employed by the Chameleon and the whole system was deployed over a lossy link. Experiments were run to determine if using the Chameleon to encapsulate all traffic using the lossy link into the TCP tunnels using the HACK protocol would prove beneficial. The results for this are shown in Chapter 5 Section 7.

1.1 Objectives of work

The main objectives of this work are as follows:

1) To search for and acquire various hardware and / or software tools which can generate application specific traffic which follow TCP dynamics.
2) To verify and evaluate the properties and capabilities of TCP tunnels.
3) To develop a complete system which uses TCP tunnels to provide QoS for various classes of traffic. This system should also be adaptive and thus be able to adjust itself to the traffic conditions on the network.
4) To develop a mechanism by which the system can be used as an efficient yet reliable link layer protocol for transmitting TCP data over lossy links.

1.2 Related Work

In [49], Wang has done a study of the properties of TCP trunks that are similar in nature to the TCP tunnels used in the Chameleon. He studied the concept of TCP trunks and showed some properties of the trunks that are useful for network traffic management. B. P Lee et al studied the properties of TCP tunnels in depth in [6]. The results of that thesis are summarised in Chapter 1 Section 2.

A lot of work has also been done in the area of protocol tunnelling. Most of this work concentrates on tunnelling various protocols within the IP protocol [2,24,55,58]. However, work has also been done to use PPP as the tunnelling protocol [56,57].

There have been a number of initiatives to provide QoS guarantees over IP. These include protocols like RSVP [27], and architectures like Differentiated Services [26,42]. There are also other methods that use virtual circuits to provide QoS, e.g., ATM and MPLS [12,39].

There have also been a number of initiatives to improve TCP's performance over lossy links. A common solution is to add Forward Error Correction (FEC) to the data being sent over lossy links. [28] covers the issues in using FEC to improve the performance of satellite links. The Indirect-TCP (I-TCP) protocol [1] splits a TCP connection between a fixed and mobile host into two separate connections and hides TCP from the lossy link by using a protocol optimised for lossy links. The SNOOP protocol [13] caches packets at the base station and performs local retransmissions over the lossy link.

The use of Explicit Congestion Notification (ECN) [25][43] in the TCP/IP protocol enables routers to inform TCP senders about the onset of congestion and may assist in distinguishing packet losses due to congestion and corruption. Other explicit notification schemes include Explicit Loss Notification (ELN) [14] and Explicit Bad State Notification (EBSN) [7].

1.3 Organisation of Thesis

This thesis is organised as follows. Chapter 2 explains the need for accurate traffic generators and includes a description of various hardware and software traffic generators. This is followed in Chapter 3 with a description of TCP tunnels and their usefulness. Chapter 4 describes the features, design, and implementation of the Chameleon. The experimental results of the Chameleon are shown in Chapter 5 and Chapter 6 summarises this thesis together with a description of future work that can be done in this area.

CHAPTER 2

2 Traffic Generation

This chapter outlines the need for realistic traffic generators and describes some of the available software and hardware traffic generators.

2.1 Need for Realistic Traffic Generators

During experiments, it is usually very important that we recreate a test environment as possible that is as realistic as possible. This means that the test environment should resemble the actual environment (the environment the test is based on) as far as possible. Hence network experiments should accurately reflect the actual operating conditions of the networks the experiments are simulating.

One of the main distinguishing features between different networks is the kind of traffic each network carries. For example, a backbone network would carry a very different mix of traffic as compared with a local area network. It is thus important that network experiments that attempt to simulate these different networks, accurately model the different traffic mixes found on the networks. If the traffic mix used in experiments is wrong, the results of the experiments are useless as they do not reflect any real world scenario and hence have no real merit except from a purely theoretical viewpoint.

However, generating accurate traffic models is very difficult. Many mathematical models have been developed to model different classes of traffic. However, these models tend to be approximations, as the actual traffic mixes generally do not fall nicely into any standard mathematical model. For example, voice traffic is generally modelled using on on-off mathematical model. However, this model still does not provide a 100% guarantee that it is accurately recreating the actual voice traffic found on commodity networks. The only way to be 100% sure that the model is accurate is to use actual traffic traces. I.e., the traffic through a network is captured and then replayed later when needed. However, traces tend to be very large and not very flexible. For example, if your traffic trace only contains 100 HTTP users, it will prove to be difficult to use that trace to simulate 1000 HTTP users. This is one of the reasons why mathematical models are still preferred as they can easily be scaled to simulate a large number of users.

However, it is difficult to find hardware and / or software that generates different kinds of traffic based on these mathematical models. Many of the traffic generators tend to be "stress testers". I.e. they generate a continuous burst of IP traffic. These traffic generators are thus used to test the load capabilities of networks. They are unable to generate specific application level traffic like HTTP etc., which follow TCP dynamics. Hence, these traffic generators are useless for experiments which requires different kinds of user traffic like telnet, HTTP, voice, video etc.

For my experiments, I need to generate various classes of user traffic as the Chameleon is designed to provide QoS for these various kinds of traffic. Hence, it was imperative that I managed to obtain traffic generators that could accurately generate TCP Bulk (ftp), HTTP, voice and video traffic. The remainder of this chapter describes the various hardware and software traffic generators that I tested and gives a short description as to what they are, what they do and how useful they are for my purposes.

2.2 Hardware Traffic Generators

Hardware traffic generators use dedicated hardware devices to generate and analyse the traffic being generated. However, these hardware generators are normally used to stress test production networks and hence they are generally unable to generate application level traffic and they are unable to support the dynamics of TCP. A description of the various hardware based traffic generators is given below with the web site of the company selling the product.

2.2.1 Hewlett Packard Broadband Series Test System

The Hewlett Packard (HP) Broadband Series Test System (BSTS) is a hardware traffic generation and measurement system originally designed for testing ATM networks. However, recently an Ethernet testing module has been added to the BSTS. However, this module is meant for stress testing Ethernet / IP networks and does not really simulate application level traffic like telnet, HTTP, ftp etc. The module also does not support TCP traffic dynamics at all. Thus it can not do the retransmissions and flow control as mandated by the TCP protocol. This product has since been taken over by

Agilent, which was, spun off by HP to take care of all their measurement and testing related services and products.

http://www.agilent.com

2.2.2 Adtech AX/4000 Max 1 Gbps Generator / Analyser

Adtech also develops a number of hardware traffic generators and analysers for use on ATM networks. However, recently they have introduced a 1 Gbps hardware generator and analyser module specifically designed to test IP over Ethernet networks. This module is capable of generating IP level application traffic at 1 Gbps. However, this module does not do any form of TCP testing and thus it is not suitable for our needs.

http://www.adtech-inc.com/

2.2.3 QosNetics QA Robot System

QosNetics has developed a full hardware based traffic generator and analyser specifically designed to test TCP level application traffic. They have specific modules to test HTTP traffic and UDP traffic. However, QosNetics was purchased by HP in 1999 and hence, their products were unavailable on the market when it became necessary to enquire about them. It proved impossible to make any form of direct contact with QosNetics people and the local HP dealers knew nothing about QosNetics products. It was speculated that the QosNetics technology would be integrated into the next revision of the HP BSTS system. The net result was that it was impossible to obtain any vendor-based information about the QosNetics traffic generator.

http://www.qosnetics.com/

2.2.4 GN Nettest interWATCH 95000

GN Nettest has also developed a hardware traffic generator and analyser for use on ATM networks. However, they too have recently added an Ethernet module to their traffic generator. However, this module is not capable of generating application level traffic which follow TCP dynamics and hence it was suitable.

http://www.gnnettest.com/

2.2.5 Netcom Systems SmartBits

Netcom Systems hardware traffic generator and analyser is the SmartBits system. This system is primarily designed to stress test production networks and hence it is unable to generate specific application level traffic, which follow the TCP dynamics. As such, it was not very suitable.

http://www.netcomsystems.com/

2.2.6 Wendell and Goltermann Domino Gigabit Internetwork Analyser

Wendell and Goltermann (WG) is the regional distributor for the Domino series of hardware traffic generators and analysers. However this system is primarily designed as a traffic analysis system and hence, it's traffic generation capabilities are very simple. It is also unable to generate application level traffic, which follow TCP dynamics.

http://www.wg.com/

2.3 Software Traffic Generators

Since the hardware based traffic generators and analysers proved to be unsuitable for my needs, a search began for software based traffic generators which could prove useful. All of these traffic generators are freely available on the web and their web sites are given. They all support application level traffic and they all fully support the TCP protocol.

2.3.1 TTCP

Ttcp [33] was developed by M. Muuss and T. Slattery at the US Army Ballistics Research Lab in Nov. 1985. It was further developed by Silicon Graphics Incorporated (SGI) in Oct. 1991. Ttcp is able to generate TCP and UDP bulk traffic streams. It operates using a client / server mechanism. I.e., to run ttcp, you need to have a server and a client. The same program can act as both (with different command line parameters). The usefulness of ttcp is that it shows the throughput of the connection on both the client and server sides. However, it does not display any instantaneous throughput values, but rather it only displays the throughput after all the data has been transmitted.

ftp://ftp.sgi.com/sgi/src/ttcp/

2.3.2 IPERF

Iperf [30] was designed to be an updated version of ttcp. In that respect, it has succeeded as iperf has a number of features that make it better than ttcp. Like ttcp, iperf also uses the client / server paradigm with the same program being able to act as both the client and the server. However iperf has many new features which are not found in ttcp. The main changes are that iperf can display instantaneous throughput values at intervals specified by the user (lowest value is every 1 second) and that iperf has support to display jitter calculations for UDP transfers. This feature is very useful for experiments involving multimedia traffic, as jitter is one of the metric that are usually required to be measured.

http://dast.nlanr.net/Projects/Iperf/

2.3.3 SURGE

Surge [35] was created by Paul Barford and Mark Crovella at the Boston University. It is a series of tools which attempts to realistically model HTTP requests. Surge consists of many tools which provide as their end product a script containing the access patterns to a set of files. This script will be created based on a number of parameters supplied by the user such as the arrival rate of HTTP requests, the size distribution of files requested etc. Surge will create all the necessary data files being accessed by the script. The user has to then transfer those files onto a real web server. Surge provides a network client which will read the script containing the access patterns. The client will then access the files on the web server based on the information in the script. Surge is thus very accurate in modelling HTTP traffic as it uses a real web server to respond to HTTP requests. Surge can also model HTTP 1.0 and HTTP 1.1 requests and access patterns.

http://cs-people.bu.edu/barford/

2.3.4 IPB

IPB [5] was developed by Bruce Mah at the University of California at Berkeley (now at the Sandia National Laboratories). It is also a HTTP traffic generator. However, unlike SURGE, IPB provides both the client and the server components of the traffic generator. SURGE, in comparison, only provides the client and requires a real web server to act as the server. The client and server programs for IPB are different.

2.3.5 AB

AB [3], also known as the Apache benchmark, is supplied together with the Apache [3] web server. This benchmark generates HTTP traffic and is used to stress test Apache web servers.

http://www.apache.org/

2.3.6 TCPLIB

TCPLIB [36,37] is a set of library functions developed by Sugih Jamin and Peter Danzig at the University of Southern California in 1991. They analysed the traffic on a Wide Area Network (WAN) and produced a traffic distribution model for various classes of traffic. The library functions access that distribution. Since this library is based on actual network measurements, it is very accurate for the classes of traffic it models. However, the measurements were taken before the advent of the World Wide Web (WWW) and hence it may not accurately reflect the traffic distributions found on modern day networks.

http://irl.eecs.umich.edu/jamin/papers/tcplib/

2.3.7 TG

TG is a program initially written by Rusty Eddie from ISI. This program uses the TCPLIB functions and distributions to generate traffic and thus serves as a front end to the TCPLIB library. However, there were many bugs in the original program and as a result, TG was almost completely re-written by Lee Boon Peng. TG can simulate voice and video UDP streams but the distribution is not very accurate. Hence, the voice and video traffic generations routines should not be used when it is necessary to have an accurate voice or video stream. Since TG used the distributions of TCPLIB, it's other traffic generation routines (those that use TCPLIB) are very accurate.

http://www.isi.edu/~eddy/tg/tg.html for original version

Heavily modified internal version is currently unavailable for public consumption.

2.3.8 NETPERF

NETPERF is a benchmarking developed by Rick Jones. It can be used to measure the performance of TCP and UDP bulk streams.

http://www.netperf.org/netperf/NetperfPage.html

2.3.9 NETSPEC

NETSPEC [19] was developed at the University of Kansas. It is touted as a completely distributed traffic generation system that is able to generate multiple classes of traffic. However, the program was unable to compile and run properly at the time when it was tested and was thus useless.

http://www.ittc.ukans.edu/netspec/

ftp://ftp.ittc.ukans.edu/pub/netspec/archive-user-26063/

2.3.10 rtpplay and rtpdump

Rtpdump and Rtpplay are tools developed by Hennig Schulzrinne at Columbia University. These tools allow a user to replay traffic traffic of RTP [15] voice and video samples. As such, they provide a very accurate traffic generator for RTP video and voice sources. However, they require actual RTP voice and video traces in order to generate the traffic. There are a number of such traces on the Internet but they tend to be very large. However, this is offset by the fact that the traffic being generated is very realistic.

http://www.cs.columbia.edu/~hgs/software/

2.3 Summary

The hardware traffic generators were all found to be unsuitable for my needs. However, a number of software traffic generators were found to be suitable. Table 1 summarises the main types of traffic which can be generated by the various software traffic generators.

	Ttcp	**Iperf**	**Surge**	**IPB**	**AB**	**Tcplib**	**TG**	**Netperf**	**Netspec**	**Rtpplay**
Telnet	NO	NO	NO	NO	NO	YES	YES	NO	YES	NO
ftp *(1)* **(TCP bulk)**	YES	YES	NO	NO	NO	NO	NO	YES	YES	NO
ftp *(2)* **(realistic)**	NO	NO	NO	NO	NO	YES	YES	NO	NO	NO
HTTP	NO	NO	YES	YES	YES	NO	NO	NO	YES	NO
UDP voice *(3)* **(UDP bulk)**	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO
UDP video *(4)* **(UDP bulk)**	NO	NO	NO	NO	NO	NO	YES	NO	NO	NO
RTP voice *(5)* **(realistic)**	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES
RTP video *(6)* **(realistic)**	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES
Works properly?	YES	YES	YES	YES	YES	Only a library *(7)*	NO *(8)*	YES	NO *(9)*	YES

Table 1 : Traffic Generated by Different Software Traffic Generators

Notes :

1) TCP Bulk traffic refers to a continuous stream of TCP traffic sent at the maximum possible rate with the largest possible MTU.
2) ftp realistic traffic includes modelling of the ftp protocol including the connection establishment, the sending and receiving of various ftp commands like *get*, *dir* etc., and the actual data transfer itself.
3) UDP bulk voice traffic refers to a continuous stream of UDP traffic with a MTU of 384.
4) UDP bulk video traffic refers to a continuous stream of UDP traffic with a MTU of 1500.
5) RTP voice traffic refers to realistic voice traffic generated using a traffic trace.
6) RTP video traffic refers to realistic video traffic generated using a traffic trace.
7) TCPLIB is only a library and hence requires the user to write a program to access the library.
8) TG did not work properly and required extensive in-house modifications before it was usable.
9) I was unable to compile and run NETSPEC properly. A new version is available on the web page which I have not tested.

CHAPTER 3

3 TCP Tunnels

This chapter gives an overview of TCP tunnels and their usefulness. A TCP tunnel is a TCP circuit which carries IP frames over the Internet. By layering user flows over this construct, we hope to benefit from the congestion control mechanism of TCP/IP, especially in segregating unresponsive flows from TCP-friendly flows. Kung & Wang [16,49] did related work on the properties of TCP trunks which are similar to TCP tunnels. In their paper, they demonstrate how TCP trunking protects web users from competing ftp traffic. They also examine how TCP trunking may be used to allow a site to control its offered load into a backbone network so that the site can assure some QoS for its packets over the backbone.

The concept of tunnelling is not new. A lot of work has been done on the use of IP as a tunnelling protocol to carry many other protocols over IP networks. The PPP protocol also uses the concept of tunnelling to carry IP and other protocols over point-to-point links. Tunnelling is used to provide the benefits of one kind of network protocol to traffic using another network protocol. For example, IP tunnelling is used to provide an easy way for IPX traffic (used by the Novell Netware suite of products) to traverse the Internet (which is predominantly IP based). However, the use of TCP as the tunnelling protocol is relatively new and introduces a few new properties, which are not present when other protocols like IP and / or PPP as used as the tunnelling mechanism. Some of these interesting properties are outlined below. More detailed descriptions,

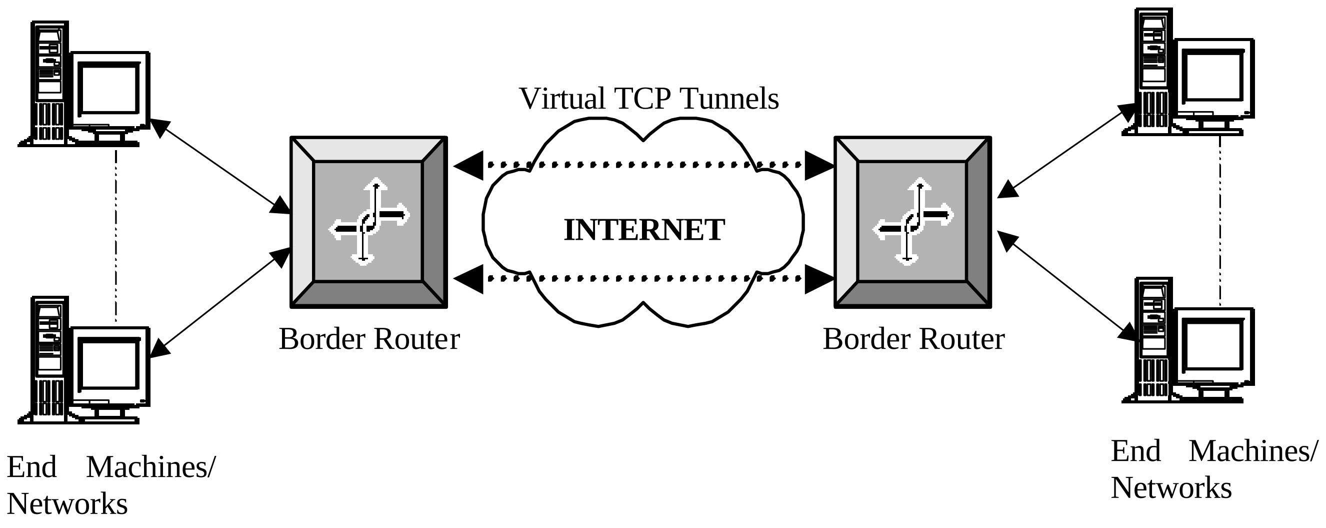


Figure 2 : TCP Tunnels

experimental analysis and verifications of some of these properties are found in [6]. Some of the more relevant results from that paper are reported here in abridged form. Fig. 2 shows how TCP tunnels can be used at border routers to interconnect different networks.

3.1 Protection of TCP bulk flows from unresponsive UDP flows

Tunnels were used to separate TCP bulk flows from unresponsive UDP flows over a congested link. It was shown that without tunnels, the UDP flows consumed most of the bandwidth available over the congested link. As a result, the TCP bulk flows were being starved by the unresponsive UDP flows. By placing TCP tunnels across the congested link, the proportion of bandwidth consumed by the TCP bulk flows improved. Table 2 shows some of the results mentioned in [6].

Per TCP flow statistics	Without TCP Tunnel	With TCP Tunnel	% Improvement
Retransmitted bytes	385783.90	57773.36	(+) 567.75
Retransmitted packets	266.78	39.93	(+) 568.12
Retransmitted data	28.10	6.10	(+) 360.51
Data segments sent	624.45	397.47	(+) 57.11
Acknowledgement	324.27	247.32	(+) 31.11
Data segments over	65.82	61.66	(+) 6.74
Total Segments	948.72	644.79	(+) 47.14
Average throughput	1.18	2.85	(+) 141.53

Table 2 : Protection of TCP bulk flows from unresponsive UDP flows

3.2 Protection of Interactive Traffic over Congested Links

TCP Tunnels can also protect interactive traffic flows (HTTP, telnet etc.) from other more aggressive flows over congested links. Table 3 shows the results of using TCP tunnels to protect interactive traffic (in this case, HTTP traffic).

Per http flow statistics	Without TCP Tunnels	With TCP tunnels
Total Packets transferred	27.75	55.48
Forward/Reverse Aggregated Throughput (Bps)	1260 / 64247	1582 / 126325
Successful / Total http connections made	45 / 208 (21.6%)	290 / 624 (46.5%)

Table 3 : Protection of Interactive Traffic over Congested Links

3.3 Protection of “TCP Friendly” Flows

Interactive traffic flows and TCP bulk flows are both classified as “TCP Friendly” flows. All TCP-friendly flows can co-exist very nicely inside the TCP tunnels. They can share bandwidth in a fair way and larger TCP flows do not kill smaller TCP flows unnecessarily. This is shown in [6] and the results have been summarised in Chapter 3 Sections 1 and 2.

3.4 Protection from Fragmentation

Normally, data traversing a lower MTU link would get fragmented and remain fragmented for the rest of its journey. Placing TCP tunnels across the lower MTU link can eliminate this, as any fragmentation caused by the lower MTU link will be confined within the TCP tunnels.

This is because a TCP tunnel behaves like a level 2 (datalink) circuit and hides the different MTUs of the underlying networks and links it spans over the Internet. The user flows (in the form of IP packets which is the tunnel’s payload) are oblivious to these different MTUs and hence are free to use its own value, presumably the largest (limited by the sender’s interface’s MTU setting) to maximise throughput. If the use of larger MTUs is permitted, each packet carries less header overhead and this allows TCP connections to ramp up their congestion windows faster.

The results of using TCP tunnels to overcome lower throughput due to fragmentation are shown in Table 4.

Path MTU (T1-T2) (bytes)	**Mean Throughput of 10 runs (Mbps) No TCP Tunnel**	**Mean Throughput of 10 runs (Mbps)**
1500	7.40	7.40 (+0%)
1006	7.10	7.40 (+4%)
576	6.50	7.08 (+9%)
296	5.00	6.70 (+34%)

Table 4 : Protection from Fragmentation

3.5 Aggregation of flows

The tunnels aggregate multiple flows into one or more flows and this has been shown to be effective by Morris [41]. The aggregation of flows also contributes to the effectiveness of the TCP tunnels in providing the benefits listed below.

3.6 Shortening of Router Queue Lengths

Tail-Drop has been adopted widely on routers as the default packet-dropping policy. However, this policy causes many TCP flows to throttle their sending rates at the same time, when the queue overflows. For this reason, Random Early Discard (RED) [45,47] was designed to penalise TCP-friendly flows which consume more than their fair share of bandwidth by randomly dropping packets from the flows when the average queue size exceeds the minimum threshold. However, RED has no effect on UDP or unresponsive flows which do not respond to packet drops.

TCP tunnels improve the situation as they aggregate all flows, whether they are TCP, UDP, or something else, into TCP tunnels. Hence, the RED algorithm running on the routers in the congested link only see the TCP packets of the TCP tunnels. RED is very effective in handling buffer management and packet drops in congested links if it is only dealing with responsive TCP flows. As a result, placing TCP tunnels over congested links served by RED queues helps in reducing the overall queue length on those RED queues. This is shown in Fig 3.

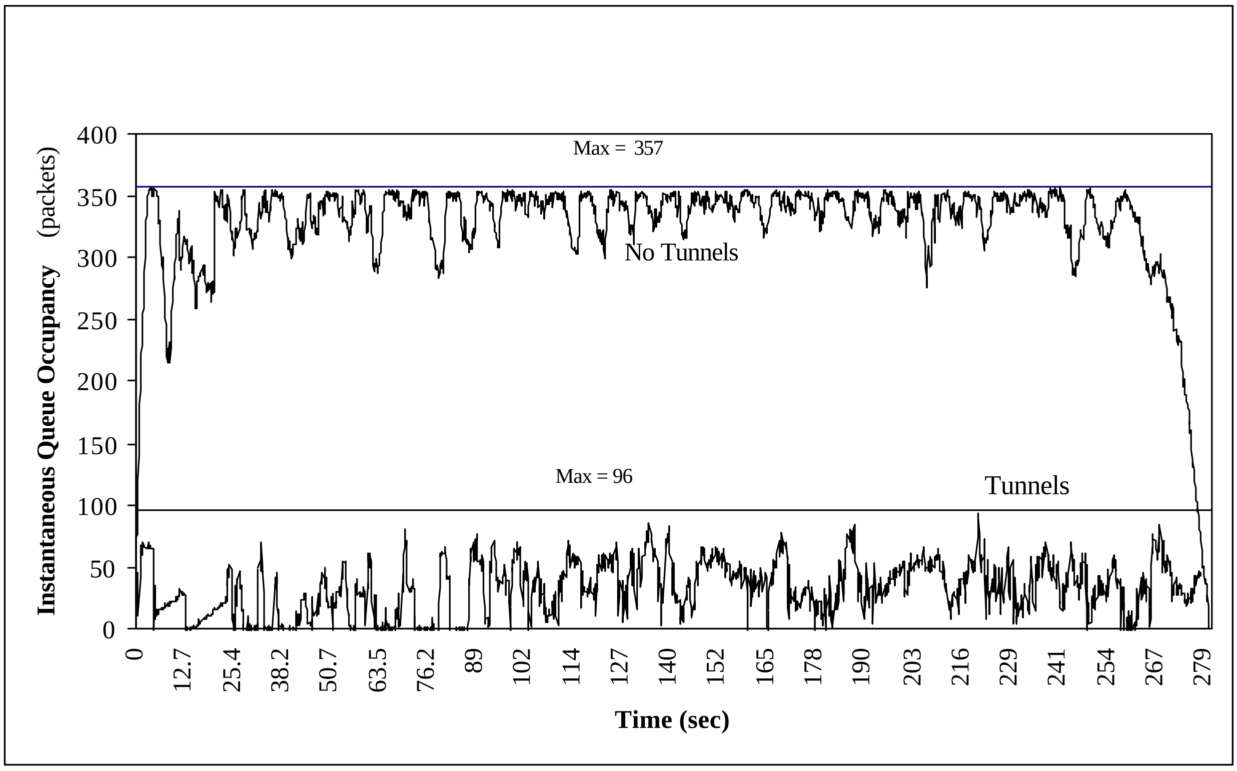


Figure 3 : Instantaneous Queue occupancies for core router with and without TCP tunnels

3.7 Back-Pressure Effects of TCP Tunnels

[44,45] discuss the problems of unresponsive flows and the danger of congestion collapse of the Internet while [11] discusses the use of back pressure in moving congestion away from the core. The rates of UDP flows are not limited by congestion control and such aggressive flows only starve TCP-friendly flows of their rightful share of bandwidth. In addition, UDP flows typically exacerbate congestion problems and waste precious bandwidth when their packets are dropped during the onset of congestion.

TCP tunnels provide a possible solution out of this dilemma. Tunnels could be used to wrap UDP flows at edge routers. The tunnels carry the UDP flows over the Internet and

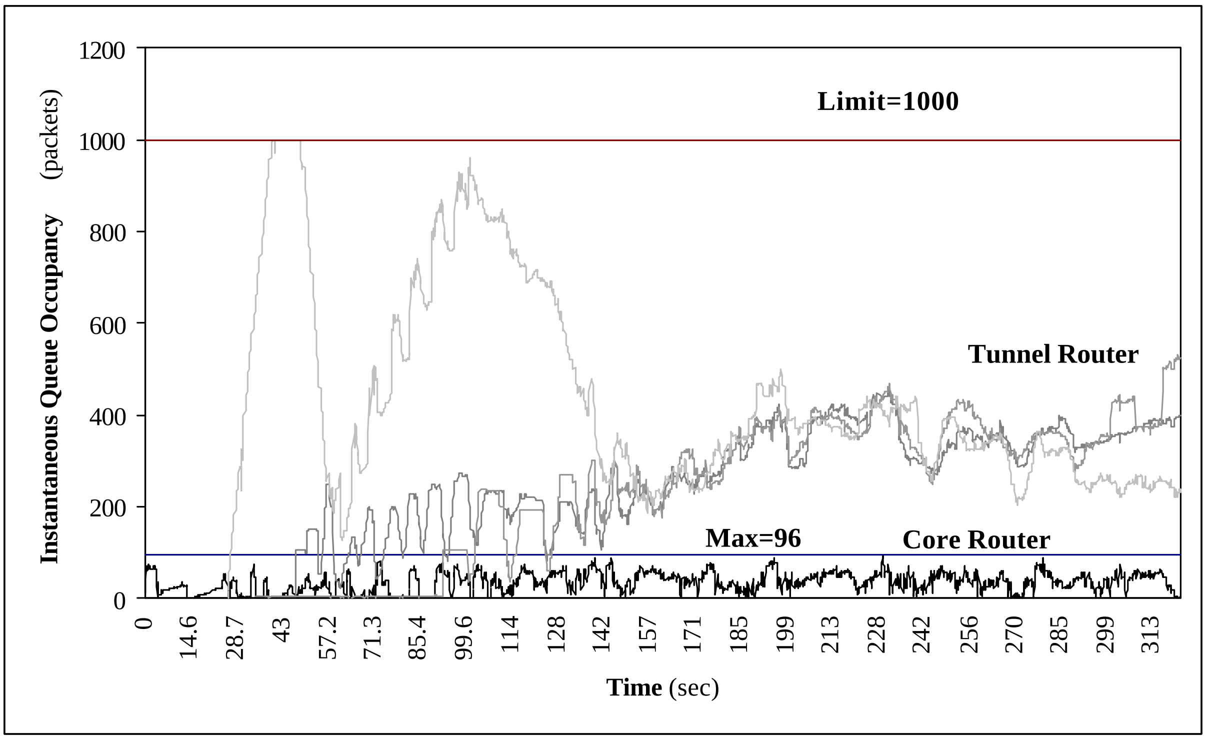


Figure 4 : Instantaneous Queue occupancies for core and tunnel routers using TCP tunnels

are then subjected to the ever-changing conditions of the network. When congestion occurs, the packet drops force the tunnels to throttle their sending rates. The advantage here is that the UDP flows are now TCP-friendly (so to speak) and sensitive to packet drops from the perspective of the core routers. This is of particular relevance when RED is deployed on routers as the default packet dropping policy. In short, packet losses in the congested core routers are now pushed to edge routers. As such, congestion is moved away from the core to the edge of the backbone when tunnels are used, i.e., congestion and packet drops occur at the border routers across which the tunnels are deployed and not within the core. This is shown in Fig. 4 and Table 5.

Number TCP flows in the core	**Without Tunnels**	**With Tunnels**
60	3.12%	0.15%
120	7.27%	0.14%
180	11.80%	0.15%
240	19.41%	0.16%

Table 5: Percentage of dropped packets in the core router

3.8 Avoiding Congestion Collapse

[44] discusses at length the forms and causes of congestion collapse. TCP tunnels can help to minimise and / or prevent congestion collapse of the following forms:

- Classical congestion collapse [21]
- Congestion collapse from undelivered packets

- Fragmentation-based congestion collapse
- Congestion collapse from increased control traffic

Please refer to [6] for a complete discussion on how TCP tunnels can help prevent congestion collapse of the forms listed above.

3.9 Acting as a reliable data link layer

Wireless and / or lossy links are becoming increasingly common in the Internet these days. These links are characterised by high bit error rates compared with traditional wired links. Hence, it is imperative that all network protocols run over those links have some sort of reliability built in to overcome the errors in transmission introduced by the wireless medium.

TCP tunnels can act as a reliable data link layer by placing the tunnels across the end-points of the lossy link. The TCP tunnels effectively act as a reliable layer 2 mechanism by making the tunnels use forms of TCP which are able to overcome errors caused by the wireless medium. These could include protocols like Snoop, I-TCP etc., or even other error detection and / or correction mechanisms like ELN and FEC etc. The tunnels can also use the HACK [40] variant of TCP, which is specially optimised for use in lossy environments.

CHAPTER 4

4 The Chameleon System

Fig. 1 in Chapter 1 shows the logical representation of the Chameleon. This system was designed to provide an easy way for network administrators to regulate the use of their border routers. The Chameleon is meant to be deployed between two border routers. On deployment, the Chameleon establishes a number of TCP connections between the two border routers. These connections serve as the TCP tunnels into which all traffic entering the Chameleon is encapsulated. A series of scripts, tools, and programs are also provided which allow the network administrator to allocate different QoS guarantees to different classes of traffic. The Chameleon provides these QoS guarantees using a two pronged approach. Firstly, the use of a traffic management mechanism which does the admission control, traffic filtering, and shaping of the incoming traffic flows and secondly, the use of TCP tunnels connecting the two end points of the Chameleon to carry the shaped traffic. The reason for using TCP tunnels is because of the various good properties they have, as shown in Chapter 3. The detailed flow path of data within the Chameleon will now be explained.

Data entering the Chameleon passes through two subsystems before entering the TCP tunnels. The first subsystem deals with the admission control and traffic management mechanisms while the second subsystem deals with the TCP tunnelling and packet encapsulation mechanisms. The Chameleon also has a third subsystem, which is at the exit of the TCP tunnel and deals with the decapsulation mechanisms necessary for handling the output data streams of the first two subsystems.

Various classes of traffic enter the Chameleon where they are segregated and bandwidth is provisioned for these classes of traffic based on some pre-arranged traffic contract (e.g. UDP Video is to get at least 5% of the bandwidth but no more than 20% of the bandwidth). The administrator of the border routers sets up these contracts and it is the administrators' responsibility to ensure that all the contracts are satisfiable (i.e. the sum of all the bandwidth guaranteed to the various traffic classes does not exceed the expected minimum bandwidth of the network between the two border routers). Based on these contracts, the Chameleon encapsulates data packets into different TCP tunnels and sends them to the border router at the other end of the tunnel where they are decapsulated and sent on their way to their destination.

The Chameleon is also designed to be adaptive to network conditions, i.e., when the data rate of one class of traffic decreases, the Chameleon is supposed to automatically allocate the excess bandwidth to the other classes in a fair manner. However, all classes should still be guaranteed any minimum bandwidth specified in their contracts. But, if there is insufficient traffic for a particular class to satisfy its minimum bandwidth requirements, that excess bandwidth should automatically be allocated to other classes that could use the excess bandwidth. The Chameleon is designed to provide this adaptive property seamlessly.

This system has the benefit of being completely transparent to end-users and requires no changes in networking code or hardware for any end user. The only change is the additional software installed on the two border routers. The Chameleon can even be implemented by placing a low-end commodity unix machine in front of the actual border router hardware avoiding changes to the actual border router itself.

The logical diagram of the Chameleon is shown in Fig. 5. The control mechanisms for the TCP tunnels are implemented as a user space program. TCP connections are created at the user level for the various classes of traffic. Each class of traffic will have one or more TCP tunnels assigned to them. These tunnels are effectively virtual circuits that form a point-to-point connection between the two border routers using the Chameleon. Data entering one end of the tunnel is encapsulated into the TCP connection and will be sent to the other end of the tunnel where the data packet is decapsulated and routed to its destination via normal routing algorithms.

TCP tunnels are used because of all the benefits they provide as mentioned in Chapter 2. Using tunnels does add an additional overhead caused by the additional TCP/IP headers for the encapsulated packet. However, these additional overheads are justifiable in the face of the benefits that tunnels provide.

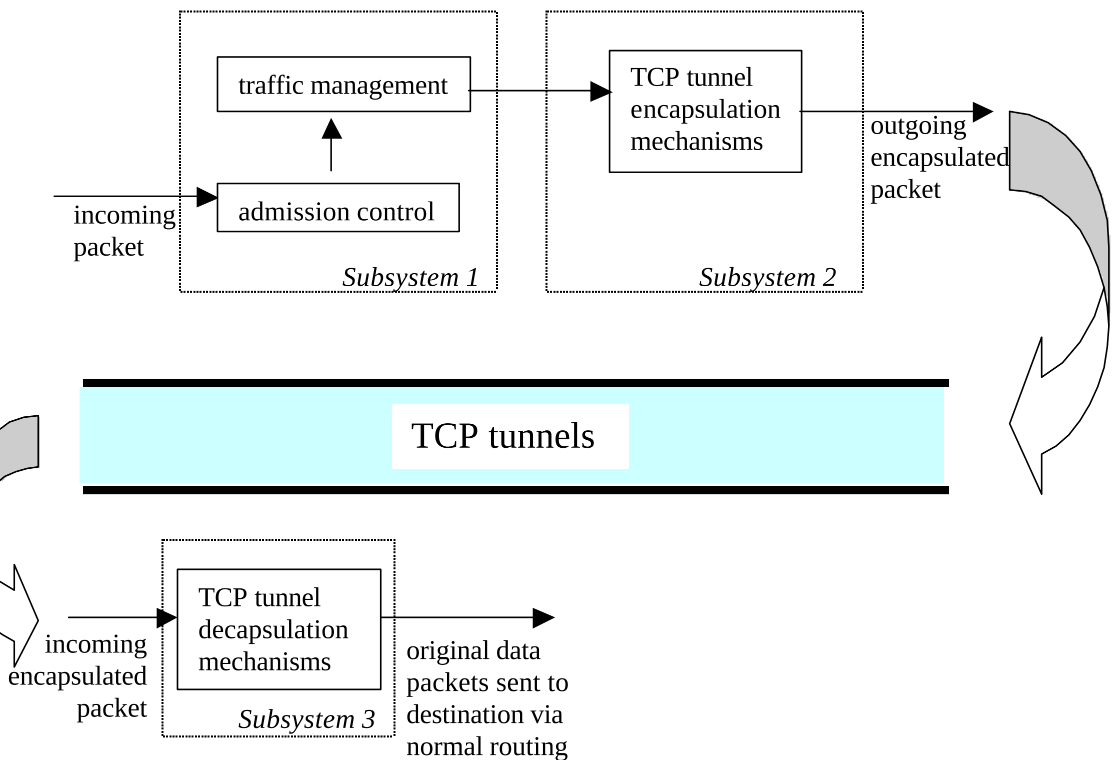

traffic management
admission control
incoming packet
Subsystem 1
TCP tunnel encapsulation mechanisms
Subsystem 2
outgoing encapsulated packet
TCP tunnels
incoming encapsulated packet
TCP tunnel decapsulation mechanisms
Subsystem 3
original data packets sent to destination via normal routing

Figure 5 : Logical Diagram of Chameleon

4.1 Admission Control

Admission control mechanisms are necessary to ensure that the traffic contracts assigned to various classes of traffic can be enforced. Without admission control, there is a risk that too much non-contracted data could enter the Chameleon, which would make traffic management very difficult.

For the experiments, a static admission control mechanism was used. In this method, specific flows were manually added into the system before the start of the experiments. However, in a real environment, any other form of admission control can be used and integrated with the system without any difficulties. A number of admission control algorithms are detailed in [34].

4.2 Traffic Management

The logical diagram of the traffic management mechanism used in the Chameleon is shown in Fig. 6. Data packets entering the traffic management mechanism first go into the packet filter. Based on the class of traffic, the packet filter diverts the packet to the appropriate buffers in the buffering and packet dropping module. Different classes of traffic will be assigned to different buffers in the buffering and packet dropping module and the appropriate packet dropping algorithms are used for each class (e.g., RED is used for classes where tail-drop is not appropriate etc.). Before data packets leave the traffic management mechanism, they go into a rate control module which ensures that all the minimum and/or maximum guarantees for each class of traffic are satisfied according to their traffic contracts. The traffic management mechanism uses the traffic control extensions (TC) in the Linux kernel [52]. TC is used to provide RED and FIFO queues for the various traffic classes used in the experiments. TC also was used to provide a CBQ scheduling mechanism for the various classes of traffic.

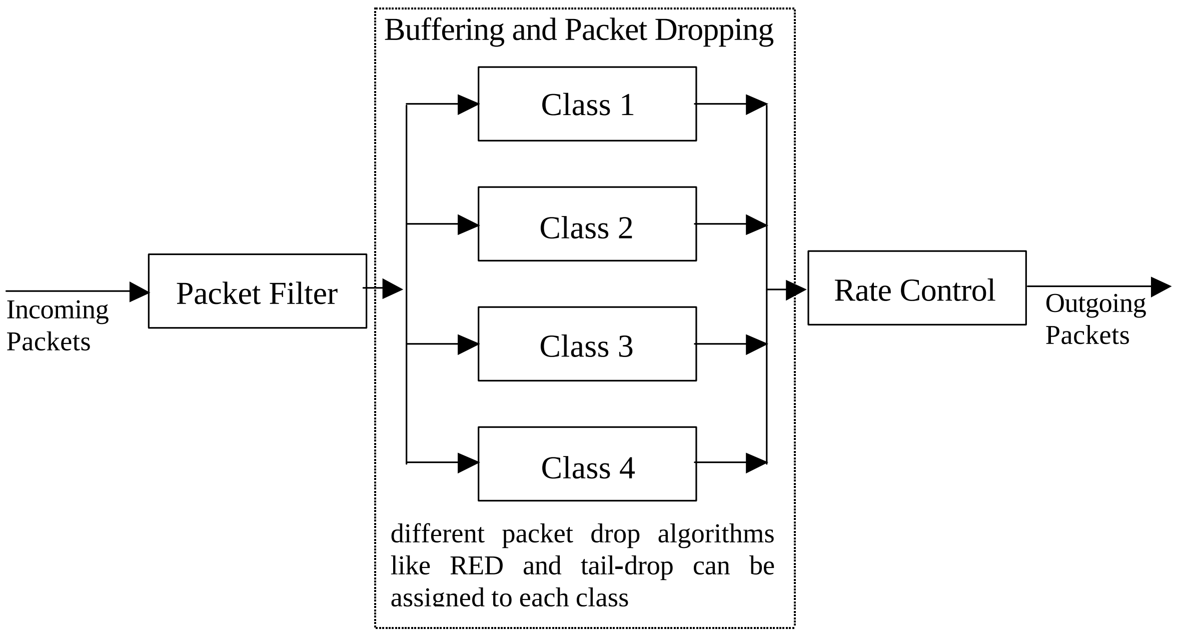

Buffering and Packet Dropping
Class 1
Class 2
Class 3
Class 4
Incoming Packets
Packet Filter
Rate Control
Outgoing Packets
different packet drop algorithms like RED and tail-drop can be assigned to each class

Figure 6 : Logical Diagram of the Traffic Management Mechanism

4.2.1 Adaptive Nature of Chameleon

The adaptive nature of the Chameleon is provided by the traffic management mechanisms. This mechanism enforces traffic contracts for the various classes of traffic. When a particular class of traffic has insufficient incoming traffic to fully utilise it's traffic contract, the access bandwidth is distributed to the other classes of traffic as stipulated by the administrator of the Chameleon. The rules for sharing any access bandwidth can be either fair or unfair. This is entirely up to the administrator of the Chameleon. This adaptive property is made possible through the use of proper packet marking and flexible scheduling mechanisms like CBQ.

4.3 System Implementation

The Chameleon comes in two parts; the traffic management mechanism and the TCP tunnels mechanism. In this section, the specific implementation details relevant to each part are detailed.

4.3.1 Traffic Management Mechanism

The logical diagram of the traffic management mechanism is shown in Fig 6. The implementation details of the traffic management mechanism are as follows:

1) The QoS routines are compiled into the Linux kernel. This is done by compiling the kernel and enabling all the options under the "Network Options -> QoS" menu.
2) This enables the various QoS providing buffer and scheduling mechanisms (like RED, CBQ etc.) in the Linux kernel.
3) To use these various mechanisms in the Linux kernel, a user level program is required. This program is known as TC and is found in the iproute2 package. This package can be found at ftp://ftp.sunet.se/pub/Linux/ip-routing/
4) Activating the various buffer and scheduling mechanisms requires different parameters to be passed into the program TC. For example, to enable a simple

FIFO queue of queue length 100000 bytes requires the command

“tc qdisc add dev eth1 root bfifo limit 100000”

5) The user is advised to look at the various examples that come together with the iproute2 package for information on how to enable the more complicated queues like RED and CBQ.

Sample scripts used in the Chameleon are provided in Appendix 1.

4.3.2 TCP Tunnels Mechanisms

The concept of TCP tunnels is very simple. What happens is that traffic passing through a machine gets placed into another TCP connection. I.e., it gets encapsulated into TCP (and the corresponding IP). Fig. 7 shows the encapsulation that happens in a TCP tunnel.

Encapsulating packets into TCP tunnels requires two steps .The first step is to retrieve the packet from the incoming interface and the second is to place the packet the TCP tunnel connection on the outgoing interface.

The implementation of the TCP tunnels being used is very simple. The TCP tunnels are being run as a completely user space program. This is very inefficient as incoming data packets have to be copied from the kernel space into the user space before they can be

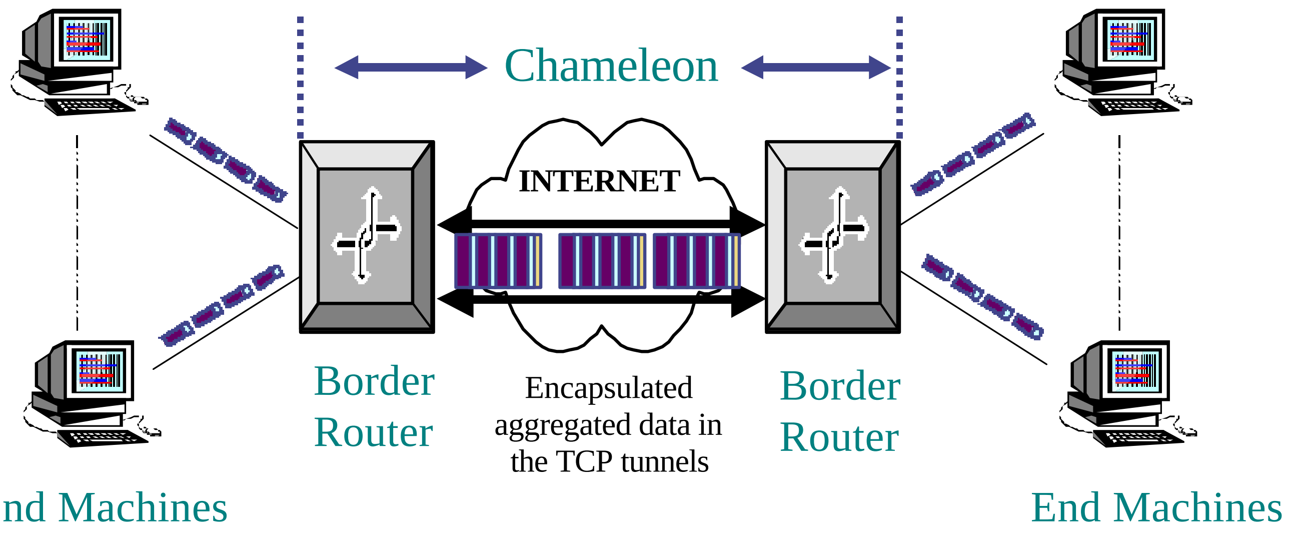


Figure 7 : Aggregation of Incoming Data into TCP Tunnels by Chameleon

processed by the user space program. This sets a hard limit on the maximum rate at which packets are arriving on the incoming interface. If packets arrive faster than the maximum rate, the user space program will need to be able to transfer the packets from the kernel space to the user space fast enough and as a results, incoming data packets will be lost. Experiments have shown that the user space program can support a maximum incoming data rate of about 15 Mbps. Hence, running experiments at 10 Mbps will ensure that no packets are lost by the TCP tunnel software.

The TCP tunnel software is fully written in C using the standard socket programming interface [54] and is split into 3 parts. The 1st part is responsible for removing packets from the incoming interface and copying them from kernel space to user space. To facilitate the extraction of packets from the incoming interface, the Berkeley libpcap library [59] is used. This library allows a user space program to receive a copy of any incoming ethernet packet which satisfies certain user defined filtering requirements. However, libpcap only sends a copy of the incoming packet to the user space program. The original packet is still in the kernel space. Hence, to remove this packet from the kernel space, it is necessary to turn off IP forwarding. Another way to remove this packet is to install a firewall (ipchains [60], iptables [60] etc.) in the system and actively discard all packets which are to be sent to the user space. Once the packet is in user space, the TCP tunnel software places this packet into a user space linked list.

The 2nd part of the TCP tunnel software removes packets from the linked list and places them into the TCP tunnel itself. The TCP tunnel is simply a user space TCP connection opened when the TCP tunnel software is first run. Packets in the linked list are removed from the list and sent out over a TCP tunnel via standard socket write () calls (with the socket id of the previously opened TCP tunnel). Which TCP tunnel is selected to carry the packet depends on the individual traffic management rules for that kind of packet. For e.g., all UDP packet may only be sent over TCP tunnel number 3. To ensure that the receiver can recover the original packets from the encapsulated data stream, the length of each packet is added to the front of the packet before it is written into the TCP tunnel socket.

As can be seen, the 1st and 2nd parts of the TCP tunnel software need to be run concurrently. As such, they are implemented as separate threads within the same program with mutal exclusion locks used to ensure proper access of the shared linked list data structure. Parts 1 and 2 of the TCP tunnel software make up the sender portion of the TCP tunnel software.

The 3rd and final part of the TCP tunnel software is the receiver. This part is responsible for extracting the original packets from the encapsulated data stream in the TCP tunnel, and then sending the original packets towards their destination. To this end, the receiver does the following:

1) Read the length of the next packet in the TCP tunnel data stream. This length is always the first byte of any packet.
2) Read the packet from the data stream using the length previously read.
3) Extract the IP destination address from the retrieved packet.
4) Use this address to forward the packet towards it's final destination. This is done by writing the packet into a raw socket with the IP destination address as one of the arguments.

Fig 8. shows the various components of the TCP tunnel software. The code for the TCP tunnel software is provided in Appendix 2.

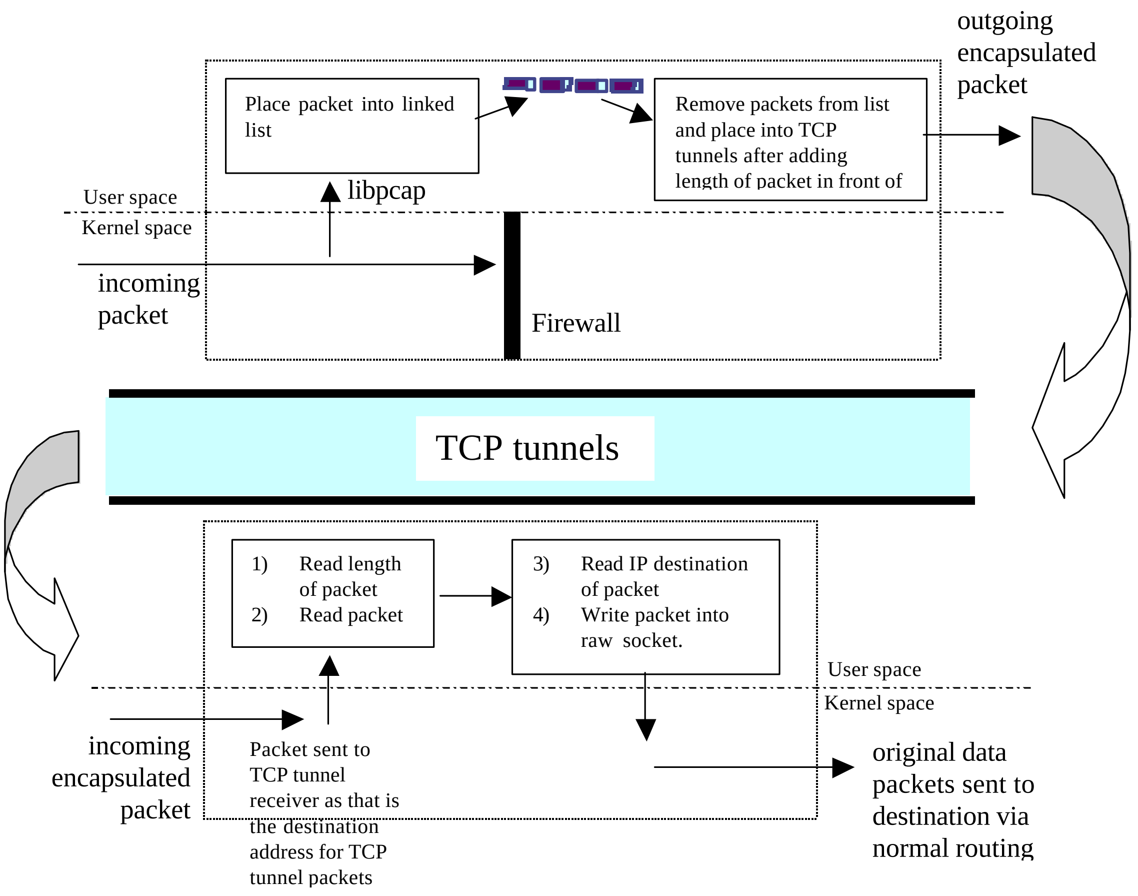

outgoing encapsulated packet
Place packet into linked list
Remove packets from list and place into TCP tunnels after adding length of packet in front of
libpcap
User space
Kernel space
incoming packet
Firewall
TCP tunnels
1) Read length of packet
2) Read packet
3) Read IP destination of packet
4) Write packet into raw socket.
User space
Kernel space
incoming encapsulated packet
Packet sent to TCP tunnel receiver as that is the destination address for TCP tunnel packets
original data packets sent to destination via normal routing

Figure 8 : Data Flow Diagram of TCP Tunnel Software

4.4 System Deployment and Setup

The Chameleon is deployed and setup as follows.

1) The Chameleon code is installed on the 2 border routers.

2) The receiver software of the Chameleon is run on both the border routers.

3) The sender software of the Chameleon is then run on both the border routers. This sender software is a client server program which will talk to the receiver software on the opposite gateway and create the TCP tunnels between the 2 gateways using standard socket calls like connect () etc.

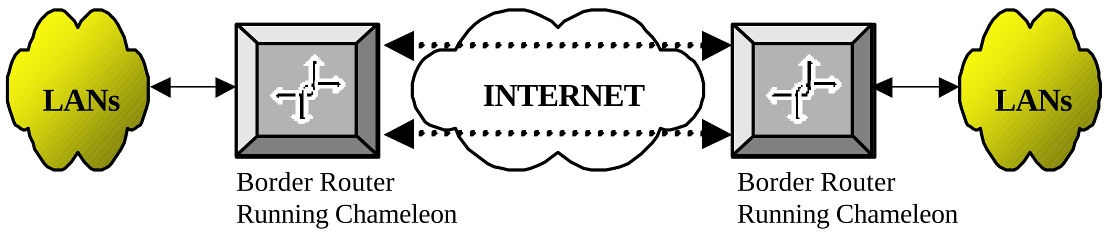


Figure 9 : Deployment and Setup of Chameleon on Border Routers

4) The traffic control mechanisms are put in place on both the border routers using TC scripts provided in the Chameleon package.

5) The admission control and / or firewall rules are put in place on the border routers using scripts provided in the Chameleon package.

After these 5 steps, the Chameleon is up and running. In cases where it is not possible to place any extra software on the border routers, the following method may be used.

1) Place a low-end commodity machine running Linux in front of the border routers. This machine should have at least 2 network interfaces and they should be running at the same speed as the network interfaces on the border routers.

2) Run the Chameleon on the commodity machine.

3) Connect the machine in such a way that traffic that normally would go to the border router, now has to go through the commodity machine first.

4) Connect the other network interface of the commodity machine to the border router.

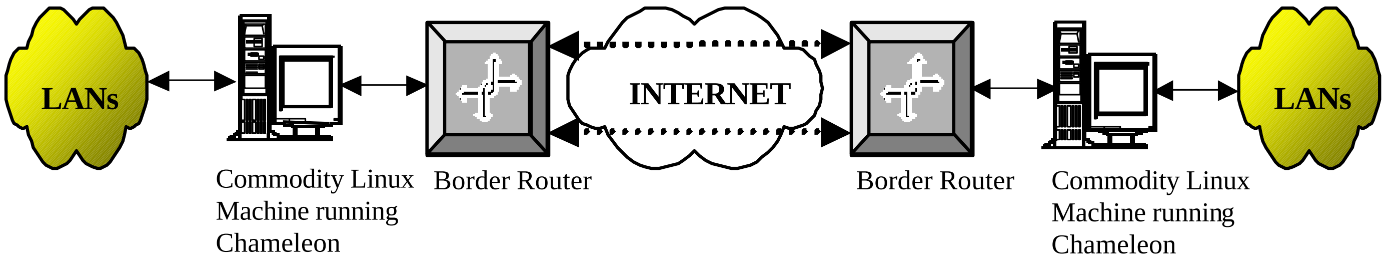

LANs
INTERNET
LANs
Commodity Linux Machine running Chameleon
Border Router
Border Router
Commodity Linux Machine running Chameleon

Figure 10 : Deployment and Setup of Chameleon on Commodity Linux Machines placed in front of the Border Routers

4.5 Limitations of the System

Following are some of the current limitations of the Chameleon.

1) It adds additional overheads into the network due to the additional TCP/IP headers used in the encapsulation process. However, these overheads are minimal with respect to the size of the data packet for typical MTUs.

2) The Chameleon is designed as a logical point-to-point system, i.e., it is assumed that all traffic that enters through one of the border routers will only exit through the other border router. The details of handling traffic that needs to exit the TCP tunnels in between the two border routers using the Chameleon still need to be worked out.

3) The TCP retransmission mechanisms used by the TCP tunnels are unnecessary for certain applications and can even cause large delays in some cases. However, since the Chameleon is meant to be deployed on commodity

machines, this problem can be overcome by creating a modified TCP algorithm that does not do retransmissions. This algorithm can then be used to create TCP tunnels for those classes of traffic that do not require retransmissions, e.g., UDP and / or RTP applications.

CHAPTER 5

5 Experimental Results

In order to verify the effectiveness of the Chameleon, it was necessary to put it through different tests. A series of tests were designed to verify the effectiveness of the Chameleon in protecting various classes of traffic. Tests were also conducted to verify the adaptive property of the Chameleon.

The experiments were conducted on the testbed shown in Fig. x and the Chameleon was deployed on the two border router machines. To simulate a typical RTT delay between the border routers, a 50ms delay was enabled on the delay machine. The bandwidth of the entire testbed was set to 10Mbps.

Each experiment lasted 300 seconds and was run using one of the two traffic mixes given above (depending on what was to be measured). Each experiment was repeated at least 5 times and the results were consistent across multiple runs. The effects of the Chameleon on multiple classes of TCP traffic was not measured as this was not necessary for the following reason: UDP traffic is more aggressive than TCP traffic in consuming bandwidth and experiments were run to determine if the Chameleon could protect TCP traffic from UDP traffic. Replacing the more aggressive UDP flow with a less aggressive TCP flow cannot increase the load. Hence, if it is shown that the Chameleon can protect TCP streams from UDP streams, it follows that the Chameleon can also protect TCP streams from other TCP streams.

The results of the experiments using the different traffic mixes are given below.

5.1 Experimental Setup

For the experiments, some initial feasibility studies were done using simulation but all the results in this thesis were generated on a real testbed with real traffic.

The logical set-up of the testbed is shown in Fig. 11.

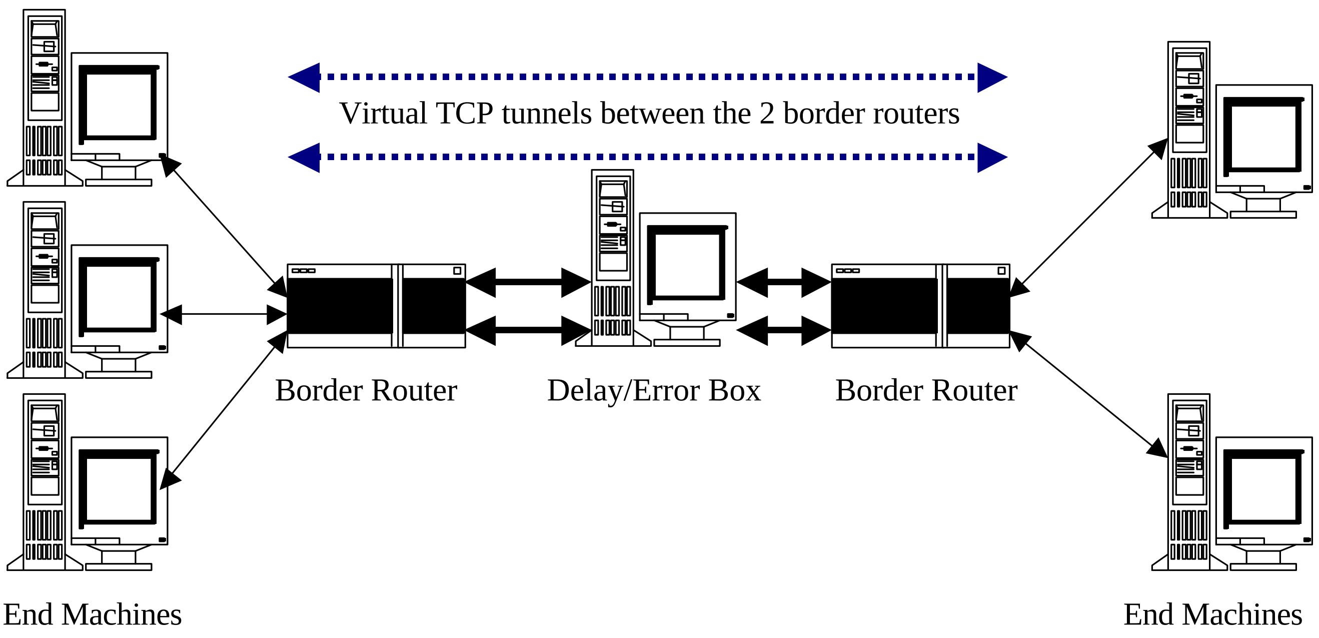


Figure 11 : Logical setup of Experimental Testbed

The end machines were used to generate the various traffic mixes used in the experiments. The Chameleon was deployed on the two border routers and TCP tunnels were created to send traffic between the two border routers. The delay/error box was used to introduce round trip time (RTT) delays and/or packet loss into the network where necessary.

The machines used were all Celeron 300A machines with 128 MB of memory each. All the machines were running Linux 2.2.12 and the implementation of the Chameleon was optimised for this particular operating system.

5.2 Traffic mix of Experiments

Each of the experiments was run for 300 seconds using the traffic mixes given in Table 6. There were two aggressive UDP streams, which were an exponential on/off UDP stream and a CBR UDP stream respectively. Either 10 bulk TCP transfers or 100 HTTP connections were used as the TCP traffic source depending on the experiment. The UDP streams were generated using software created locally. The TCP bulk traffic was generated using ttcp and the HTTP streams were generated using SURGE.

Traffic Mix #1	**Traffic Mix #2**
One 2Mbps UDP CBR stream One 2Mbps UDP exponential on/off stream Ten TTCP bulk transfer	One 2Mbps UDP CBR stream One 2Mbps UDP exponential on/off stream One hundred HTTP sources (making a number of requests each)

Table 6 : Traffic Mixes used in Experiments

5.3 TCP Bulk Sources

To verify the effectiveness of the Chameleon in protecting TCP bulk traffic, multiple experiments using traffic mix #1 were run. Fig. 12 shows the throughput of the TCP bulk sources (generated using ttcp) before and after implementing the Chameleon. These measurements were taken on the end machines shown in Fig. 11. As shown, the Chameleon caused a dramatic improvement in the throughput of the TCP bulk sources. The improvement with the Chameleon is about 500% better than without the Chameleon in place.

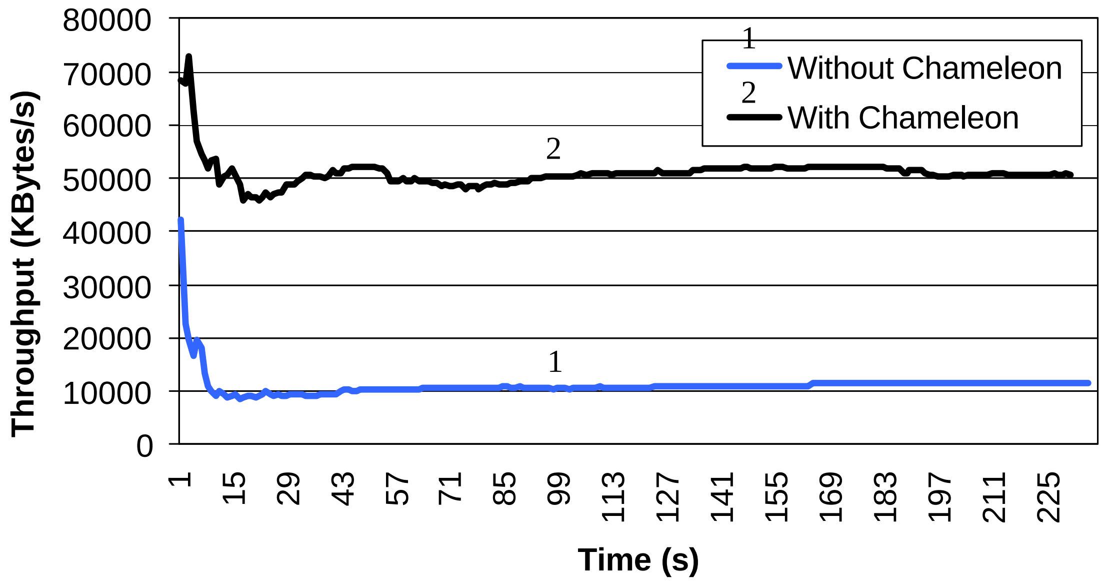


Figure 12 : Graph of Throughput Versus Time (TCP Bulk Traffic)

The throughput of the TCP bulk sources increased because they were no longer being "killed" by the aggressive UDP sources. The UDP sources were responsible for the decrease in the throughput of the TCP bulk sources and the Chameleon isolated the TCP bulk sources from the aggressive UDP sources.

5.4 TCP Interactive Sources

The effects of the Chameleon on TCP interactive traffic were also measured. All measurements were done on the end machines. Traffic mix #2, using HTTP streams as the interactive traffic source, was used for this experiment. For these sources, end-to-end response time is an important performance metric, and a major component of this response time is the TCP RTT. The RTT values will fluctuate widely if these interactive streams encounter delays in the network due to congestion.

The values in Fig. 13 are the average values of all the experiments conducted. It can be

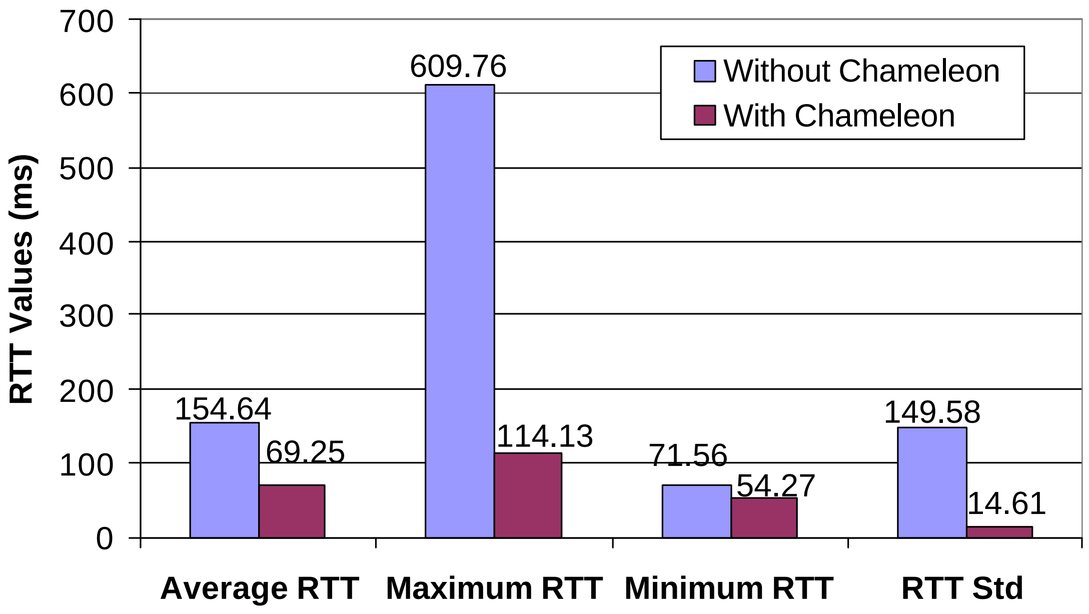


Figure 13 : Graph of RTT Values (HTTP Streams)

seen that the Chameleon provided a substantial improvement in the RTT values for the

HTTP Streams. This was because the Chameleon isolated the multiple short HTTP streams from the more aggressive streams. This result also holds if the UDP streams are replaced with TCP bulk traffic. With the Chameleon, the HTTP streams are not forced to compete along with the same network resources as the more aggressive and longer streams. The percentage improvements in the RTT values are shown in Fig. 14. Note that the mean, standard deviation (std), and the swing between the maximum and the minimum RTT values have all improved substantially.

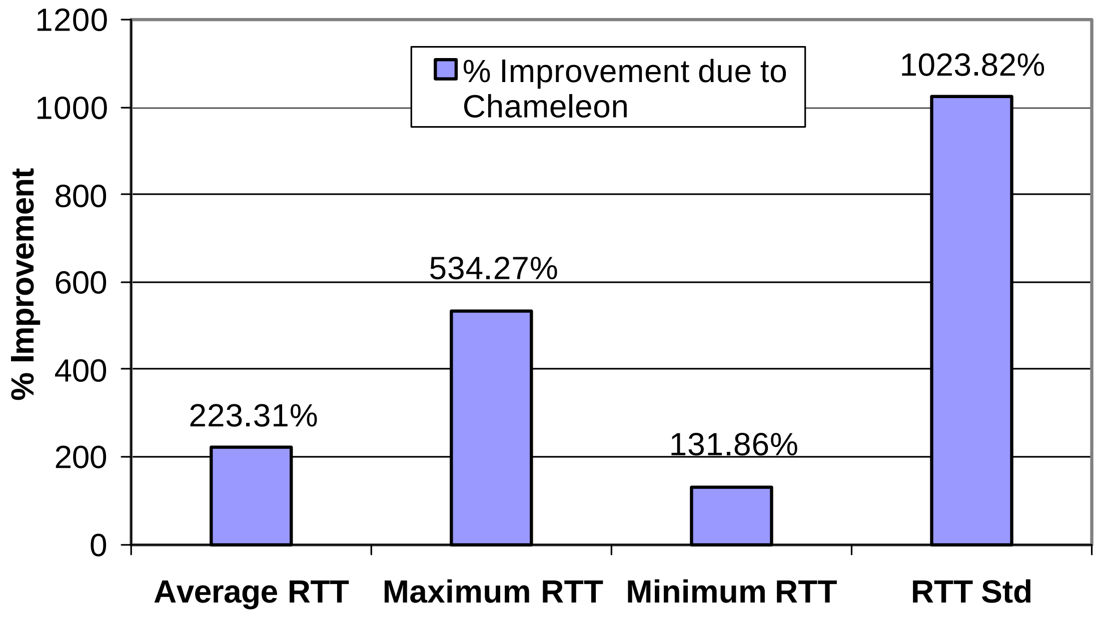


Figure 14 : Graph of the % Improvement in RTT Values due to Chameleon

The Chameleon also aggregated these multiple HTTP streams into longer TCP streams within the TCP Tunnels. This aggregation enabled the shorter streams to better resist packet drops in the network. Previously, if a HTTP stream lost a packet, it would usually go into slow start as the stream did not have enough data packets to trigger TCP's fast retransmit and recovery mechanisms [23,44]. However, when these multiple streams are aggregated, the resulting stream has enough data packets to allow fast retransmit and recovery. Thus, slow and expensive slow starts are avoided and this results in much better RTT values for the individual streams. A paper by Morris [34] has shown the benefits caused by aggregation of flows.

5.5 UDP Sources

For UDP traffic, the amount of packet loss is an important metric. Hence, for the same 2Mbps UDP stream, the packet losses of the stream with and without the Chameleon were measured on the end machines and the results were as follows (shown in Fig. 15). Without the Chameleon, the end machine received 79.65% of the data packets sent. This percentage increased to 96.70% when the Chameleon was used. Hence, the packet loss rate was 20.35% without the Chameleon and 3.30% with the Chameleon.
The experimental results show that the Chameleon dramatically improves the number of packets of the UDP stream received by the UDP receiver. The improvement is as much

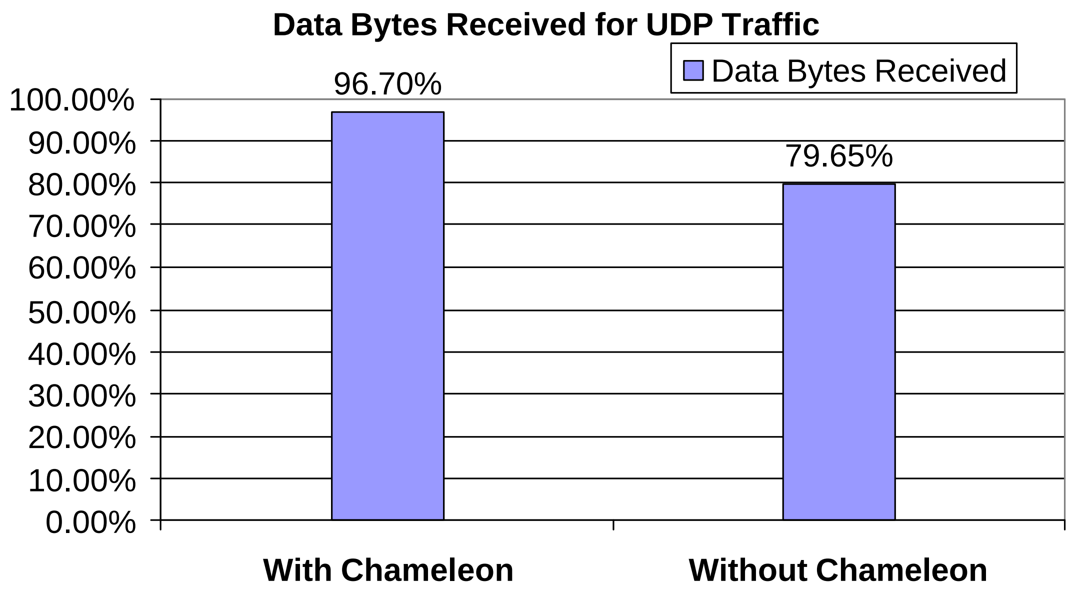


Figure 15 : Graph of Percentage of Data Bytes Received

as 17% or more in the number of packets received by the UDP receiver.

5.6 Adaptive nature of the Chameleon

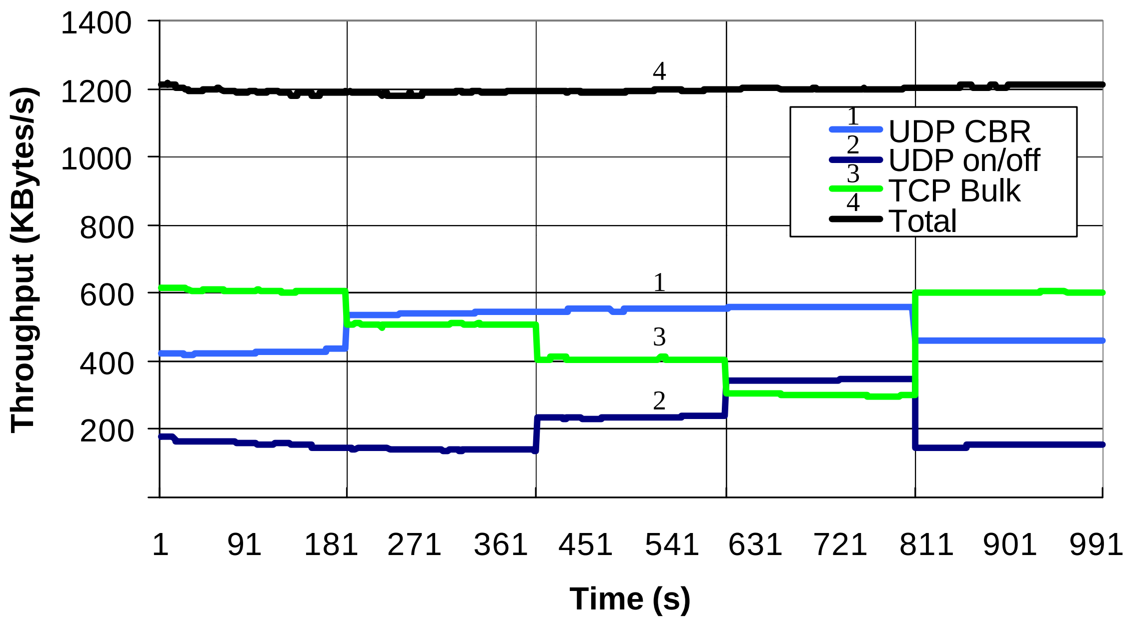


Figure 16 : Graph of Throughput Versus Time (Adaptive)

In this series of experiments, traffic mix #1 was used. However, different bandwidth guarantees were given to the various UDP sources while the experiment was going on (i.e., the contracts for the UDP sources were changed periodically while the experiment was running). For the Chameleon to be adaptive, the non-constrained traffic (TCP bulk traffic in this case) should automatically ramp up and down according to the bandwidth available after the new contracts for the UDP sources have been satisfied.

The measurements were taken on the on the border routers. The bandwidth used by the UDP sources changed every 200 seconds (indicated by the grid lines on the x-axis for Fig. 16). This was expected as the contracts for the UDP sources changed every 200 seconds as well. As Figure 8 shows, the bandwidth for the TCP bulk traffic automatically adjusted itself to use the remaining bandwidth. The total bandwidth used by the Chameleon did not fluctuate much. This indicates that the Chameleon was always making full use of all available bandwidth. Hence, this experiment clearly shows the adaptive nature of the Chameleon to changes in the available bandwidth.

5.7 Chameleon as a reliable and effective data-link layer

In recent years, wireless networks have become increasingly common and an increasing number of devices are communicating with each other over lossy links. Unfortunately, TCP performs poorly over lossy links as it is unable to differentiate the packet corruption in lossy links from packet congestion.

Hence, it becomes necessary to find a way to increase the throughput of TCP over these lossy links. However, it may also be difficult for the end user to become aware of the existence of these lossy links as the lossy link may be just one link out of many links in the network. In cases where lossy links make up just a few links in a network, a better solution would be to deploy the Chameleon across those links and enable TCP and / or IP modifications that improve performance over these lossy links. These modifications could be to enable Forward Error Correction (FEC) checksums, to turn on Explicit

Loss Notification (ELN), or to use modified TCP algorithms optimised for lossy environments like I-TCP and / or SNOOP.

One TCP modification that performs especially well over lossy links is HACK. This algorithm was developed by Lee and Balan et al and is fully described in [40].

HACK was used in the Chameleon and experiments were run to determine if the Chameleon could serve as a reliable data link layer. The Chameleon (with HACK) was compared with normal TCP implementation like SACK [23,32,38] and NewReno [17,18,23,46] over lossy links.

The Chameleon was deployed over different kinds of lossy links and the effects of using various TCP protocols with the Chameleon were measured. TCP bulk data generated using iperf was transferred between the end machines and the throughput was measured. 2 Megabytes of data was transferred between the client and the server for each run. The lossy link was set to 10 Mbps. Fig. 17 shows the setup for this experiment.

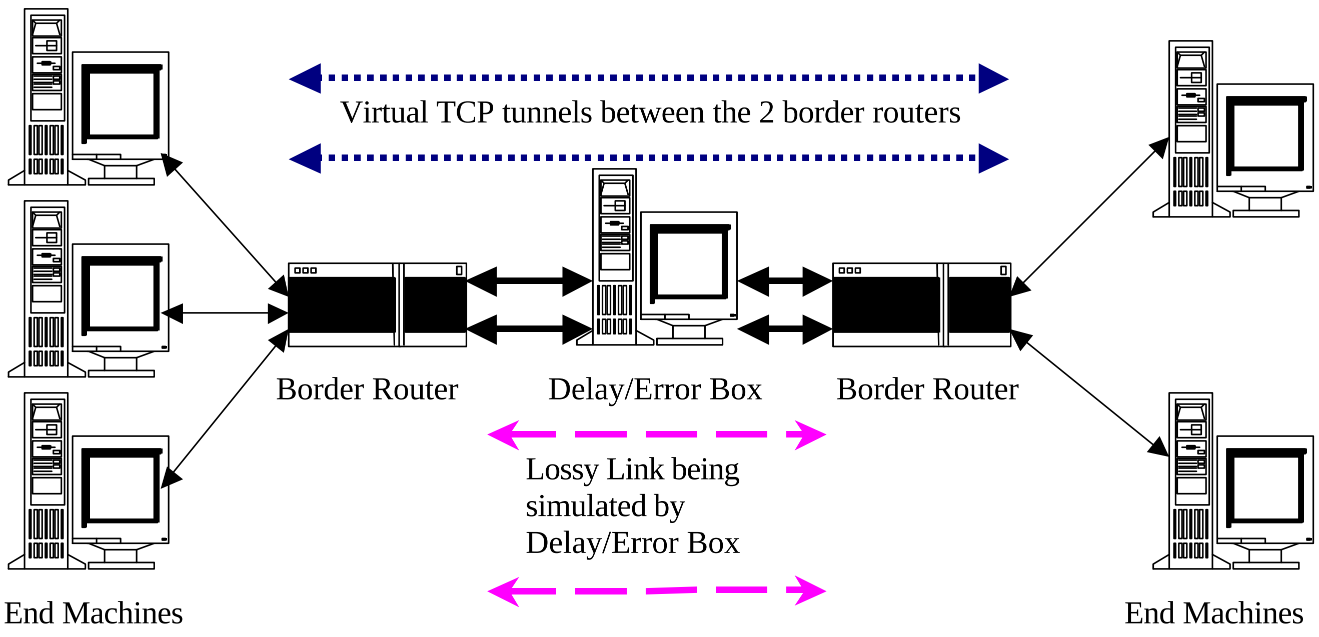

Virtual TCP tunnels between the 2 border routers
Border Router
Delay/Error Box
Border Router
Lossy Link being simulated by Delay/Error Box
End Machines
End Machines

Figure 17 : Logical Setup of Experimental Testbed for lossy link experiments

5.7.1 Random Single Packet Errors

Fig. 18 shows the throughput of the Chameleon for different packet corruption percentages (ranging from 2% to 15%) over a short latency (10 ms RTT) lossy link. The corruption is modelled as random single packet errors. As can be seen, the Chameleon performs best when either SACK or HACK are employed as the TCP implementation. NewReno does not perform as well as either SACK or HACK.

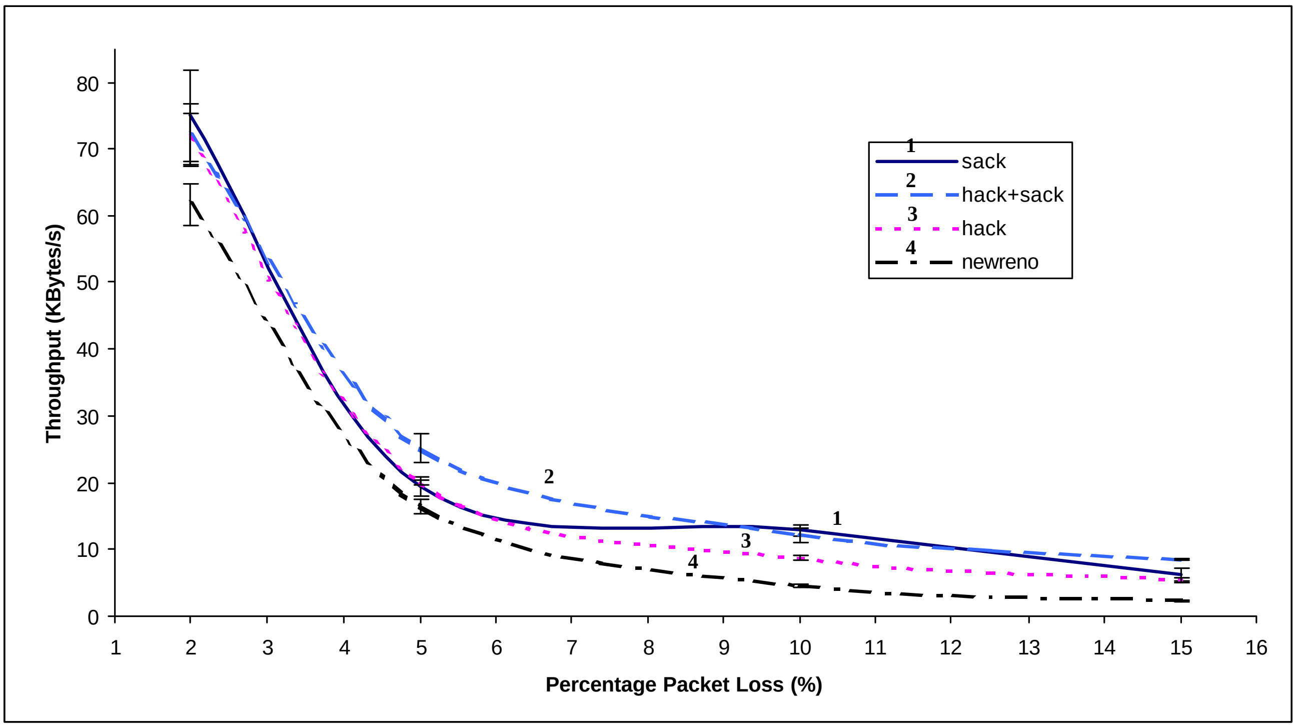


Figure 18 : Throughput of Chameleon versus percentage packet loss for short latency (10 ms) link with random single packet errors

Fig. 19 shows the results of the same experiment. The only difference is that the lossy link is now a long latency link (300 ms RTT). The packet corruption percentages and type of corruption (random single packet errors) remain the same as the previous experiment. The results are similar to the experiment with the short latency lossy link. SACK and HACK perform better than NewReno when used by the Chameleon as the TCP implementation.

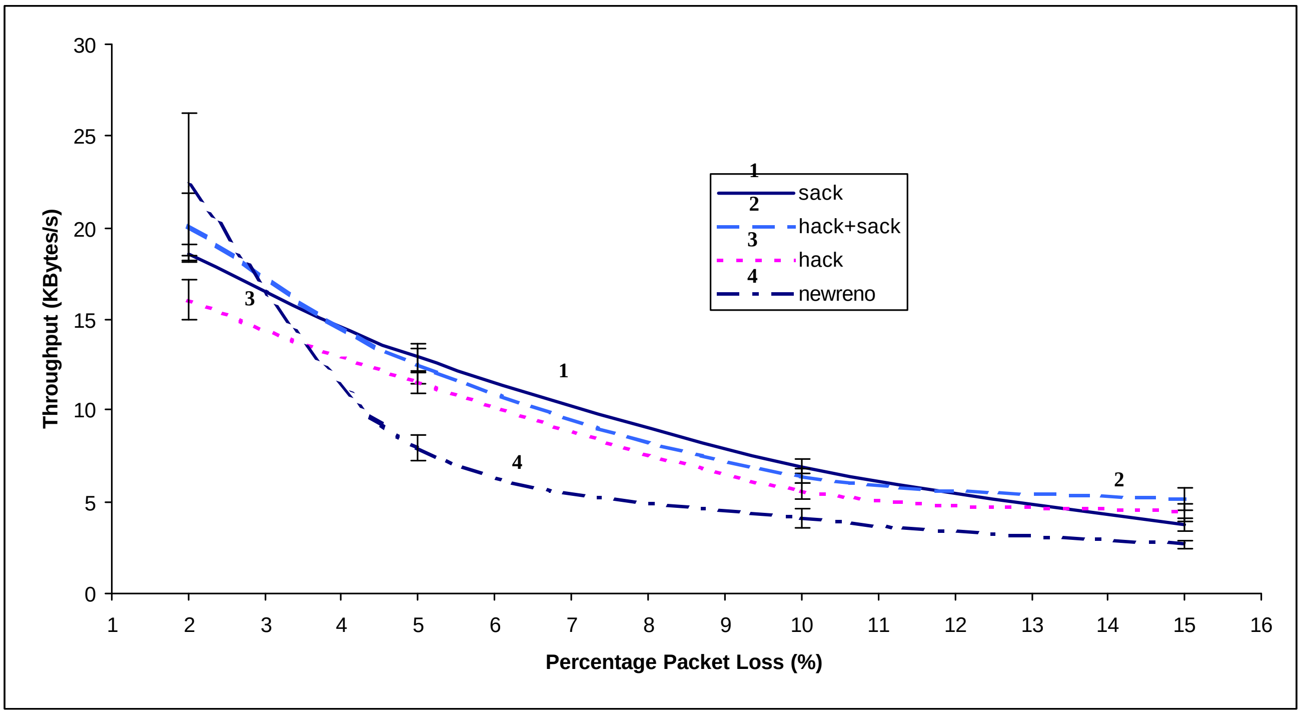


Figure 19 : Throughput of Chameleon versus percentage packet loss for long latency (300 ms) link with random single packet errors

5.7.2 Burst Errors

The situation is very different when burst errors are present in the lossy link. In this scenario, a burst of packets gets corrupted whenever errors occur on the lossy link. TCP is notoriously ineffective in the presence of burst errors as it assumes that the contiguous sequence of corrupted packets are lost due to congestion in the network. Hence, TCP will throttle its sending rate and this results in poor throughput for TCP based applications running over a lossy link with burst errors.

The Chameleon can provide protection from this ineffectiveness of TCP by running a modified version of TCP which is burst error aware over the lossy link. One such modified TCP implementation is HACK.

Experiments were run over a lossy link for different packet corruption percentages and with different burst error lengths for each of those corruption percentages. The results

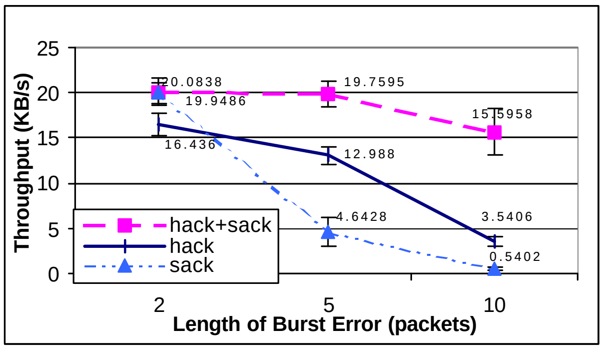


Figure 20 : Throughput of Chameleon for 2% burst error for various burst lengths

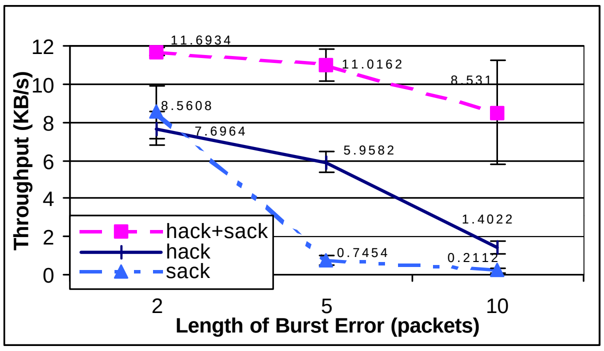


Figure 21 : Throughput of Chameleon for 5% burst error for various burst lengths

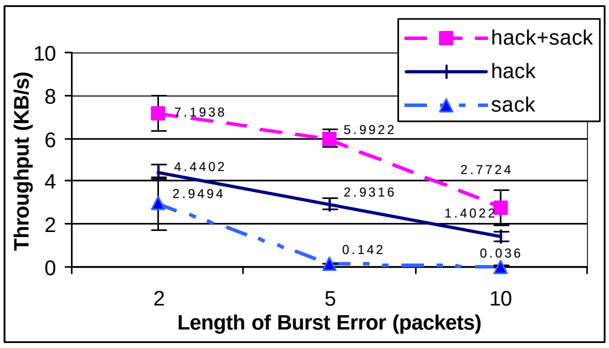


Figure 22 : Throughput of Chameleon for 10% burst error for various burst lengths

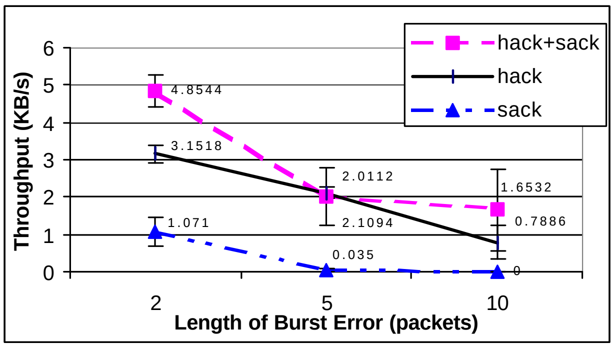


Figure 23 : Throughput of Chameleon for 15% burst error for various burst lengths

are shown in Figs. 20 to 23.

From the results, it can be seen that SACK performs terribly in situations where burst errors are prevalent. HACK performs much better and the best performance is achieved when HACK is implemented together with SACK. This is because HACK is able to leverage upon the out of order packet retransmission algorithms in SACK. HACK creates these out of order situations as it may not be able to recover the headers of all the packets corrupted in a burst due to the random nature of the bit errors within each packet. For example, if say 5 packets are corrupted, HACK may only be able to recover the headers of packets 2, 4 and 5 with packets 1 and 3 being irretrievable. This creates gaps in the receiving window, as HACK will only ask for retransmissions of the packets whose headers it can recover. However, if SACK is activated, these gaps will be detected and handled accordingly. Another example would be as follows; suppose the TCP receiver receives segments x+1, x+2 and x+3 correctly but segment x is corrupted. In this case, the receiver will generate one 'special' ACK in response to segment x and three normal ACKs in response to segments x+1, x+2 and x+3. However, the three normal ACKs will appear to the TCP sender as dupacks as they all will be acknowledging segment x (the next segment expected by the receiver). Hence, the sender will needlessly go into fast retransmit. SACK eliminates this problem as it will be able to inform the TCP sender about the gaps in the receiving window. This leveraging is possible because the HACK and SACK have disjoint sets of operations, thus preventing any conflicts during packet processing. However, it must be re-emphasised that HACK without SACK is still much better than just SACK alone (albeit with potentially more out of order packets being generated). Thus both SACK and HACK can benefit very nicely from each other's properties.

Overall, the results show that the Chameleon really benefits from using a TCP implementation that is capable of differentiating packet losses due to errors from packet losses due to congestion. Thus if the Chameleon is setup across a lossy link using TCP HACK with SACK enabled, the Chameleon can serve as a reliable yet effective datalink layer for use over lossy links.

CHAPTER 5

6 Summary and Future Work

This chapter describes some future work that can be do to the Chameleon system described in this thesis. This chapter also summarises the contents of this thesis.

6.1 Future Work

There is quite a lot of work that can be done to improve the Chameleon. Much of this work is currently in progress. The work that needs to be done is:

1) Port the Chameleon fully into the Linux kernel. At the moment, the TCP tunnel part of the Chameleon runs as a user space programme. This is slow and expensive and it limits the maximum bandwidth supported by the Chameleon to only 10 Mbps. Work is currently underway to move the TCP tunnel aspect of the Chameleon into the Linux kernel.
2) More work needs to be done to test the performance of the Chameleon under various network traffic conditions. The effect of traffic entering in between the Chameleon gateways and the effect of congestion occurring between the Chameleon gateways all need to be studied in greater detail.
3) The Chameleon needs to be optimised to handle UDP traffic. At the moment, TCP tunnels are being used for these kinds of traffic. However, this may not be

feasible for UDP traffic as this kind of traffic does not require the out of order processing and retransmissions provided by TCP. Hence, it is necessary to add a UDP protocol with TCP like flow control to the Chameleon. This modified UDP protocol can be used to create tunnels used to carry UDP traffic.

4) Along with 2, more work needs to be done to create a complete Voice over IP solution using the Chameleon. Hundreds and even thousands of voice streams need to be generated and run over the Chameleon. After this, the traffic management routines, admission control routines and types of TCP/UDP tunnels used can be fine tuned to optimise the Chameleon as a Voice over IP gateway solution.

5) The use of the Chameleon as a reliable yet efficient link level protocol needs to be studied further. In particular, the Chameleon needs to be deployed over real lossy links (such as a satellite link) and its performance measured.

6) Work also needs to be done to make the Chameleon a fully automated system. At the moment, many routines and scripts that control the behaviour of the Chameleon are being run manually. This was to enable us to get the system up and running in as short a time as possible. However, before the Chameleon can be released for use by the general public, many of the processes being run manually at the moment will need to be automated.

7) Finally, the Chameleon needs to be deployed on a real production network. I.e., a network that is used to carry real-world traffic. Only then will the true advantages and disadvantages of the Chameleon become apparent. It will also be possible to optimise the various scripts and routines used to control the traffic management routines of the Chameleon for real world traffic conditions.

6.2 Summary

The Chameleon has been designed as a system meant to be deployed across border routers to provision QoS for various classes of traffic in an easy yet effective manner. No modifications to existing protocols and applications have to be done when deploying the Chameleon and thus it is totally transparent to the end users. Further, no changes need to be done to the actual border routers either as the system can be deployed on a commodity machine placed just in front of the border router.

Experiments on a testbed have shown that the Chameleon is capable of effectively provisioning QoS for various classes of traffic. TCP bulk sources, TCP interactive sources and UDP sources all benefit when the system is operational. The Chameleon is also adaptive and reacts seamlessly to changes in network bandwidth. Thus there is no need to manually provision for rules and/or mechanisms to take care of any excess bandwidth in the network (this could be caused by sources shutting down for example). The Chameleon will automatically detect any change in bandwidth and it will adjust the bandwidth of its various TCP tunnels accordingly. However, it will still satisfy all QoS contracts provided that those contracts are satisfiable.

Work is being done to test the effects of other admission control and traffic management mechanisms on the Chameleon. More work is also being done to fully verify the adaptive properties of the Chameleon under different traffic and network conditions.

Mechanisms to overcome the limitations of the Chameleon mentioned in Chapter 4 Section 5 are currently being developed.

It is also neccessary to deploy the Chameleon across a public network and see how it works with real world traffic and network conditions. Currently, the Chameleon has only been tested in private networks and with generated traffic.

Along with this, work is being done to test the effectiveness of the Chameleon in protecting voice traffic streams as part of our long-term goal of deploying the Chameleon in IP telephony border gateways.

References

[1] A. Bakre, B. R. Badrinath, *"Handoff and System Support for Indirect TCP/IP"*, Proceedings of Second Usenix Symposium on Mobile and Location-Independent Computing, April 1995.

[2] A. Conta, S. Deering, *"Generic Packet Tunneling in IPv6 Specification"*, RFC2473.

[3] *"Apache Webserver"*, http://www.apache.org/.

[4] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, L. Zhang, *"Promoting the Use of End-to-End Congestion Control in the Internet"*, RFC2309

[5] B. Mah, P. Sholander, L. Martinez, L. Tolendino, *"IPB: An Internet Protocol Benchmark Using Simulated Traffic"*, Proc. 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1998

[6] B. P. Lee, R. K. Balan, L. Jacob, W. K. G. Seah, A. L. Ananda, *"TCP Tunnels: Avoiding Congestion Collapse"*, To appear in the proceedings of LCN 2000, Nov. 2000.

[7] B. S. Bakshi, P. Krishna, N. H. Vaidya, D. K. Pradhan, *"Improving Performance of TCP over Lossy Networks"*, Texas A&M University.

[8] C. A. Kent, J. C. Mogul, *"Fragmentation Considered Harmful"*, Dec 1987, Digital Western Research Laboratory.

[9] C. Dovrolis, D. Stiliadis, P. Ramanathan, *"Proportional Differentiated Services: Delay Differentiation and Packet Scheduling"* SIGCOMM 1999.

[10] C. Dovrolis, P. Ramanathan, *"A Case for Relative Differentiated Services and the Proportional Differentiated Model"* IEEE Network, 13(5):26-34, September 1999

[11] Carlos M. Pazos, Juan C. Sanchez Agrelo, Mario Gerla, *"Using Back-Pressure to Improve TCP Performance with Many Flows"*, INFOCOMM 1999.

[12] E. C. Rosen, A. Viswanathan and R. Callon, *"Multiprotocol Label Switching Architecture"*, IETF Draft, April 1999 (work in progress).

[13] H. Balakrishnan, S. Seshan, E. Amir, R. H. Katz, *"Improving TCP/IP Performance over Lossy Networks"*, Proc. 1st MOBICOM, Nov 1995.

[14] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, R. H. Katz, *"A Comparison of Mechanisms for Improving TCP Performance over Lossy Links"*, IEEE/ACM Transactions on Networking, Dec 1997.

[15] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, *"RTP"*, RFC1889.

[16] H. T. Kung and S. Y. Wang, *"TCP Trunking: Design, Implementation and Performance"*, Proc. 7th ICNP, October 1999

[17] J. Hoe, *"Improving the Startup Behaviour of a Congestion Control Scheme for TCP"*, ACM SIGCOMM 1996.

[18] J. Hoe, *"Startup Dynamics of TCP's Congestion Control and Avoidance Schemes"*, Master's Thesis, MIT, 1995

[19] J. Roelof, *"NetSpec: Philosophy, Design and Implementation"*, M.Sc. Thesis, University of Kansas, 1998.

[20] J. Mogul, S. Deering, *"Path MTU Discovery"*, RFC1191

[21] J. Nagle, *"Congestion Control in IP/TCP Internetworks"*, RFC896

[22] J. Postel, *"Transmission Control Protocol"*, RFC793

[23] K. Fall, S. Floyd, *"Simulation-based Comparisons of Tahoe, Reno, and SACK TCP"*, Computer Communications Review, July 1996.

[24] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, G. Zorn, *"Point-to-Point Tunneling Protocol"*, RFC2637.

[25] K. K. Ramakrishnan, S. Floyd, *"A Proposal to add Explicit Congestion Notification (ECN) to IP"*, RFC2481.

[26] K. Nichols, V. Jacobson and L. Zhang, *"A Two-bit Differentiated Services Architecture for the Internet"*, RFC 2638, July 1999.

[27] L.Zhang, S. Deering, D. Estrin, S. Shenker and D. Zappal, *"RSVP : A New Resource Reservation Protocol"*, IEEE Network 7(5), September 1993, pp. 8-18.

[28] M. Allman, D. Glover, L. Sanchez, *"Enhancing TCP Over Satellite Channels using Standard Mechanisms"*, RFC2488

[29] M. Allman, V. Paxson, W. Stevens, *"TCP Congestion Control"*, RFC2581

[30] M. Gates, A. Warshavsky, *"Iperf version 1.1"*, http://www.east.isi.edu/~tgibbons/iperf/index.html

[31] M. Mathis, J. Mahdavi, *"Forward Acknowledgement: Refining TCP Congestion Control"*, ACM SIGCOMM 1996.

[32] M. Mathis, J. Madhavi and S. Floyd, A. Romanow *"TCP Selective Acknowledgement Options"*, RFC 2018.

[33] M. Muuss and T. Slattery, *"ttcp: Test TCP"*, US Army Ballistics Research Lab (BRL), Nov 1985. Enhanced version by Silicon Graphics Incorporated. October 1991, ftp://ftp.sgi.com/sgi/src/ttcp

[34] N. Tang, S. Tsui and L. Wang, *"A Survey of Admission Control Algorithms"*, http://www.cs.ucla.edu/~tang/papers/admission_control_paper.ps, UCLA class paper, March 1998

[35] P. Barford and M. Crovella, *"Generating Representative Web Workloads for Network and Server Performance Evaluation"*, Proc. ACM SIGMETRICS 1998, June 1998

[36] P. Danzig, S. Jamin, *"Characteristics of Wide-Area TCP/IP Conversations"*, Proc. ACM SIGCOMM 1991

[37] P. Danzig, S. Jamin, *"tcplib: A Library of TCP/IP Traffic characteristics"*, University of Southern California Tech Report, USC-CS-91-495 October 1991

[38] R. Bruyeron, B. Hemon, L. Zhang, *"Experimentations with TCP Selective Acknowledgement"*, ACM SIGCOMM

[39] R. Callon et al., *"A Framework for Multiprotocol Label Switching"*, IETF Draft, November 1997 (work in progress).

[40] R. K. Balan, B. P. Lee, K. R. R. Kumar, L. Jacob, W. K. G. Seah, A. L. Ananda, *"TCP HACK : TCP Header Checksum Option to improve Performance over Lossy Links"*, Submitted to INFOCOMM 2001.

[41] R. Morris, *"Scalable TCP Congestion Control"*, Proc. INFOCOMM 2000

[42] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss, *"An Architecture for Differentiated Service"*, RFC 2475, December 1998.

[43] S. Floyd, *"TCP and Explicit Congestion Notification"*, ACM CCR, October 1994.

[44] S. Floyd, K. Fall, *"Promoting the Use of End-to-End Congestion Control in the Internet"*, IEEE/ACM Transactions on Networking, August 1999.

[45] S. Floyd, K. Fall, *"Router Mechanisms to Support End-to-End Congestion Control"*, Technical report, February 1997.

[46] S. Floyd, T. Henderson, *"The NewReno Modification to TCP's Fast Recovery Algorithm"*, RFC2481

[47] S. Floyd, V. Jacobson, *"Random Early Detection Gateways for Congestion Avoidance"*, IEEE/ACM Transactions on Networking, August 1993.

[48] S. Ostermann, *"tcptrace"*, Ohio University, http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html, Dec 1997

[49] S.Y. Wang, *"Decoupling Control From Data for TCP Congestion Control"*, Ph.D. Thesis, Harvard University, September 1999.

[50] T. Sheppard, *"xplot"*, ftp://mercury.lcs.mit.edu/pub/shep/, August 1997.

[51] V. Jacobson, *"Congestion Avoidance and Control"*, Proc. SIGCOMM '98, August 1998.

[52] W. Almesberger, *“Linux Network Traffic Control – Implementation Overview”*, ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps , April 1999.

[53] W. R. Stevens, *"TCP/IP Illustrated, Volume 1"*, Addison-Wesley Publishing Company, 1994.

[54] W. R. Stevens, *"UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI”*, Prentice Hall, 1998.

[55] W. Simpson, *“IP in IP Tunneling”*, RFC1853

[56] W. Simpson, *“The Point-to-Point Protocol (PPP)”*, RFC1661.

[57] W. Simpson, *“PPP Vendor Extensions”*, RFC2153.

[58] W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, B. Palter, *“Layer Two Tunneling Protocol L2TP”*, RFC2661.

[59] *“Libpcap library”*, http://www.tcpdump.org/

[60] *“Ipchains firewall”*, http://netfilter.kernelnotes.org/

Appendix 1 :- Traffic Management (TC) Scripts

```
#!/bin/sh -x

# 3 meg tcp unbounded flow of prio 1 (src 0.3)
# 6 meg udp bounded flow of prio 7 (src 0.2)

tc qdisc del dev eth1 root

tc qdisc add dev eth1 root handle 1:0 cbq bandwidth 10Mbit allot 1514
cell 8 avpkt 1500 mpu 0

tc class add dev eth1 parent 1:0 classid 1:1 cbq bandwidth 10Mbit
rate 3Mbit avpkt 1500 prio 1 allot 1514 weight 3MB

tc qdisc add dev eth1 parent 1:1 red limit 60KB min 15KB max 45KB
burst 20 avpkt 1500 bandwidth 10Mbit probability 0.4

tc class add dev eth1 parent 1:0 classid 1:2 cbq bandwidth 10Mbit
rate 6Mbit avpkt 1500 prio 7 allot 1514 weight 6MB bounded

tc qdisc add dev eth1 parent 1:2 bfifo limit 30000

#tc filter add dev eth1 parent 1:0 prio 1 protocol ip u32 match ip
src 10.10.0.3/32 police rate 3Mbit burst 10K continue classid 1:1

#VENUS
tc filter add dev eth1 parent 1:0 prio 1 protocol ip u32 match ip src
10.10.0.3/32 classid 1:1

#MERCURY
tc filter add dev eth1 parent 1:0 prio 7 protocol ip u32 match ip src
10.10.0.2/32 classid 1:2
```

Appendix 2 :- TCP Tunnel Software

TCP Tunnel Sender Software

```
/* create 4 trunks with messy code to same dest port */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>

#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/tcp.h>


#include <pcap/pcap.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <netdb.h>
#include <ctype.h>

#include "addrtoname.h"
#include "machdep.h"
#include "gmt2local.h"

#define NUM_THREADS 4         /* number of threads */

#define TRUE      1
#define FALSE     0

#define DEFAULT_MAX_BUFFER 100

int num_threads;      /* number of threads                   */

int packettype;

int32_t thiszone;          /* seconds offset from gmt to local time */

int max_buf_length=0,
        buf_length=0,
        dropped_packets=0;

/* Length of saved portion of packet. */
int snaplen = 2000;

struct printer {
       pcap_handler f;
       int type;
};
```

```
void ether_if_print(u_char *, const struct pcap_pkthdr *,const u_char *);
void sl_if_print(u_char *, const struct pcap_pkthdr *,const u_char *);
void sl_bsdos_if_print(u_char *, const struct pcap_pkthdr *,const u_char *);
void ppp_if_print(u_char *, const struct pcap_pkthdr *,const u_char *);
void ppp_bsdos_if_print(u_char *, const struct pcap_pkthdr *,const u_char *);
void fddi_if_print(u_char *, const struct pcap_pkthdr *,const u_char *);
void null_if_print(u_char *, const struct pcap_pkthdr *,const u_char *);
void fddi_if_print(u_char *, const struct pcap_pkthdr *,const u_char *);
void raw_if_print(u_char *, const struct pcap_pkthdr *,const u_char *);
void atm_if_print(u_char *, const struct pcap_pkthdr *,const u_char *);

static struct printer printers[] = {
       { ether_if_print,   DLT_EN10MB },
       { ether_if_print,   DLT_IEEE802 },
/*     { sl_if_print,             DLT_SLIP },
       { sl_bsdos_if_print,       DLT_SLIP_BSDOS },
       { ppp_if_print,            DLT_PPP },
       { ppp_bsdos_if_print,      DLT_PPP_BSDOS },
       { fddi_if_print,    DLT_FDDI },
       { null_if_print,    DLT_NULL },
       { raw_if_print,            DLT_RAW },
       { atm_if_print,            DLT_ATM_RFC1483 }, */
       { NULL,                    0 },
};

static pcap_t *pd;

extern int optind;
extern int opterr;
extern char *optarg;

int cnt=-1;
char *cp, *device;
char cmd_buf[255];

struct packet_list {
   struct packet_list *prev,*next;
   int                  length;
   char                 *data;
};

struct encapsulate_type {
    short packet_length;
    char packet_data[2000];
} packet;

int trunk1, trunk2, trunk3, trunk4;
int trunk1_port, trunk2_port, trunk3_port, trunk4_port;

char *hostname = 0;
int port = 5117, tries = 0;
char *mylocalIPaddrp;

int num_packets=0, max_list_size;

struct packet_list *head,*tail;  /* pointer to link list of packets */

pthread_t *thread_desc; /* array of thread descriptors */

/* mutual exclusion variables for updating total number of primes and
   the value of the maximum prime                                      */
pthread_mutex_t tail_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t head_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t buf_length_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t num_packets_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_attr_t scheduling_policy1, scheduling_policy2;
```

```
static pcap_handler
lookup_printer(int type)
{
	struct printer *p;

	for (p = printers; p->f; ++p)
		if (type == p->type)
			return p->f;

	error("unknown data link type 0x%x", type);
	/* NOTREACHED */
}

void usage(void)
{

	(void)fprintf(stderr, "trunk code version 1.00\n");

	(void)fprintf(stderr,
"Usage: trunk [-l localIPAddr] [ -p port ]\n");
	(void)fprintf(stderr,
"\t\t[ -i interface ] [ -s snaplen (default 2000)]\n");
	(void)fprintf(stderr,
"\t\t[ -h hostname (required) ]\n");
	exit(-1);
}




/* make a clean exit on interrupts */

void cleanup(int signo)
{
	struct pcap_stat stat;

	/* Can't print the summary if reading from a savefile */
	if (pd != NULL && pcap_file(pd) == NULL) {
		(void)fflush(stdout);
		putc('\n', stderr);
		if (pcap_stats(pd, &stat) < 0)
			(void)fprintf(stderr, "pcap_stats: %s\n",
			    pcap_geterr(pd));
		else {
			(void)fprintf(stderr, "%d packets received by filter\n",
			    stat.ps_recv);
			(void)fprintf(stderr, "%d packets dropped by kernel\n",
			    stat.ps_drop);
		}
	}
	exit(0);
}

void initialize_list() {

 struct packet_list *tmp, *tmp1;
 int count;

 head=tail=NULL;

 max_list_size = (max_buf_length / 46) + 1; /* size of buffer divided by min
packet size */

 printf("creating list with %d entries\n",max_list_size);
 fflush(stdout);
 if ((tmp = malloc(sizeof(struct packet_list))) < 0) {
     printf("error with malloc in initialize list!!\n");
     exit(1);
 }

 if ((tmp->data = malloc(sizeof(char) * 2000)) < 0) {
     printf("error with malloc in initialize list!!\n");
     exit(1);
 }

 head = tmp;
 tmp1 = tmp;
 tmp->prev = head;

 for (count=1; count<max_list_size; count++) {

     if ((tmp = malloc(sizeof(struct packet_list))) < 0) {
          printf("error with malloc in initialize list!!\n");
          exit(1);
     }

     if ((tmp->data = malloc(sizeof(char) * 2000)) < 0) {
         printf("error with malloc in initialize list!!\n");
         exit(1);
     }
/*    printf("count = %d ",count); */
    tmp1->next=tmp;
    tmp->prev = tmp1;
    tmp1 = tmp;
 }

 tmp1->next=head;
 head->prev = tmp1;
 tail=head;

 printf("Created a circular linked list with %d packets\n",max_list_size);
 fflush(stdout);

}

void
ether_if_print(u_char *user, const struct pcap_pkthdr *h, const u_char *p)
{
       u_int caplen = h->caplen;
       u_int length = h->len;
       struct ether_header *ep;
       u_short ether_type;
       extern u_short extracted_ethertype;
       const u_char *packetp;
       const u_char *snapend;
       static int count=1;
       struct ip *ip;
       char *data;

       struct packet_list *tmp;

/*        printf("if_print 2 %d %d\n",getpid(), getppid()); */
       if (caplen < sizeof(struct ether_header)) {
              printf("[|ether]");
       }

       /*
        * Some printers want to get back at the ethernet addresses,
        * and/or check that they're not walking off the end of the packet.
        * Rather than pass them all the way down, we set these globals.
        */
       packetp = p;
       snapend = p + caplen;

       length -= sizeof(struct ether_header);
       caplen -= sizeof(struct ether_header);
       ep = (struct ether_header *)p;
       p += sizeof(struct ether_header);



/*     data = (char*)malloc( (h->caplen * sizeof(char)) + 2); */
       if ( ((buf_length+length) < max_buf_length)
            && (num_packets < max_list_size)) {
          memcpy(tail->data+2,p,h->caplen);
          memcpy(tail->data,(char *)&length,2);
       }


       ip = (struct ip*) p;

       ether_type = ntohs(ep->ether_type);


       switch (ether_type) {

       case ETHERTYPE_IP:

#ifdef DEBUG
              printf("received packet %d %s -> %s (%d) ",count++,
                     ipaddr_string(&ip->ip_src),ipaddr_string(&ip->ip_dst),
                     length);
              fflush(stdout);

#endif
              if (((buf_length+length) < max_buf_length)
                   && (num_packets < max_list_size))    {

                tail->length = length;

/*              printf("old tail = %d ",tail); */

                tail = tail->next;

/*              printf("new tail = %d\n",tail); */

                pthread_mutex_lock(&buf_length_lock);
                buf_length+=length;
                pthread_mutex_unlock(&buf_length_lock);

                pthread_mutex_lock(&num_packets_lock);
                num_packets++;
                pthread_mutex_unlock(&num_packets_lock);

#ifdef DEBUG
                printf("(%d)\n",num_packets);
                fflush(stdout);
#endif

              }

              else {

#ifdef DEBUG
               if (buf_length+length > max_buf_length)
                   printf("max buffer length exceeded!! packet is dropped");
               else
                   printf("max num of packets exceeded!! packet is dropped");
#endif

               dropped_packets++;

#ifdef DEBUG
               printf(" (%d) (%d) (%d)\n",dropped_packets,buf_length,
                     num_packets);
               fflush(stdout);
#endif

               }

              break;

       default :
             printf("this is not an IP packet!!!\n");
       }
}


void * trunking_thread (void * dummy) {

  struct packet_list *tmp;
  struct ip *ip_hdr;
  struct tcphdr *tcp_hdr;
  struct udphdr *udp_hdr;
  int hlen;
  int left;
  int n;
  char *p;
  int count;
  char array[2000];
  int count1;
  int *trunk;
  static int trunk_count;

  count=1;
  trunk_count=1;
  printf("trunker 2 %d %d\n",getpid(), getppid());
  while(1){
    /* start looping : 1. read a packet from link list
                       2. encapsulate the packet by inserting a
                          length field in front of the packet
                       3. send out the encapsulated packet
                       4. remove packet from list
     */
     if (num_packets > 0) {

       ip_hdr = head->data+2;    /* attach ip header */
       hlen = ip_hdr->ip_hl * 4; /* get ip header length */

       switch(ip_hdr->ip_p) {
            case IPPROTO_TCP:
               tcp_hdr = ip_hdr+hlen;
            break;

            case IPPROTO_UDP:
               udp_hdr = ip_hdr+hlen;
            break;

            default :  break;
       }

     /* insert scheduling code here */
```

```
       if (ip_hdr->ip_dst.s_addr == inet_addr("10.10.0.2")) {
          trunk = &trunk1;
       }

       else {

         if (ip_hdr->ip_dst.s_addr == inet_addr("10.10.0.3")) {
            trunk = &trunk2;
         }

         else {

           if (ip_hdr->ip_dst.s_addr == inet_addr("10.10.0.4")) {
              trunk = &trunk3;
           }

           else {
#ifdef DEBUG
              printf("strange ip address %x (%s) %x\n",
                      ip_hdr->ip_src.s_addr,
                      ipaddr_string(&ip_hdr->ip_src),inet_addr("10.10.1.4"));
#endif
              trunk = &trunk4;
              goto DONE;
           }
         }
       }

       n = head->length;
        left = n + sizeof(short);
/*        p = (char *) &packet; */
       p = head->data;

#ifdef DEBUG
        printf("get a packet, n %d left is %d\n", n, left);
#endif

       while ((n = write(*trunk, p, left)) != left){
          if (n < 0) {
            printf("write failed !!!!!!!!1\n");
             perror("");
            exit(0);
          }
          p += n;
         left -= n;
       }

#ifdef DEBUG
       printf("removed packet %d (%d) ",count++,trunk_count);
       fflush(stdout);
#endif

DONE :
       pthread_mutex_lock(&buf_length_lock);
       buf_length-=head->length;
       pthread_mutex_unlock(&buf_length_lock);

       pthread_mutex_lock(&num_packets_lock);
       num_packets--;
       pthread_mutex_unlock(&num_packets_lock);

#ifdef DEBUG
       printf("(%d)\n",num_packets);
       fflush(stdout);
#endif
```

```
/*       printf("old head = %d ",head);       */

       head = head->next;

/*       printf("new head = %d\n",head); */

    }
   }
  exit(0);

}

void * network_snooper (void * dummy) {

       register int cnt, op, i;
       bpf_u_int32 localnet, netmask;
       register char *cp, *infile, *cmdbuf, *RFileName, *WFileName;
       pcap_handler printer;
       struct bpf_program fcode;
       u_char *pcap_userdata;
       char ebuf[PCAP_ERRBUF_SIZE];

         printf("snooper 2 %d %d\n",getpid(), getppid());
       printf("in snoop thread\n");
       if (device == NULL) {
              device = pcap_lookupdev(ebuf);
              if (device == NULL)
                     error("%s", ebuf);
       }
       pd = pcap_open_live(device, snaplen, 1, 1000, ebuf);
       if (pd == NULL)
              error("%s", ebuf);
       i = pcap_snapshot(pd);
       if (snaplen < i) {
              printf("snaplen raised from %d to %d", snaplen, i);
              snaplen = i;
       }
       if (pcap_lookupnet(device, &localnet, &netmask, ebuf) < 0) {
              localnet = 0;
              netmask = 0;
              printf("%s", ebuf);
       }
       /*
        * Let user own process after socket has been opened.
        */
       setuid(getuid());


       if (pcap_compile(pd, &fcode, cmdbuf, 1, netmask) < 0)
              error("%s", pcap_geterr(pd));

       if (pcap_setfilter(pd, &fcode) < 0)
              error("%s", pcap_geterr(pd));

       printer = lookup_printer(pcap_datalink(pd));
       pcap_userdata = 0;

       (void)fprintf(stderr, "trunk snoop: listening on %s\n",
                     device);
       (void)fflush(stderr);

       if (pcap_loop(pd, cnt, printer, pcap_userdata) < 0) {
              (void)fprintf(stderr, "trunk snoop: pcap_loop: %s\n",
                  pcap_geterr(pd));
              exit(1);
       }
       pcap_close(pd);
```

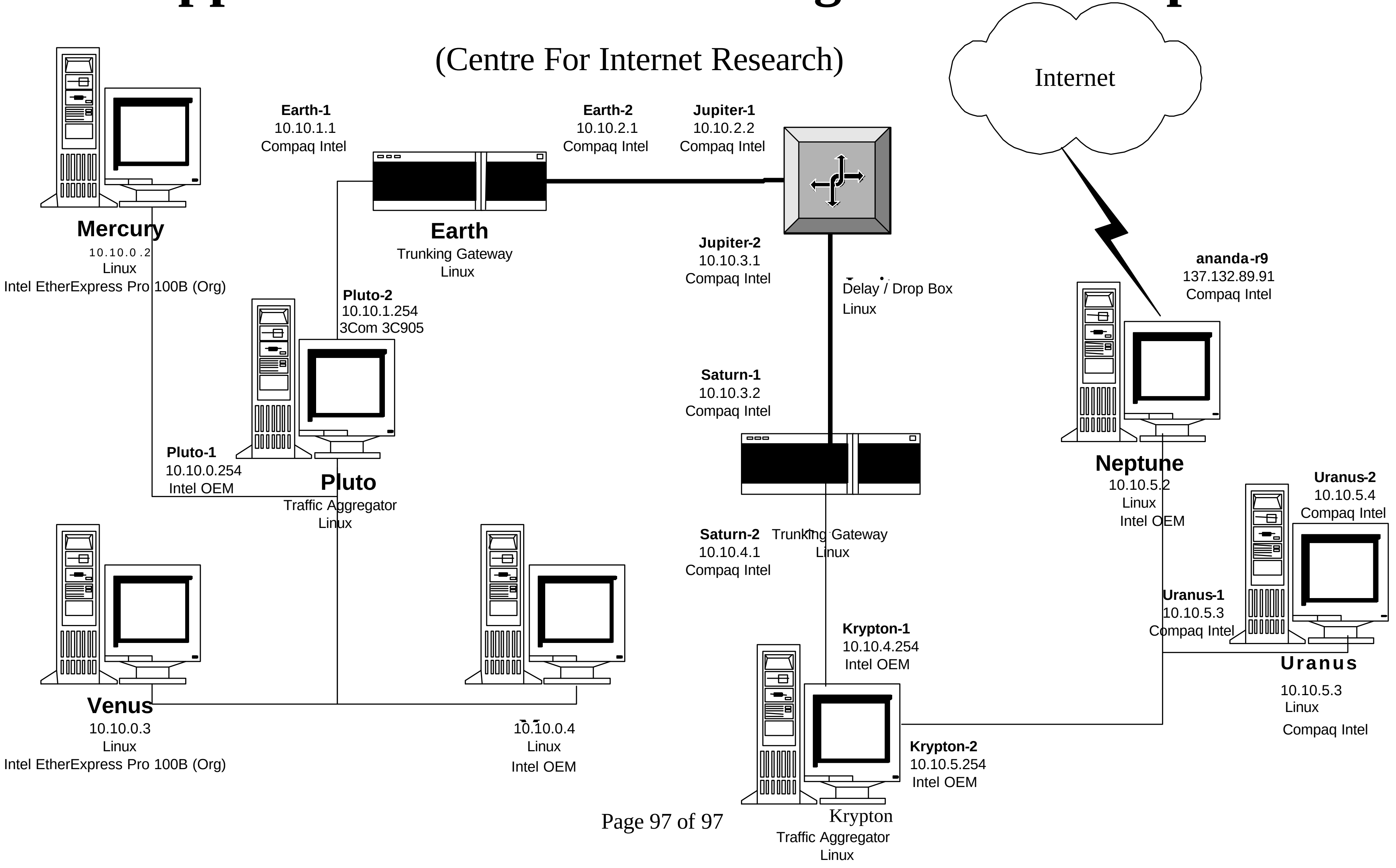
```
      exit(0);

}

int getline_aux (FILE *file, char *buffer, unsigned int n)
{
  int reading = 0;
  while (!reading) {
    if (!fgets (buffer, n, file))
      return 0;
    reading = (buffer [0] != '#');
  }
  return 1;
}


void error_msg(char *msg)
{
  fprintf(stderr, "trunksend: error in %s\n",msg);
  exit(-1);
}


int fread_header(FILE * fp) {
  char buf[256];

   if (!getline_aux (fp, buf, 256) || (sscanf(buf, "%d", &max_buf_length) !=
1))
    error_msg("fread_header: max_buf_length");

  if (!getline_aux (fp, buf, 256) || (sscanf(buf, "%255c\n", &cmd_buf) != 1))
    error_msg("fread_header: cmd_buf");

  if (!getline_aux (fp, buf, 256) || (sscanf(buf, "%d %d %d %d",
      &trunk1_port, &trunk2_port, &trunk3_port, &trunk4_port) != 4))
    error_msg("fread_header: trunk_port");

  return 1;

}

void read_options() {
  FILE *fp=NULL;

  printf("reading options!!\n");
  fp = fopen("input","r");

  if (fp==NULL) {
    max_buf_length = DEFAULT_MAX_BUFFER;
    strcpy(cmd_buf,"ip");
    trunk1_port = 2000;
    trunk2_port = 2001;
    trunk3_port = 2002;
    trunk4_port = 2003;
  }
  else {
    if ( fread_header(fp) != 1)
      error ("fread_header bug on exit");
    fclose(fp);
  }

  printf("The max buffer length is %d bytes\n",max_buf_length);
  printf("The filtering command is \"%s\"\n",cmd_buf);
  printf("the  ports  on  senders  side  are  %d  %d  %d  %d\n",trunk1_port,
trunk2_port,
          trunk3_port, trunk4_port);
  fflush(stdout);
```

```
}

void setup_trunks () {

  int bufsize, i, op;
  int writesize, pagesize, total;
  struct sockaddr_in sin;
  struct sockaddr_in sinme_trunk1, sinme_trunk2, sinme_trunk3, sinme_trunk4;
  struct hostent *hp;
  int tries=0;
  int yes=1;

  bzero(&sin, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_port = htons(port);

  if(isdigit(hostname[0])){
    sin.sin_addr.s_addr = inet_addr(hostname);
  } else {
    hp = gethostbyname(hostname);
    if(hp == 0){
      fprintf(stderr, "TCPtrunkSend: no such host %s\n", hostname);
      exit(1);
    }
    bcopy(hp->h_addr_list[0], &sin.sin_addr, 4);
    hostname = hp->h_name;
  }

 again:

  trunk1 = socket(AF_INET, SOCK_STREAM, 0);
  if(trunk1 < 0){
    perror("TCPtrunkSend: socket");
    exit(1);
  }

  trunk2 = socket(AF_INET, SOCK_STREAM, 0);
  if(trunk2 < 0){
    perror("TCPtrunkSend: socket");
    exit(1);
  }

  trunk3 = socket(AF_INET, SOCK_STREAM, 0);
  if(trunk3 < 0){
    perror("TCPtrunkSend: socket");
    exit(1);
  }

  trunk4 = socket(AF_INET, SOCK_STREAM, 0);
  if(trunk4 < 0){
    perror("TCPtrunkSend: socket");
    exit(1);
  }

  bzero(&sinme_trunk1, sizeof(sinme_trunk1));
  sinme_trunk1.sin_family = AF_INET;
  sinme_trunk1.sin_port = trunk1_port;
  sinme_trunk1.sin_addr.s_addr = inet_addr(mylocalIPaddrp);

  if (bind(trunk1, (struct sockaddr *) &sinme_trunk1, sizeof(sinme_trunk1)) <
0) {
      perror("bind");
    }

  bzero(&sinme_trunk2, sizeof(sinme_trunk2));
  sinme_trunk2.sin_family = AF_INET;
  sinme_trunk2.sin_port = trunk2_port;
  sinme_trunk2.sin_addr.s_addr = inet_addr(mylocalIPaddrp);

  if (bind(trunk2, (struct sockaddr *) &sinme_trunk2, sizeof(sinme_trunk2)) <
0) {
      perror("bind");
    }

  bzero(&sinme_trunk3, sizeof(sinme_trunk3));
  sinme_trunk3.sin_family = AF_INET;
  sinme_trunk3.sin_port = trunk3_port;
  sinme_trunk3.sin_addr.s_addr = inet_addr(mylocalIPaddrp);

  if (bind(trunk3, (struct sockaddr *) &sinme_trunk3, sizeof(sinme_trunk3)) <
0) {
      perror("bind");
    }

  bzero(&sinme_trunk4, sizeof(sinme_trunk4));
  sinme_trunk4.sin_family = AF_INET;
  sinme_trunk4.sin_port = trunk4_port;
  sinme_trunk4.sin_addr.s_addr = inet_addr(mylocalIPaddrp);

  if (bind(trunk4, (struct sockaddr *) &sinme_trunk4, sizeof(sinme_trunk4)) <
0) {
      perror("bind");
    }


  if(setsockopt(trunk1, IPPROTO_TCP, TCP_NODELAY, &yes, sizeof(yes)) < 0)
    perror("TCP_NODELAY");

  if(setsockopt(trunk2, IPPROTO_TCP, TCP_NODELAY, &yes, sizeof(yes)) < 0)
    perror("TCP_NODELAY");

  if(setsockopt(trunk3, IPPROTO_TCP, TCP_NODELAY, &yes, sizeof(yes)) < 0)
    perror("TCP_NODELAY");

  if(setsockopt(trunk4, IPPROTO_TCP, TCP_NODELAY, &yes, sizeof(yes)) < 0)
    perror("TCP_NODELAY");

  /* trunk is open */
  if(connect(trunk1, (struct sockaddr *) &sin, sizeof(sin)) < 0){
    tries++;
    perror("TCPtrunkSend: Try connect");
    if(tries < 4){
      close(trunk1);
      sleep(5);
      goto again;
    }
    exit(1);
  }

  if(connect(trunk2, (struct sockaddr *) &sin, sizeof(sin)) < 0){
    tries++;
    perror("TCPtrunkSend: Try connect");
    if(tries < 4){
      close(trunk2);
      sleep(5);
      goto again;
    }
    exit(1);
  }

  if(connect(trunk3, (struct sockaddr *) &sin, sizeof(sin)) < 0){
    tries++;
    perror("TCPtrunkSend: Try connect");
    if(tries < 4){
      close(trunk3);
      sleep(5);
      goto again;
    }
    exit(1);
  }

  if(connect(trunk4, (struct sockaddr *) &sin, sizeof(sin)) < 0){
    tries++;
    perror("TCPtrunkSend: Try connect");
    if(tries < 4){
      close(trunk4);
      sleep(5);
      goto again;
    }
    exit(1);
  }

}


int main(int argc,char *argv[]) {

  int count,op;

  head=tail=NULL;
  num_threads=2;

  cnt = -1;
  device = NULL;


       while (
           (op = getopt(argc, argv, "i:p:h:l:s")) != EOF)
              switch (op) {

              case 'i':
                     device = optarg;
                     break;

              case 'p':
                     port = atoi(optarg);
                     break;

              case 'h':
                     hostname = optarg;
                     break;

              case 'l':
                     mylocalIPaddrp = optarg;
                     break;

              case 's':
                     snaplen = atoi(optarg);
                     if (snaplen <= 0)
                            error("invalid snaplen %s", optarg);
                     break;

              default:
                     usage();
                     /* NOTREACHED */
              }


  printf("arguments are i=%s, p=%d, h=%s, l=%s, s=%d\n",
          device,port,hostname,mylocalIPaddrp,snaplen);
  fflush(stdout);

  if (hostname == NULL) {
      printf("hostname cannot be NULL\n");
      usage();

  }

  read_options();

  initialize_list();

  setup_trunks();

  if  ((thread_desc = malloc((num_threads+1)*sizeof(pthread_t))) == NULL) {
    printf("Not enough memory for malloc to allocate thread_array!\n");
    printf("Reduce the size of the array from %d\n",num_threads);
    exit(1);
  }

    printf("main 1 %d %d\n",getpid(), getppid());

  /* set thread scheduling policy to Realtime Round Robin */

    pthread_attr_init(&scheduling_policy1);
    pthread_attr_setschedpolicy(&scheduling_policy1,SCHED_FIFO);

    pthread_attr_init(&scheduling_policy2);
    pthread_attr_setschedpolicy(&scheduling_policy2,SCHED_RR);

  /* create thread to start trunking packets */
    pthread_create(&thread_desc[1],&scheduling_policy1,trunking_thread,NULL);

  /* create thread to start snooping network */

    pthread_create(&thread_desc[0],&scheduling_policy1,
network_snooper,NULL);

    printf("main 2 %d %d\n",getpid(), getppid());

  /* wait for all the threads to end */
    pthread_join(thread_desc[1], NULL);

  printf("Trunking has ended!!");


} /* main () */
```

TCP Tunnel Receiver Software

```
/* receiver using select. i.e. single threaded concurrent server */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <ctype.h>
#include <sys/time.h>
#include <signal.h>
#include <errno.h>
#include <netinet/tcp.h>

#include <sched.h>

#include "ip.h"

#define MAXPACKETSIZE 2000

#define ipaddr_string(p) getname((const u_char *)(p))



struct sockaddr whereto;
struct sockaddr_in *to = (struct sockaddr_in *)&whereto;

int thread_desc_array[4];

int     nfds;                   /* from-address length */
fd_set  rfds;                   /* read file descriptor set */
fd_set  afds;                   /* active file descriptor set */

int readn(int fd, char *p, int size) {
  int left;
  int n;

    left = size;
    while ((n = recv(fd, p, left,0)) != left){
      if (n < 0) {
        printf("read failed !!!!!!!!! %d\n",errno);
/*        exit(0); */
        return;
       }
      p += n;
      left -= n;
     }
}

void socket_read(int read_socket, int write_socket) {

  int count=1;

  struct ip *ip;
  char *ipdst;
  char *ipsrc;
  int n, on;

  struct encapsulate {
    short packet_length;
    char packet_data[2500];
  } packet;

/*  printf("in socket read!! readsocket = %d\n",read_socket); */
  fflush(stdout);

  memset(to, 0, sizeof(*to));
  to->sin_family = AF_INET;
  ip = packet.packet_data;
  ipdst = &(ip->ip_dst);

  readn(read_socket, (char *)&packet.packet_length, sizeof(short));
  readn(read_socket, (char *)&packet.packet_data, packet.packet_length);
/*
  ip->ip_len = ntohs(ip->ip_len);
  ip->ip_off = ntohs(ip->ip_off);
*/
  to->sin_addr.s_addr = ipdst[3] << 24 | ipdst[2] << 16 | ipdst[1] << 8
                        | ipdst[0];

#ifdef DEBUG
  printf("packet %d received %s -> %s (%d) (%d)\n",count++,
       ipaddr_string(&ip->ip_src),ipaddr_string(&ip->ip_dst),
              packet.packet_length,read_socket);
/*    printf("%s\n",packet.packet_data); */
#endif

  n = sendto(write_socket, (char *)&packet.packet_data, packet.packet_length,
      0, &whereto, sizeof(whereto));

}

main(argc, argv)
char *argv[];
{
  int s, bufsize, i, s1, cc, yes = 1;
  int readsize, pagesize;
  struct sockaddr_in sin;
  char *p;
  int touchflag = 0;
  int port = 5117;
  int seconds = -1; /* -T */

  int listenfd, *iptr[10];
  socklen_t addrlen, len;
  struct sockaddr *cliaddr;
  pid_t childpid[4];
  struct     sockaddr_in  fsin;  /* the from address of a client */
  char *service;           /* service name or port number */
  int  msock;              /* master  sockets */
  int  fd;
  int  alen;               /* from-address length */
  int sndsock, on, ssock;
struct sched_param sched_policy;

/*
memset(&sched_policy, NULL, sizeof(struct sched_param));
sched_policy.sched_priority = sched_get_priority_max(SCHED_FIFO);
i = sched_setscheduler(0, SCHED_RR, &sched_policy);
printf("scheduling status/errno=%d,%d\n", i, errno);
*/


  printf("in socket read!! s = %d\n",s);
  fflush(stdout);

  if ((sndsock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
      printf("sndsock = socket( failed\n");
      exit(0);
    }
  if (setsockopt(sndsock, IPPROTO_IP, IP_HDRINCL, (char *)&on,
      sizeof(on)) < 0) {
      printf("setsockopt(sndsock failed\n");
      exit(1);
  }



  if (argc < 2) {
      printf("trunkreceive: trunkreceive <port number>\n");
      exit(1);
  }

  port = atoi(argv[1]);

  printf("listening on port %d\n",port);
  fflush(stdout);
  s = socket(AF_INET, SOCK_STREAM, 0);
  if(s < 0){
    perror("socket");
    exit(1);
  }

  if(setsockopt(s, IPPROTO_TCP, TCP_NODELAY, &yes, sizeof(yes)) < 0)
    perror("TCP_NODELAY");

  bzero(&sin, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_port = htons(port);

  if(bind(s, &sin, sizeof(sin)) < 0){
    perror("bind");
    exit(1);
  }
  listen(s, 10);

  cliaddr = (struct sockaddr*) malloc(sizeof(struct sockaddr));


  nfds  = getdtablesize();       /* Max No. of files process can open*/
  FD_ZERO(&afds);          /* clear set */
  FD_SET(s, &afds);        /* add discriptor of msock to set*/

  while (1)
  {   /* indefinite loop to keep serving*/
    memcpy(&rfds, &afds, sizeof(rfds));

    if (select(nfds,&rfds,(fd_set *)0,(fd_set *)0, (struct timeval *)0) < 0)
{
      printf("select:%s\n",strerror(errno));
        exit(1);
    }

    if(FD_ISSET(s,&rfds))    /* test msock ready to read*/
      {

      len = sizeof(cliaddr);
      /* obtain new connection */
      ssock = accept(s, cliaddr, &len);

      printf("accepted a connection at socket %d\n",ssock);

      if (ssock <0) {
         printf("accept failed: %s\n", strerror(errno));
           exit(1);
         }

        /* add the new slave socket to discriptor sets */
        FD_SET( ssock, &afds );
        }

        /* perform echo fuction for each slave socket ready to read */
    for (fd=0;fd<nfds;++fd)
             if(fd != s && FD_ISSET(fd, &rfds))
           socket_read(fd,sndsock);
  }

  close(s);
  while(1);

}
```

Appendix 3 :- TCP-Trunking Testbed Setup


(Centre For Internet Research)
Internet
Earth-1
10.10.1.1
Compaq Intel
Earth-2
10.10.2.1
Compaq Intel
Jupiter-1
10.10.2.2
Compaq Intel
Mercury
10.10.0.2
Linux
Intel EtherExpress Pro 100B (Org)
Earth
Trunking Gateway
Linux
Jupiter-2
10.10.3.1
Compaq Intel
Delay / Drop Box
Linux
ananda-r9
137.132.89.91
Compaq Intel
Pluto-2
10.10.1.254
3Com 3C905
Saturn-1
10.10.3.2
Compaq Intel
Pluto-1
10.10.0.254
Intel OEM
Pluto
Traffic Aggregator
Linux
Neptune
10.10.5.2
Linux
Intel OEM
Uranus-2
10.10.5.4
Compaq Intel
Saturn-2
10.10.4.1
Compaq Intel
Trunking Gateway
Linux
Uranus-1
10.10.5.3
Compaq Intel
Krypton-1
10.10.4.254
Intel OEM
Uranus
10.10.5.3
Linux
Compaq Intel
Venus
10.10.0.3
Linux
Intel EtherExpress Pro 100B (Org)
10.10.0.4
Linux
Intel OEM
Krypton-2
10.10.5.254
Intel OEM
Krypton
Traffic Aggregator
Linux