TCP PERFORMANCE EVALUATION USING TIGER NETWORK TESTBED

---

by

Sridatta Viswanath

Submitted to

the graduate faculty

of the Department of Computer Science

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science

in

Computer Science

March 23, 1999

Clemson University

Clemson, SC 29634-1906

Abstract

Asynchronous Transfer Mode (ATM) delivers important advantages over existing LAN and WAN technologies, including the promise of scalable bandwidths at unprecedented price and performance points and Quality of Service (QoS) guarantees, which facilitate new classes of applications such as multimedia. ATM technology has started to play an important role in the evolution of current workgroup, campus and enterprise networks. These networks internetwork with the vast installed base of existing local and wide area networks through routers. The aim of this research is to setup a heterogeneous network testbed which is similar to real-world networks.

A significant amount of today's Internet traffic, including WWW(HTTP), file transfer (FTP), email (SMTP), and remote access (Telnet) traffic, is carried by the TCP transport protocol. Though there have been many studies to understand the behavior of TCP, it is still not clearly understood due to its complexity. In this paper, the performance of TCP is evaluated on the testbed and the results are explained. The various issues that are investigated include: variation of ATM card transmission rate and variation of number of transmit buffers in the intermediate router. The anomalous TCP behavior, that was observed, is explained. The performance of Solaris TCP and Linux TCP is also compared.

Acknowledgments

First and foremost, I thank Dr. Westall for providing an opportunity to work on this exciting research project. Without his motivation, guidance and suggestions, I would not have been able to complete this project. He gave me invaluable tips for writing this paper, corrected me when I capered minor details in my enthusiasm for bigger results, and also helped me improve my presentation.

My thanks also go to Dr. Ligon of Dept of Electrical and Computer Engineering, who allowed us to use the facilities in Barnes lab and to Ishraq who initially worked with me while setting up the testbed before embarking on the task of writing a device driver. I also thank Dr.Geist for being my second advisor.

Lastly, I would like to express my gratitude to my parents, sister and brother-in-law, and my friends, for their unflinching support.

Contents

Page

ABSTRACT i

ACKNOWLEDGMENTS ii

LIST OF FIGURES v

CHAPTER

1 Introduction 1

- Background 1
- Organization of the remainder of the paper 2

2 Overview 3

- Overview of TCP 3
  - Window Adaptation in TCP 4
- ATM 8
- IP over ATM 9

3 Network Testbed 12

- Architecture of TIGER 12
- Features of the TIGER Testbed 16
- Implementation of TIGER 17
  - Varying the number of transmit buffers on RouterPC 17
  - Controlling ATM card transmission rate 19
  - Enabling IP forwarding 19
  - Capturing sequence number and time 20
  - Varying window size 20
- Implementation Issues 21
  - Connecting 8285 to 8260 21
  - Cell dropping in ATM switch 21
  - Varying transmission rate of the IBM Turboways 25 ATM card 21

4 A TCP Performance Study 22

- Nominal ATM bit rate and TCP throughput 22
- Variation of number of Router Buffers 24
  - Analysis and Explanation of anomalous TCP behavior 27
- Solaris-TCP Investigation 35

5 Concluding Remarks and Future Research 37
Selective Acknowledgments (SACKs) 37
Validated Performance Modeling 38
Self-Similar Traffic Models 38

References 40

APPENDIX 42

A Modification to TCP on chattooga 42
Header file 42
Copy from kernel to user space 42
Reset kernel values 43
Makefile 43
Capturing the data 44
User level get program 45
User level reset program 46

B Barnes Lab 8285 configuration 47
Device Specification 47
Port Status 48
Status of port 7: *bart*'s port 48
Status of port connected to F-D IBM 8260 49
Static route to F-D IBM 8260 49
End System Identifiers (ESI) of registered machines 49

C Acronyms 51

List of Figures

Figure Page

2.1 Protocol layering 3

2.2 Visualization of TCP sliding window 5

2.3 Visualization of slow start and congestion avoidance. *ssthresh* is 16 in this case 7

3.1 TIGER testbed 13

3.2 ATM backbone network with building names 14

3.3 Protocol architecture 15

4.1 Comparison of ATM card transmission rate, theoretical rate and obtained rate. Transmit buffer count is set to maximum (63), segment size is 1448 bytes and window size is 48 KB 23

4.2 Effect of varying number of buffers on throughput. Transmission rate is set to 5000 Kbps and segment size is 1448 bytes. 24

4.3 Queue length for different window sizes. Transmit buffer count is set to maximum (value of 63). Transmission rate is set to 5000 Kbps and segment size is 1448 bytes. 26

4.4 Sequence number versus time with buffer counts of 7 and 8. 30

4.5 Variation of congestion window with time for 7 and 8 buffers 31

4.6 Triggering the flushing of packets in *bart*. 34

4.7 Effect of varying number of buffers on throughput for Solaris. ATM card transmission rate is 5000 Kbps, window size is 48 KB and segment size is 1448 bytes. 35

5.1 TCP SACK Options 37

Chapter 1
Introduction

1.1 Background

Many studies have been carried out to study the performance of TCP. "Congestion Avoidance and Control" by Van Jacobson [6] is one of the first studies to characterize the behavior of TCP. The author demonstrates the working and behavior of different algorithms that are used for implementing of TCP such as slow start, congestion avoidance, and round-trip time calculations.

Vern Paxson [10] discusses the findings from a large-scale study of Internet packet dynamics and characterizes the prevalence of unusual network events such as out-of-order delivery and packet corruption. Several interesting observations emerge from his study about the TCP behavior:

1. Common assumptions such as in-order delivery, FIFO bottleneck queuing, path symmetries are frequently violated.

2. TCP's retransmission strategies work in a sufficiently conservative fashion.

3. The combination of path asymmetries and reverse path noise render sender-only measurement techniques markedly inferior to those that include receiver co-operation.

Allison Mankin [7] studies the performance of Random Drop algorithm for congestion recovery in gateways. This study used stateless gateways where IP packets were forwarded as soon as the processor is available to them. While packets wait for the processor or the link, they are held in a queue that has a size limit and a simple first-come first-serve discipline. When the size limit is reached, each new arrival

to the queue is discarded. The experiments showed that random dropping did not improve performance.

The authors, Jeffrey Semke, Jamshid Mahdavi, and Mathis Matthew [12], identify the requirements for an automatically-tuning TCP to achieve maximum throughput across all connections simultaneously within the resource limits of the sender. They implement an automatic buffer tuning TCP which dynamically adjusts socket buffers to achieve maximum transfer rates on each connection without manual configuration.

Allyn Romanow and Sally Floyd [11] investigate the performance of TCP connections over ATM without ATM-level congestion control, and compare it to the performance of TCP over packet switched networks. The authors also describe two packet discarding techniques: Partial Packet Discard and Early Packet Discard.

1.2 Organization of the remainder of the paper

This paper is organized as follows: Chapter 2 gives an overview of the TCP protocol and an introduction to ATM. The network testbed is detailed in chapter 3. It describes the modifications that were made to the ATM device driver and to the linux kernel and addresses some problems that were encountered and overcome while setting up this testbed. The code for the modification is attached in Appendix A. Chapter 4 presents the results of our TCP performance study and explains the observed anomalous TCP behavior. Chapter 5 identifies more issues that can be investigated on TIGER testbed. The configuration of the ATM machine in Barnes Lab is detailed in Appendix B. Appendix C lists the acronyms used in the paper.

Chapter 2
Overview

2.1 Overview of TCP

Transmission Control Protocol (TCP) is a connection oriented, reliable, adaptive window flow control protocol. Figure 2.1 shows the protocol layering in a TCP/IP environment.

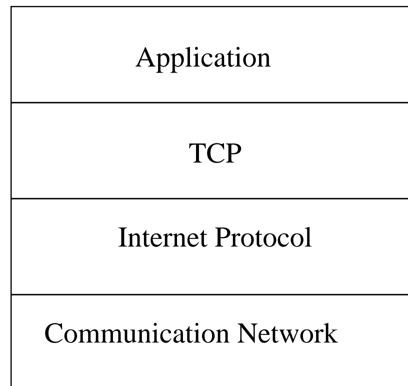

Figure 2.1 Protocol layering

The services provided by TCP to the application layer have the following characteristics:

1. Connection Oriented

   There are exactly two end points communicating with each other on a TCP Connection. The two applications using TCP (normally considered a client and a server) establish a TCP connection with each other before they can exchange data.

2. Reliable

   TCP packetizes the user data into segments, sets a timeout any time it sends data, acknowledges data received by the other end, reorders out-of-order data, discards duplicate data and calculates and verifies a mandatory end-to-end checksum.

3. Byte Stream

   TCP connection exchanges a stream of 8-bit bytes between two applications. Message boundaries are not recognized and are not preserved.

4. Flow Control

   Each end of a TCP connection has a finite amount of buffer space. A receiving TCP only allows the other end to send as much as it can buffer. A sliding window protocol is used for this purpose. If the sender has transmitted a full window of unacknowledged packets, it must stop and wait for an acknowledgement.

2.1.1 Window Adaptation in TCP

Figure 2.2 demonstrates the working of sliding window protocol. In the figure, the numbers represent bytes. The window advertised by the receiver is called the *offered window* and covers bytes 4 through 9, meaning that the receiver has acknowledged all bytes up through and including number 3. The sender computes its *usable window*, which is how much data it can send immediately. In practice, the size of the window offered by a receiver typically varies between 4K bytes and 64K bytes.

The original purpose of the flow control scheme implemented via the receiver window was to prevent a fast sender from overwhelming a slow receiver. The scheme also provided some help in controlling network congestion since no sender could ever have more than a full window of unacknowledged packets outstanding in the network.

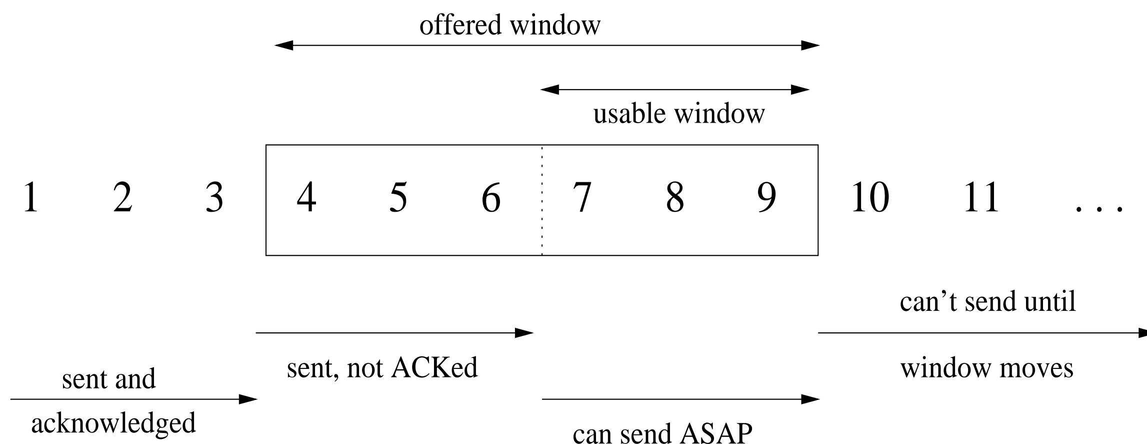

Figure 2.2 Visualization of TCP sliding window

It was found, however, that permitting multiple senders to inject full windows of packets at maximum transmission speed could lead to the phenomenon known as *congestion collapse*. This observation led to the development of the slow start and congestion avoidance algorithms. The objective of these algorithms is to limit the rate at which packets are injected to the rate at which they leave the network.

Slow Start Phase:

The congestion window, *cwnd*, is used to limit the amount of outstanding data injected into the network. When a new connection is established, *cwnd* is initialized to one segment (the segment size is the maximum packet size on the connection). Each time an ACK is received, *cwnd* is increased by one segment. The sender can transmit up to the minimum of the congestion window (*cwnd*) and the advertised window. The congestion window is the limitation imposed by the sender, while the advertised window is the flow control imposed by the receiver.

The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the congestion window is incremented from one to two, and

two segments can be sent. When each of those two segments is acknowledged, the congestion window is increased to four. This provides an exponential increase.

Congestion Avoidance Phase :

Congestion Avoidance is a flow control imposed by the sender based on assessment of perceived network congestion. Congestion avoidance and slow start are independent algorithms with different objectives. In practice they are implemented together. Two variables are maintained for each connection: a congestion window, *cwnd*, and a slow start threshold size, *ssthresh*. The combined algorithm operates as follows:

1. Initialization for a given connection sets *cwnd* to one segment and *ssthresh* to a high value.

2. The TCP output routine can send a minimum of *cwnd* and the receiver's advertised window.

3. When congestion occurs (indicated by a timeout or the reception of duplicate ACKs), one-half of the current window size is saved in *ssthresh*.

   ```
   current window = max(min(cwnd, offered window),2)
   ```

   If congestion is indicated by a timeout, *cwnd* is set to one segment (i.e., slow start).

4. When a new data is acknowledged by the other end, *cwnd* is increased but the way it increases depends on whether slow start or congestion avoidance is being performed.

If *cwnd* is less than or equal to *ssthresh*, slow start is performed; otherwise congestion avoidance is carried out. Slow start continues until halfway to where congestion had occurred (since half of the window size, when congestion occurred, was recorded in step 3), and then congestion avoidance takes over.

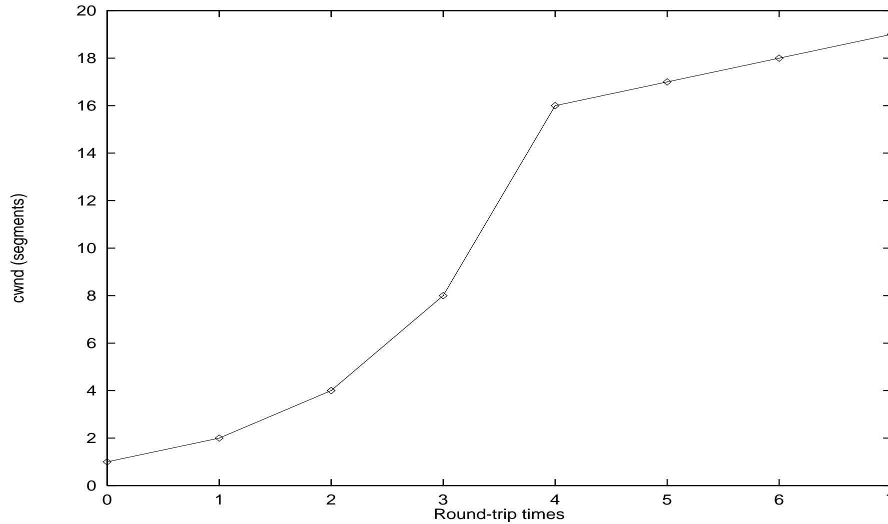

Figure 2.3 Visualization of slow start and congestion avoidance. *ssthresh* is 16 in this case

---

Slow start has *cwnd* set to one segment, and is incremented by one segment every time an ACK is received. As mentioned earlier, this opens the window exponentially. In congestion avoidance, *cwnd* is incremented by $1/cwnd$ each time an ACK is received. *cwnd* grows by one segment every round trip time. This is an additive increase, compared to slow start which has an exponential increase.

```
if(cwnd <= ssthresh)
     cwnd+=1;        /* slow start */
else
     cwnd+= 1/cwnd;  /* congestion avoidance */
```

Figure 2.3 is a visual description of slow start and congestion avoidance. We assume *cwnd* had a value of 32 segments when a timeout occurred. *ssthresh* is then set to 16 segments and *cwnd* is set to 1 segment at time 0. ACK returned at time 1 and *cwnd* is incremented to 2 segments. Two segments are then sent and assuming their ACKs return by time 2, *cwnd* is incremented to 4 segments (one for each ack).

This exponential increase continues until *cwnd* equals *ssthresh*. From this point on the increase in *cwnd* is linear.

Fast Retransmit and Fast Recovery Phase :

TCP generates an immediate acknowledgement (a duplicate ACK) when an out-of-order segment is received. The purpose of the duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected. If three or more duplicate ACKs are received in a row, it is a strong indication that the segment identified in the ACK has been lost. The missing segment is retransmitted without waiting for a retransmission timer to expire. This algorithm [14] is implemented as follows:

1. When the third duplicate ACK is received, set *ssthresh* to one-half the current *cwnd*. Retransmit the missing segment. Set *cwnd* to *ssthresh* plus 3 times the segment size.

2. Each time another duplicate ACK arrives, increment *cwnd* by the segment size and transmit a packet (if allowed by *cwnd*).

3. When the next ACK arrives that acknowledges new data, set *cwnd* to *ssthresh* (the value set in step 1) and continue with congestion avoidance.

2.2 ATM

Asynchronous Transfer Mode ATM is designed to support voice, video, and data with a single, underlying technology. (ATM) is a connection-oriented network technology that uses small, fixed-size cells at the lowest layer. Each ATM connection is identified by a 24 bit integer connection identifier that is used to route cells through the network. The circuit identifier is divided into a 8 bit Virtual Path Identifier (VPI) and a 16 bit Virtual Channel Identifier (VCI)

Two types of connections are:

1. Switched Virtual Circuit (SVC): The virtual circuit is configured upon the request of the host. The host signals its local ATM switch to begin a connection and that switch initiates the establishment of a path across the ATM network.

2. Permanent Virtual Circuit (PVC): The virtual circuit is configured manually by the system administrator. All parameters: source and destination of circuit, quality of service, and the identifiers each host uses for that circuit, are manually set up.

2.3 IP over ATM

Any attempt to create an interface between TCP/IP and ATM must deal with the issues of address resolution and packet encapsulation. There are two fundamentally different ways of running IP over ATM: Classical IP over ATM (CLIP) and Lan Emulation (LANE).

Classical IP over ATM

Classical IP over ATM (CLIP) uses an extended version of Address Resolution Protocol (ARP) called ATMARP. ATMARP translates network layer (IP) addresses into ATM addresses and vice versa. The IETF specifications for CLIP are defined in RFC 1577. RFC 1483 describes the encapsulation of IP datagrams in AAL5. CLIP uses LLC (Logical Link Control) encapsulation which consists of adding an 8 byte LLC/SNAP (Subnetwork Attachment Point) header containing the information needed to identify that this packet is an IP packet.

Hosts that are connected to the same physical ATM LAN may be configured into multiple virtual LANs known as Logical IP Subnetworks (LISs). Each host within a LIS may connect directly to every other host within the same LIS but must connect through an IP router to a host on a different LIS even if both are on the same ATM network. The selector byte of the ATM NSAP address can be used to differentiate multiple LISs for the same ESI. Each LIS must have its own ATMARP server and each host on that LIS is responsible for registering itself with the server.

CLIP may be used in an environment in which all connections are PVCs or in an environment where there are both PVCs and SVCs. In a PVC environment, all connections are manually configured by a network administrator. The signaling demon and the ILMI demon, therefore, are not required. Only the ATMARP demon is necessary. The function of the ATMARP demon, in this case, is to register the translation of IP address into VCCs for each host connected to it by PVC and to enter this information into its ATMARP table.

For SVCs, IP addresses must be resolved into ATM addresses. Every host on the LIS is required to register its IP address and ATM address with the ATMARP server. When a client wants to send data to another client for which it does not have a connection, it sends an ATMARP request containing the destination IP address to the ATMARP server. If the server knows the ATM address of the destination, it responds with an ATMARP reply. Otherwise, it sends an ATMARP NAK, which is an extension of the ARP protocol. Given the ATM address of the destination, the client can set up an SVC to the destination using signaling.

Lan Emulation

LAN Emulation "emulates services of existing LANs across an ATM network". Instead of translating IP addresses to ATM addresses, LANE takes a two step approach. It first translates IP addresses to MAC (Medium Access Control) layer addresses and then translates the MAC layer address to an ATM address. (The MAC layer is a sublayer of the datalink layer in the standard 7-layer protocol stack.) An advantage of such an approach is that by emulating a MAC service, the LAN Emulation protocol, in principle, can be used with other protocol stacks such as NetBIOS, IPX (Internet Packet Exchange), or AppleTalk, in addition to IP. The LANE specifications support emulation of either an IEEE 802.3/Ethernet or an IEEE 802.5/Token Ring LAN.

Just as CLIP can have multiple Logical IP Subnetworks, LANE may have multiple emulated LANs (ELANs). An emulated LAN is simply a group of ATM-attached

devices. There can be several ELANS configured for the same ATM LAN. Membership in an ELAN does not depend on where the end system is physically connected, and an end system could belong to multiple ELANs. Communication between ELANs is possible only through routers or bridges.

Chapter 3
Network Testbed

3.1 Architecture of TIGER

The TIGER [1] testbed is shown in Figure 3.1. It consists of an Ethernet subnet and a CLIP over ATM subnet connected by a PC which serves as a router. In the following discussion this gateway PC will be called *RouterPC*. The machines are located in the Jordan Hall lab of Department of Computer Science and in Barnes lab of Department of Electrical and Computer Engineering. The PCs in the Jordan lab use an ethernet interface and the machines in Barnes lab use an ATM interface. The Barnes lab has 2 Intel Pentium 75 Mhz PCs, named *bart* and *homer*, and the Jordan lab has 2 machines named *chattooga* and *RouterPC* all running the Linux operating system (2.1.117 kernel). The machine *chattooga* is an Intel Pentium 100 Mhz running the Linux operating system (2.1.117 kernel). The kernel in this machine is modified to capture data required for our experiments. *RouterPC* is an Intel Pentium 180 Mhz machine. TCP/IP subsystem of this machine is configured for IP forwarding and acts as a router between the ethernet subnet and the ATM subnet. This *RouterPC* resides on both the subnets. The ethernet LAN in Jordan is 10 Mbps and, in addition to *chattooga* and *RouterPC*, it also has a number of Intel Pentium machines running PC version of Solaris 2.6. These Solaris machines are part of another lab.

The two PCs in Barnes lab have IBM Turboways 25 Mbps ATM Network Interface Cards(NICs). They use the device driver [17] developed by Dr.Westall and Dr.Geist at Clemson University. *RouterPC* also has the same ATM NIC and uses a modified version of the same driver. The modification permits us to control the transmission rate and the number of transmit buffers.

---

[1] TIGER stands for Tcp Ip performance Gathering Experimental Router

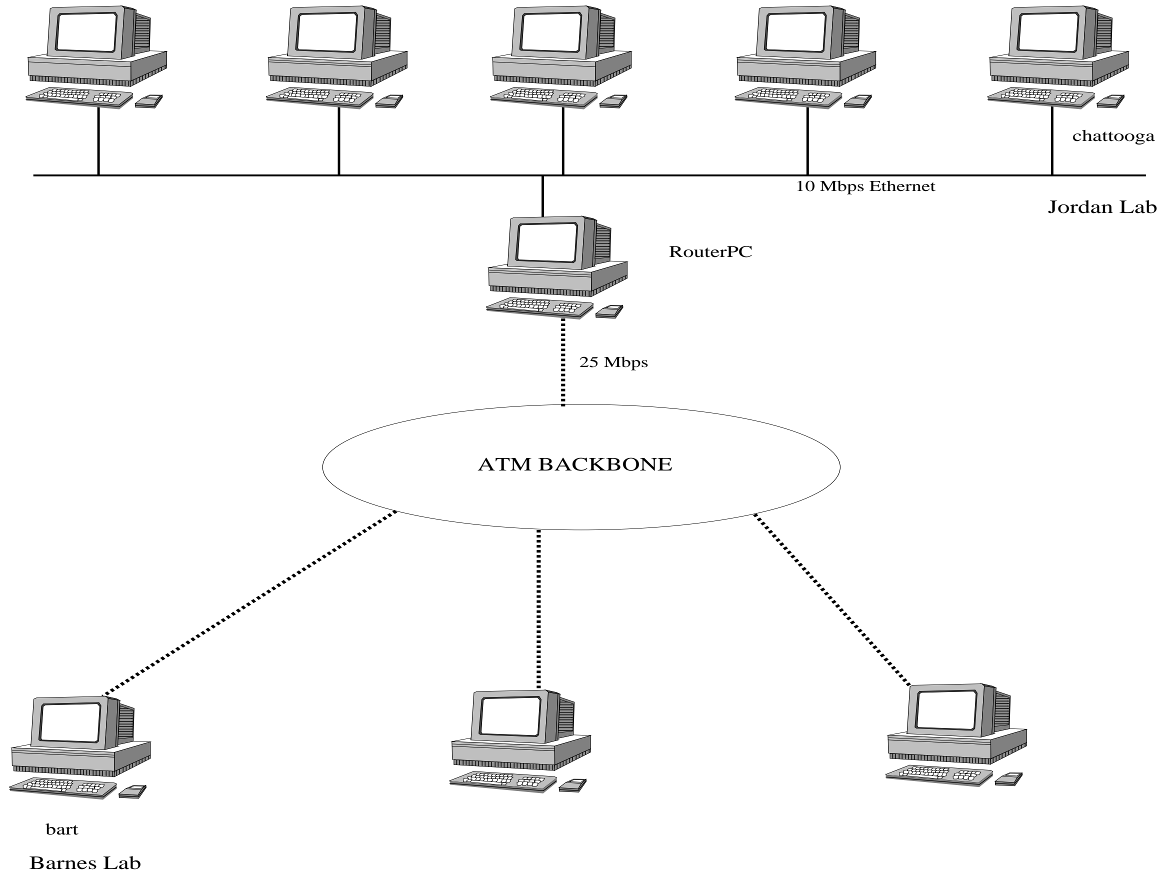

Figure 3.1 TIGER testbed

The ATM backbone consists of IBM 8265, 8260 and 8285 switches located in different buildings, as depicted in Figure 3.2. The end ATM switches in Jordan lab and in Barnes lab are IBM 8285 switches. The link connecting the 8260 switch and the 8285 switch in Barnes lab has a speed of 10 Mbps. The rest of the ATM backbone is connected by 155 Mbps links. The *RouterPC* in Jordan lab and the PCs in Barnes lab run Classical IP over ATM in a Switched Virtual Circuit (SVC) environment.

Figure 3.3 shows the path taken by data when data is transmitted from *chattooga* to *bart*. When *RouterPC* forwards a packet from ethernet interface to ATM interface,

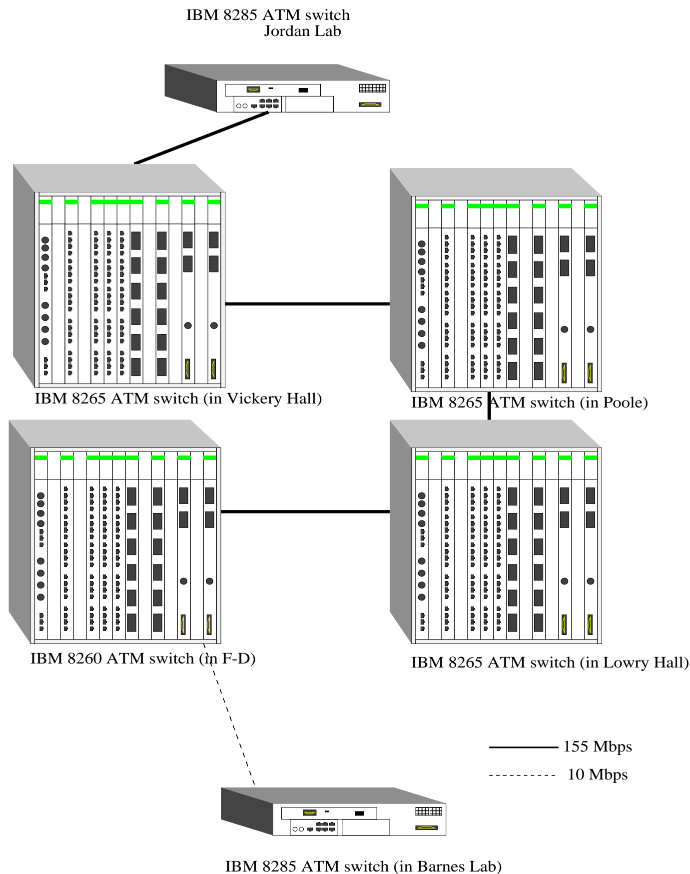

Figure 3.2 ATM backbone network with building names

---

the IP over ATM stack builds an AAL5 frame which is then passed on to the ATM driver.

IP routes are manually configured as shown. A naming convention is followed: machine name (for ethernet interface) appended with "-atm" is used for Classical IP interface. For example: The machine, *bart* on ethernet, can be accessed using *bart-atm* on CLIP interface. IP address of *bart-atm* is `130.127.201.31` where 201 subnet is the CLIP subnet. The machines in Barnes lab are also part of another ethernet subnet. However, that ethernet is not used in our setup.

```
[sridatta@chattooga sridatta]$ netstat -r
Kernel IP routing table
```

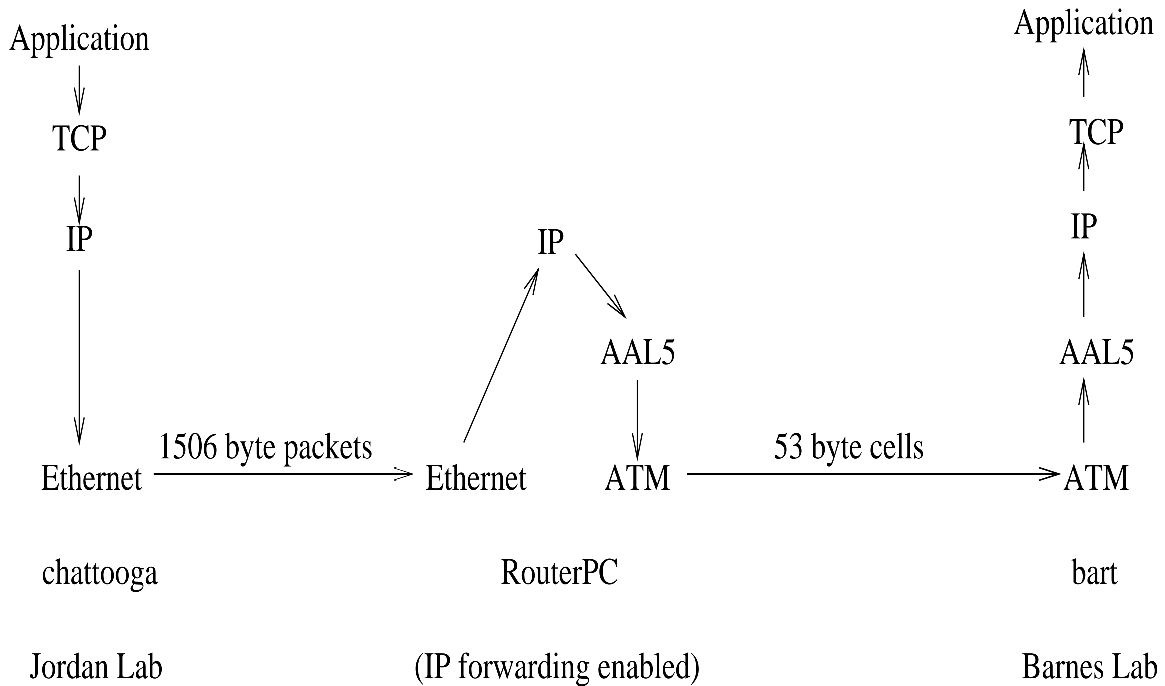

Figure 3.3 Protocol architecture

```
Destination     Gateway        Genmask         Flags MSS Wnd rtt Iface
bart-atm        routerpc.cs.c  255.255.255.255 UGH     0 0     0 eth0
130.127.201.0   *              255.255.255.128 U       0 0     0 atm0
130.127.56.0    *              255.255.255.0   U       0 0     0 eth0
130.127.56.0    *              255.255.255.0   U       0 0     0 eth0
130.127.28.0    poole-7500.cl  255.255.255.0   UG      0 0     0 lec0
130.127.4.0     *              255.255.255.0   U       0 0     0 lec0
127.0.0.0       *              255.0.0.0       U       0 0     0 lo
default         130.127.56.1   0.0.0.0         UG      0 0     0 eth0

[sridatta@bart sridatta]$ netstat -r
Kernel IP routing table
Destination     Gateway        Genmask         Flags MSS Wnd rtt Iface
chattooga.cs.cl routerpc-atm   255.255.255.255 UGH     0 0     0 atm0
130.127.201.0   *              255.255.255.128 U       0 0     0 atm0
130.127.210.0   *              255.255.255.0   U       0 0     0 eth0
130.127.210.0   *              255.255.255.0   U       0 0     0 eth0
127.0.0.0       *              255.0.0.0       U       0 0     0 lo
default         parl-gw.parl.  0.0.0.0         UG      0 0     0 eth0
```

3.2 Features of the TIGER Testbed

Asynchronous Transfer Mode (ATM) delivers important advantages over existing LAN and WAN technologies, including the promise of scalable bandwidths at unprecedented price and performance points and Quality of Service (QoS) guarantees, which facilitate new classes of applications such as multimedia. ATM technology has started to play an important role in the evolution of current workgroup, campus and enterprise networks. These networks internetwork with the vast installed base of existing local and wide area networks.

The TIGER testbed is setup keeping these developments in mind. It has a router that connects an ATM network with an existing LAN network. We believe that this scenario will become common as more and more ATM networks are installed. The unique features of TIGER are:

1. It is a heterogeneous network testbed which is similar to real world networks.

2. The number of transmit and receive buffers in *RouterPC* is dynamically configurable. The number of buffers are variable from 1 to 63 (each buffer can hold one AAL5 frame). Thus, the behavior of protocols under lossy conditions can be studied. Random drop can be readily implemented but drop tail was used in this study.

3. The transmission rate of the ATM interface in *RouterPC* is dynamically configurable and can be varied from 107 Kbps to 25600 Kbps. *RouterPC* can be made a bottleneck by reducing the rate of transmission. Aspects of the performance of protocols, such as when packets get queued and get dropped, can be studied. The transmit rate can be dynamically varied to simulate the effect of competing sources of varying intensity.

4. *RouterPC* also computes and displays the number of packets dropped, packet drop rate (packets dropped per second), transmit buffer queue length, average transmit rate, average packets receive rate, and average drop rate.

5. The ATM network and *RouterPC* are used exclusively for the experiments. Hence, the campus traffic does not interfere with the setup. The ethernet LAN is shared by the testbed and a lab in the Computer Science department. However, the data, required for analysis, is captured when the labs are not open.

6. *chattooga*, *RouterPC* and *bart* use freely available Linux operating system which gives the flexibility for modification and parameter tuning. Hence, wide range of experiments can be performed on this testbed.

7. The packet generating rate at the transmitting end can be controlled and the effect of different types of traffic like Poisson, exponential, self-similar, can be studied.

8. The traffic generator and receiver are file system independent. Hence, the results obtained are not affected by the performance of the underlying file system.

9. Due to the availability of variety of operating systems, a comparative study of various protocol-implementations can be performed.

3.3 Implementation of TIGER

We describe the modifications to the driver on *RouterPC* and to TCP on *chattooga* and *bart*. At times, we delve into portions of source code to elucidate our discussion.

3.3.1 Varying the number of transmit buffers on RouterPC

Varying the number of transmit buffers in *RouterPC* is achieved by executing a program on *RouterPC*. This program takes in the number of buffers with values ranging from 1 to 63.

The program calls an *ioctl* [2] function as shown below (the variable `limit` contains the buffer count from 1 to 63):

```
ioctl(fd, AIO_SETMAXLIM, &limit);
```

The following code is the implementation of the above *ioctl*. This section of the code initializes the driver variable `max_limit` to a value passed by the user.

```
[atmioctl.c]
   case AIO_SETMAXLIM:
      get_user(lim_len, word);
      max_limit = lim_len;
      break;
```

The device driver's sending routine is modified to drop packets if the queue length [3] increases above the `max_limit`. The number of dropped packets is also tracked in a variable, `drop_pacs`, to compute the drop rate. The following segment of the code checks the queue length and drops the packets if the limit is exceeded. The variable `ape->tbcount[lcndx]` has the length of the queue. If this is already equal to the maximum allowed, then `sk_buff`, which holds the frame, is freed. A frame is dropped if it is a data frame and not a signaling frame. A frame having `lcndx` greater than 32 indicates a data frame.

```
[atmxmit.c]
if(lcndx > 32 && ape->tbcount[lcndx] >= max_limit)
{
        atomic_inc((atomic_t *)&drop_pacs);
         if (vcc->pop == 0)
            dev_kfree_skb(skb);
         else
            vcc->pop(vcc, skb);
         return(0);
}
```

---

[2] The ioctl function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with ioctl requests.

[3] Queue length is the number of packets queued in the buffer

3.3.2 Controlling ATM card transmission rate

The IBM Turboways 25 ATM card has a peak rate of 25 Mbps. However, the rate can be set to any value below 25 Mbps by writing into appropriate control registers. To set this rate, the user program calls a library function `ape_tmqcfg()` provided by the ATM device driver. It is called as shown below (`speed` is in Kbps in range from 107 to 25600):

```
ape_tmqcfg(fd, 0, speed * 1024, 1, 1);
```

The function performs an ioctl call after initializing the variable `iobuf`. The ioctl call modifies the values in the control registers of the ATM card.

```
ioctl(apefd, AIO_WTCREG, iobuf);
```

3.3.3 Enabling IP forwarding

A linux system can be used as a router by enabling IP forwarding. Linux has a `proc` [4] file system which can be used for enabling or disabling IP forwarding. IP forwarding can be enabled by executing:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

and by setting IP forwarding to YES in the file `/etc/sysconfig/network`:

```
NETWORKING=yes
FORWARD_IPV4=YES
HOSTNAME=atm
DOMAINNAME=cs.clemson.edu
GATEWAY=130.127.56.1
GATEWAYDEV=eth0
```

[4] /proc is a pseudo-filesystem which is used as an interface to kernel data structures rather than reading and interpreting /dev/kmem. Most of it is read-only, but some files allow kernel variables to be changed.

3.3.4 Capturing sequence number and time

For every TCP packet transmitted by *chattooga* the sequence number of the packet, present size of the congestion window (*cwnd*), slow start threshold value (*ssthresh*), previous ack number, number of packets in flight, and time at which a packet is transmitted are captured. This data is used in creating graphs that assist in understanding and explaining the behavior of TCP. The data is captured in the function `tcp_transmit_skb()` which is located in the file `tcp_output.c`. All packets, whether initial or queued or retransmitting, are transmitted in this function. The captured data is buffered in an array of structures is added to the kernel. A system call `get_seqdata()` is used to retrieve this captured data to user space. The structure is shown below and the complete code is attached in the Appendix A.

```
[seqdata.h]
typedef struct _sqdata {
        unsigned long seq;
        struct timeval time;
        unsigned long cwnd;
        unsigned long ssthresh;
        unsigned long ack;
        int flight;
} _seqdata;
```

3.3.5 Varying window size

The size of the window offered by the receiver can be controlled by the receiving process. This can be changed by changing the buffer space available for storing packets at the receiver using `setsockopt()` (The variable, `bufmax`, contains the desired window size):

```
setsockopt(sock, SOL_SOCKET, SO_RCVBUF, &bufmax, sizeof(int));
```

In linux, `setsockopt()` function initializes the socket variable `rcvbuf` to twice the value of `bufmax`.

3.4 Implementation Issues

Implementing TIGER was a complex task and we encountered a number of issues and overcame difficulties.

3.4.1 Connecting 8285 to 8260

The 8285 ATM switch in F-D building does not implement PNNI protocol. So, we had to find a round about way of connecting the switch to 8260. After many futile efforts, we finally got the link working by configuring 8285 to use NNI protocol and the other end 8260 switch to use IISP [5] protocol. Now for both switches, port status shows : UP and OKAY. Appendix B includes the details of current configuration for the Barnes lab 8285 switch.

3.4.2 Cell dropping in ATM switch

We tried unsuccessfully to make the ATM switches drop cells by trying to overwhelm the switch with traffic. Though we ran sender application on two PCs at full speed, it still wasn't enough to overrun the buffers of the ATM switches. All experiments assume zero packet loss due to ATM switches.

3.4.3 Varying transmission rate of the IBM Turboways 25 ATM card

We were varying the transmission rate from 10 Kbps to 25 Mbps and we were getting absurd results for low rates. After further investigation, we found that the rate of the ATM card cannot be decreased below 107 Kbps. All the experiments presented in this paper use values greater than 107 Kbps.

[5] IISP is an old inter-switch protocol which has to be statically configured

Chapter 4
A TCP Performance Study

In this chapter we present a case study in the application of the TIGER testbed. Our objective is to measure TCP throughput as a function of the number of ATM transmit buffers in the *RouterPC*. Data is transmitted at maximum speed by *chattooga* and sent to *bart*. The hop from *chattooga* to *RouterPC* is a 10 Mbps Ethernet. Queuing occurs in *RouterPC* if the transmission speed on the ATM interface is less than the speed of the Ethernet. For the results reported here the TIGER testbed is configured as follows:

- ATM card transmission rate on *RouterPC* is set to 5000 Kbps (except as otherwise noted.)
- The buffer count on *RouterPC* is varied from 1 to 63.
- Offered window sizes are 32 KB and 48 KB (except as otherwise noted.)
- The segment size is 1448 bytes
- The amount of data transferred for each run is 50 MB.

4.1 Nominal ATM bit rate and TCP throughput

Throughput is a measure of the amount of data transferred per unit time. Because of header and trailer overhead the throughput seen at the TCP level is somewhat less than the nominal bit rate on the bottleneck ATM link. We can express the maximum TCP throughput as a function of the segment size and the bit rate of the bottleneck link. The length of an ATM cell is fixed at 53 bytes: 48 bytes of data and 5 bytes of header. Additionally, each AAL5 frame has an 8 byte trailer at the end. If the length of the frame is not evenly divisible by 48, padding bytes are appended to fill out the last cell.

Suppose F bytes of application level data are contained in a single TCP/IP packet. Forty bytes of TCP and IP headers will be added. In addition, if Classical

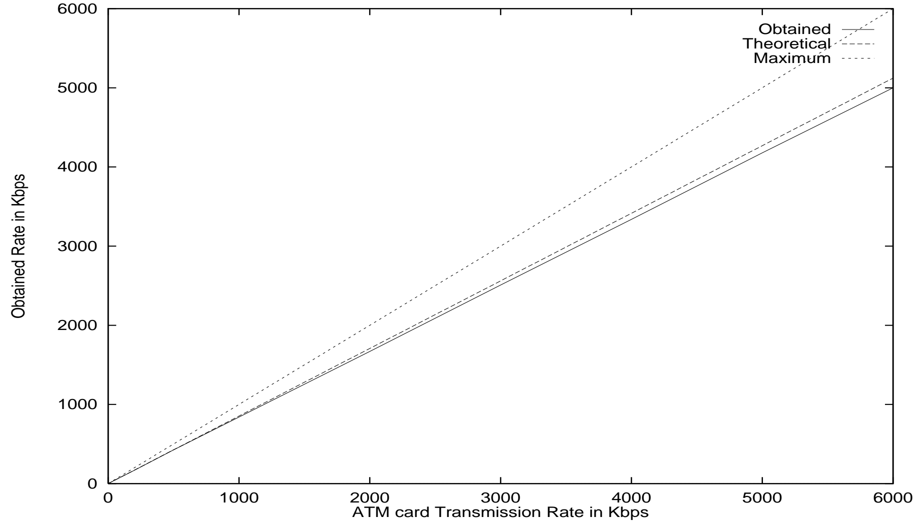

Figure 4.1 Comparison of ATM card transmission rate, theoretical rate and obtain ed rate. Transmit buffer count is set to maximum (63), segment size is 1448 bytes and window size is 48 KB

IP over ATM is used, each packet has an 8-byte LLC/SNAP header in accordance with RFC 1483. When the packet reaches the ATM protocol levels, the 8-byte trailer is appended. The data is then divided into 48-byte cells with padding, if necessary, to fill out the last cell. Each cell has a 5-byte header added. The total number of cells created with CLIP will be

$$\left\lceil \frac{(F + 40 + 8 + 8)\ bytes}{48\ bytes/cell} \right\rceil$$

The total number of bytes transmitted is

$$53 * \left\lceil \frac{(F + 56)}{48} \right\rceil\ bytes$$

For F=1448 bytes:

$$Total\ Bytes\ Transmitted = 1696$$

$$Throughput = \frac{5000 * 1448}{1696}\ Kbps = 4268\ Kbps.$$

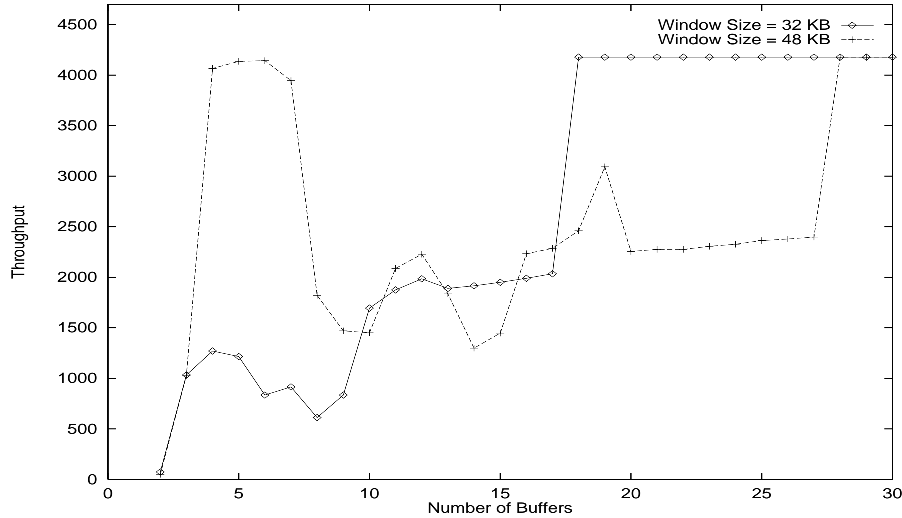

Figure 4.2 Effect of varying number of buffers on throughput. Transmission rate is set to 5000 Kbps and segment size is 1448 bytes.

Though maximum theoretical TCP throughput is 4268 Kbps, obtained throughput may be further reduced when the available bandwidth must be shared among competing connections. Figure 4.1 shows a comparison of the nominal ATM bit rate, the theoretical maximum TCP throughput rate, and the observed TCP througput as a function of nominal ATM bit rate. The observed rate is close to but slightly less than the theoretical maximum.

4.2 Variation of number of Router Buffers

In this section we describe the effect on TCP throughput of varying the number of ATM transmit buffers in *RouterPC*. Intuitively, we expect maximum throughput when the buffer count is high enough that buffers are not overrun and packets are not dropped. When the buffer count drops below a threshold value, the throughput should start decreasing until it becomes low at small buffer counts. However, in a practical scenario, we observe interesting behavior which deviates from what we expect.

The experiment is performed with data being transferred between the sender, *chattooga*, and the receiver, *bart*. The transmission rate on *RouterPC* is set to 5000 Kbps. Since, *chattooga* can transmit at a higher rate than 5000 Kbps on the 10 Mbps ethernet, the packets queue in *RouterPC*. Once the maximum queue length is reached, each new arrival to the queue is discarded.

The throughput is measured while doing a bulk transfer using the program, `atcpgen`, an application that establishes a connection to the destination and transmits 50 MB of data. The receiver runs the program, `atcpgobl` which consumes the data. As mentioned earlier, a feature of these programs is that they are independent of file system.

The data is collected for 2 window sizes: a default window size of 32 KB and a large window size of approximately 48 KB. The graph of throughput as a function of buffer count is shown in Figure 4.2. The maximum throughput obtained is 4178 Kbps for both window sizes. This value is 98.5% of the theoretical limit and is consistent with values reported by Flower et al. in [3].

For a window size of 48 KB, *RouterPC* starts dropping packets when the buffer count is less than 28 buffers (28 * 1448 = 40544 bytes). For the window size of 32 KB, *RouterPC* starts dropping when buffer count is less than 18. The buffer count at which *RouterPC* starts dropping packets is a function of both the offered window size and the transmit speed on *RouterPC's* ATM link. The queue length at *RouterPC* can clearly never exceed the size of the offered window. However, it is further constrained by the degree to which the *RouterPC* serves as the "bottleneck" device in the closed transmit/ack feedback loop. The graph in Figure 4.3 shows the observed queue length as a function of window size for a transmission rate of 5000 Kbps. The queue length increases linearly with increasing window size. The slope of the line is 0.84 indicating that buffer space in the amount of 84% of the offered window is sufficient to avoid drops. All of the preceding analysis is, of course, dependent upon the assumption that the sender transmits at fixed rate when not blocked.

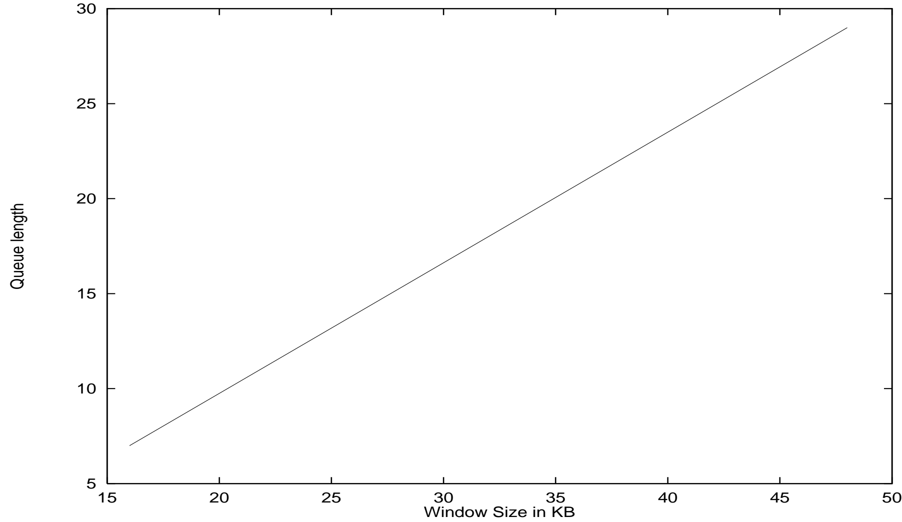

Figure 4.3 Queue length for different window sizes. Transmit buffer count is set to maximum (value of 63). Transmission rate is set to 5000 Kbps and segment size is 1448 bytes.

For a window size of 48 KB, when the buffer count is reduced from 28 to 27, *RouterPC* starts dropping packets and the throughput reduces from 4178 Kbps to 2398 Kbps, a drop of about 42%. For a 32 KB window also, we observe the same effect when the buffer count is reduced from 18 to 17. There is drop from 4178 Kbps to 2035 Kbps, a performance loss of approximately 50%.

An unexpected phenomenon occurs when the buffer count is reduced from 8 to 7. Throughput experiences a dramatic *increase*. The effect is more prominent with a 48 KB window size where throughput suddenly shoots up from 1821 Kbps to 3946 Kbps.

Table 4.1 shows the drop rate and throughput for different buffer counts. For a buffer count of 8, TCP is sending at a rate which produces a drop rate of 11 per second (on *RouterPC*) and throughput of 1820 Kbps. However, when the buffer count is reduced to 7, the throughput suddenly increases to 3950 Kbps. Counterintuitively, the drop rate also increases to 37 packet drops per second. We can infer that TCP

Buffer Count	*Throughput (Kbps)*	*Drop rate (packets/second)*
10	1450	7
9	1470	9
8	1820	11
7	3950	37
6	4140	22

Table 4.1 Variation of Throughput and Drop rate with respect to buffer count

is going into some type of aggressive state in which it retransmits at a fast rate and thus recovers from the dropped packets. In the next section, we further investigate this behavior and explain the sequence of events that induce it.

4.2.1 Analysis and Explanation of anomalous TCP behavior

The following is a portion of TCPDUMP [1] output taken on *chattooga* when the buffer count on *RouterPC* is 7.

```
 1 chattooga.cs.clemson.edu.1085  P  5289544:5290992
 2 bart-atm.33456                 .  ack 5279408
 3 chattooga.cs.clemson.edu.1085  P  5290992:5292440
 4 chattooga.cs.clemson.edu.1085  P  5292440:5293888
 5 chattooga.cs.clemson.edu.1085  P  5293888:5295336
 6 bart-atm.33456                 .  ack 5282304
 7 chattooga.cs.clemson.edu.1085  P  5295336:5296784
 8 chattooga.cs.clemson.edu.1085  P  5296784:5298232
 9 bart-atm.33456                 .  ack 5282304
10 bart-atm.33456                 .  ack 5282304
11 bart-atm.33456                 .  ack 5282304
12 chattooga.cs.clemson.edu.1085  P  5282304:5283752
13 bart-atm.33456                 .  ack 5282304
14 bart-atm.33456                 .  ack 5282304
15 bart-atm.33456                 .  ack 5282304
16 bart-atm.33456                 .  ack 5282304
17 chattooga.cs.clemson.edu.1085  P  5298232:5299680
18 bart-atm.33456                 .  ack 5282304
```

[1] TCPDUMP is network tool used for capturing and viewing packets on a network

```
19 chattooga.cs.clemson.edu.1085  P  5299680:5301128
20 bart-atm.33456                 .  ack 5289544
21 chattooga.cs.clemson.edu.1085  P  5289544:5290992
22 chattooga.cs.clemson.edu.1085  P  5301128:5302576
23 chattooga.cs.clemson.edu.1085  P  5302576:5304024
24 chattooga.cs.clemson.edu.1085  P  5304024:5305472
25 bart-atm.33456                 .  ack 5289544
26 chattooga.cs.clemson.edu.1085  P  5305472:5306920
```

In line 8, *chattooga* transmits a packet with sequence number 5296784 to *bart*. *chattooga* receives 3 duplicate ACK 5282304 [2] in lines 9-11. It retransmits the packet with sequence number 5282304 in line 12 and continues with transmission of new data with sequence number 5298232 in line 17. This is exactly the fast retransmit algorithm that we explained in section 2.1.1. TCP goes into fast retransmit phase and calls `tcp_fast_retrans()`. It gets an ACK 5289544 in line 20. Immediately, in line 21, it retransmits this packet and then continues transmitting new data from line 22. This is an example of a new performance optimization strategy which suggests that receipt of a *partial ACK* while in fast retransmit should trigger an immediate retransmission of the packet identified in the ACK. A call to `clear_fast_retransmit()` ends the fast retransmit phase and starts congestion avoidance.

The following is a portion of TCPDUMP output taken on *chattooga* when the buffer count on *RouterPC* is 8.

```
1   bart-atm.33456                 . ack 761601
2   chattooga.cs.clemson.edu.1087  P 761601:763049
3   bart-atm.33456                 . ack 761601
4   chattooga.cs.clemson.edu.1087  P 778977:780425
5   chattooga.cs.clemson.edu.1087  P 780425:781873
6   chattooga.cs.clemson.edu.1087  P 781873:783321
7   chattooga.cs.clemson.edu.1087  P 783321:784769
8   chattooga.cs.clemson.edu.1087  P 784769:786217
9   chattooga.cs.clemson.edu.1087  P 786217:787665
10  chattooga.cs.clemson.edu.1087  P 787665:789113
```

[2] Sequence number of the next expected byte

```
11 bart-atm.33456                  . ack 761601
12 chattooga.cs.clemson.edu.1087  P 789113:790561
13 bart-atm.33456                  . ack 761601
14 bart-atm.33456                  . ack 761601
15 bart-atm.33456                  . ack 761601
16 bart-atm.33456                  . ack 761601
17 bart-atm.33456                  . ack 761601
18 bart-atm.33456                  . ack 764497
19 chattooga.cs.clemson.edu.1087  P 764497:765945
20 bart-atm.33456                  . ack 764497
21 bart-atm.33456                  . ack 764497
22 bart-atm.33456                  . ack 764497
23 bart-atm.33456                  . ack 764497
24 bart-atm.33456                  . ack 764497
25 bart-atm.33456                  . ack 764497
26 bart-atm.33456                  . ack 764497
27 bart-atm.33456                  . ack 777529 win 13032
28 bart-atm.33456                  . ack 777529 win 43440
29 chattooga.cs.clemson.edu.1087  P 777529:778977
30 bart-atm.33456                  . ack 780425
31 chattooga.cs.clemson.edu.1087  P 780425:781873
32 bart-atm.33456                  . ack 781873
33 chattooga.cs.clemson.edu.1087  P 781873:783321
34 bart-atm.33456                  . ack 783321
35 chattooga.cs.clemson.edu.1087  P 783321:784769
36 bart-atm.33456                  . ack 784769
37 chattooga.cs.clemson.edu.1087  P 784769:786217
38 bart-atm.33456                  . ack 786217
39 chattooga.cs.clemson.edu.1087  P 786217:787665
40 bart-atm.33456                  . ack 787665
41 chattooga.cs.clemson.edu.1087  P 787665:789113
42 bart-atm.33456                  . ack 789113
43 chattooga.cs.clemson.edu.1087  P 789113:790561
44 bart-atm.33456                  . ack 790561
45 chattooga.cs.clemson.edu.1087  P 790561:792009
46 bart-atm.33456                  . ack 792009
47 chattooga.cs.clemson.edu.1087  P 792009:793457
48 chattooga.cs.clemson.edu.1087  P 793457:794905
```

In contrast to the previous case, we observe that there is no fast retransmission and TCP waits for a timeout when packets are lost and then retransmits them. In line 28, *chattooga* gets an ACK 777529 at time `19:41:36.228633`. It waits for a timeout (by calling function `tcp_time_wait()`) and retransmits the packet (using the

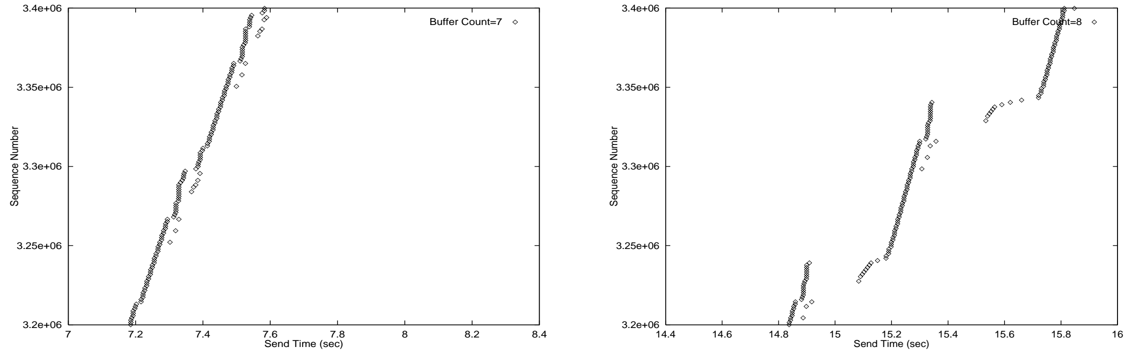

Figure 4.4 Sequence number versus time with buffer counts of 7 and 8.

function `tcp_xmit_retransmit_queue()`) at time `19:41:36.453751`, 225 ms later in line 29. From line 31 to 46, it retransmits all the successive packets waiting for ACKs for each.

A higher level and perhaps more intuitively appealing comparison of behavior of the two transmissions can be observed from the Figure 4.4 which plots sequence number versus time. A timeout followed by a slow start appears as a horizontal gap in the graph. For 8 buffers, two of these events are apparent in the graph. For 7 buffers, TCP does not timeout. As we described earlier, it does the fast retransmit and recovery.

This analysis is confirmed by comparing the behavior of the congestion window in the two transmissions. This graph is shown in Figure 4.5. For 8 buffers, after the timeout, *cwnd* is set to 1 and slow start is performed.

The foregoing explanation clearly shows that the performance anomaly is a result of repeated slow starts with 8 buffers. However, the TCPDUMP output and the associated graphs do not give us adequate information to determine the *cause* of the repeated slow starts. In order to determine this cause it was ultimately necessary to instrument the TCP running on the receiver *bart*.

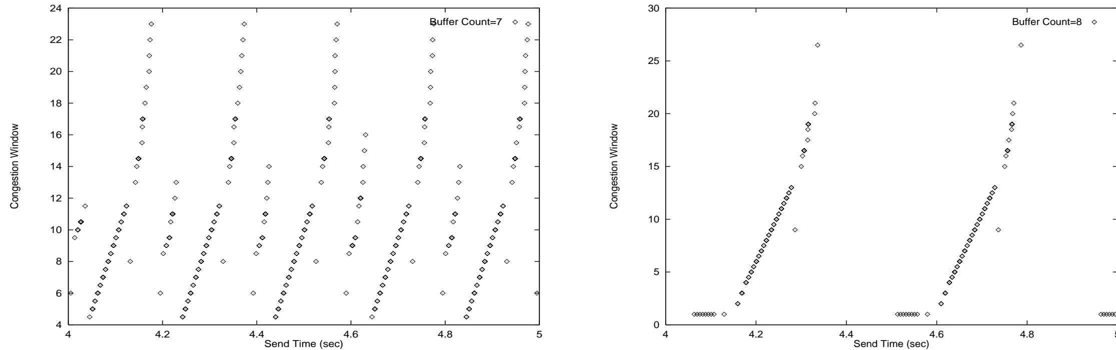

Figure 4.5 Variation of congestion window with time for 7 and 8 buffers

---

Nevertheless, an indicator of the problem is apparent in lines 29 through 48 of the 8 buffer TCPDUMP output. In those lines *chattooga* retransmits 10 packets in sequence and receives an ack for each one. This behavior would indicate that a complete block of 10 packets had somehow been lost. Since we felt it unlikely that the network could drop 10 consecutive packets, our suspicions turned to *bart*.

When a packet arrives, the following functions are executed:

```
tcp_v4_rcv() --> tcp_v4_do_rcv() --> tcp_rcv_established()
--> tcp_data()
```

In function `tcp_data()`, there is a check to see if the buffers are full:

```
if (atomic_read(&sk->rmem_alloc) > sk->rcvbuf)
        prune_queue(sk);
```

If the size of all allocated buffers exceeds the the receive buffer limit (normally twice the offered window size), `prune_queue()` is called and it removes all the unacked packets in the queue.

Disabling the flushing function improves performance

In order to verify that `prune_queue()` was indeed the source of the problem, we disabled it by forcing it to return without dropping any the packets. After this

modification throughput dropped no lower than 4101 Kbps across the entire range of buffer sizes tested. We even tried dropping 4 successive packets when queue length limit was reached. The throughput dropped marginally to 3987 Kbps. This is a significant improvement over the original performance, and demonstrates that Reno TCP actually works as expected if this anomaly is corrected.

Underlying causes of the anomaly

The final elements of the investigation were twofold. First we attempted to determine why certain buffer counts triggered numerous calls to `prune_queue()` but others did not. Second, we hoped to discover why calls to `prune_queue` lead to sender to time out and slow start even though dropping four successive packets in the network does not. We were successful in answering the first question, but the second remains open.

In order to gain insight into the problem it was necessary to instrument the TCP on *bart*. Just before `prune_queue` is called, if the allocated buffer exceeds 3/4ths the receive buffer size, we print the allocated buffer size and the receive buffer size. In linux, the size of the receive buffer is actually twice the offered window size (98000 when the window size is set to 49000).

```
[/usr/src/linux/net/ipv4/tcp_input.c]
   if (sk->rmem_alloc > (( 3 * sk->rcvbuf) / 4))
     printk("Alloc = %d Buf = %d \n", sk->rmem_alloc, sk->rcvbuf);
```

The following kernel log files show that `prune_queue` is regularly called when 8 transmit buffers were used in *RouterPC*. The complete logs also show that it was very rarely called with 7 and never called with 6 buffers.

Case 1: 8 buffers:

```
Mar  4 17:17:30 bart kernel: Alloc = 94080, Buf = 98000
Mar  4 17:17:30 bart kernel: Alloc = 103488, Buf = 98000
Mar  4 17:17:30 bart kernel: tcp_data: EXECUTING PRUNE_QUEUE
Mar  4 17:17:30 bart kernel: Alloc = 103488, Buf = 98000
```

```
Mar  4 17:17:30 bart kernel: tcp_data: EXECUTING PRUNE_QUEUE
Mar  4 17:17:30 bart kernel: Alloc = 103488, Buf = 98000
Mar  4 17:17:30 bart kernel: tcp_data: EXECUTING PRUNE_QUEUE
Mar  4 17:17:30 bart kernel: Alloc = 103488, Buf = 98000
Mar  4 17:17:30 bart kernel: tcp_data: EXECUTING PRUNE_QUEUE
Mar  4 17:17:30 bart kernel: Alloc = 94080, Buf = 98000
```

Case 2: 7 buffers:

```
Mar  4 17:19:16 bart kernel: Alloc = 75264, Buf = 98000
Mar  4 17:19:16 bart kernel: Alloc = 84672, Buf = 98000
Mar  4 17:19:16 bart kernel: Alloc = 94080, Buf = 98000
Mar  4 17:19:16 bart kernel: Alloc = 75264, Buf = 98000
Mar  4 17:19:16 bart kernel: Alloc = 84672, Buf = 98000
Mar  4 17:19:16 bart kernel: Alloc = 94080, Buf = 98000
Mar  4 17:19:16 bart kernel: Alloc = 75264, Buf = 98000
Mar  4 17:19:16 bart kernel: Alloc = 84672, Buf = 98000
```

The key to understanding the problem lies in the distinction between the following two `skb` fields:

- Allocated size for each packet, `skb->truesize` = 9408 bytes
- Actual length of the data, `skb->len` = 1448 bytes

The reason for this difference in allocated size and actual data length is that the ATM device driver defines receive buffer size as 9400 and allocates this size for each input socket buffer. This is done for compatibility with Classical IP over ATM standards.

```
[atmdd.h]: #define RB_SIZE   (9400)
[atmrecv.c]: skb = alloc_skb(RB_SIZE, GFP_ATOMIC);
```

Thus *bart* is "billed" for a full 9400 byte segment at each arrival even though only 1448 bytes are being sent. The arrival of an 11th packet to the buffer causes the allocated buffer space to exceed 98000 and triggers a call to `prune_queue`. Figure 4.6 demonstrates this sequence of events occurring in time.

- Time (1): The buffers in *RouterPC* are filled with packets numbers 20 to 27. Packet 28 arrives but since the queue is full, the packet is dropped. The packets in the buffer are transmitted and they reach *bart*.

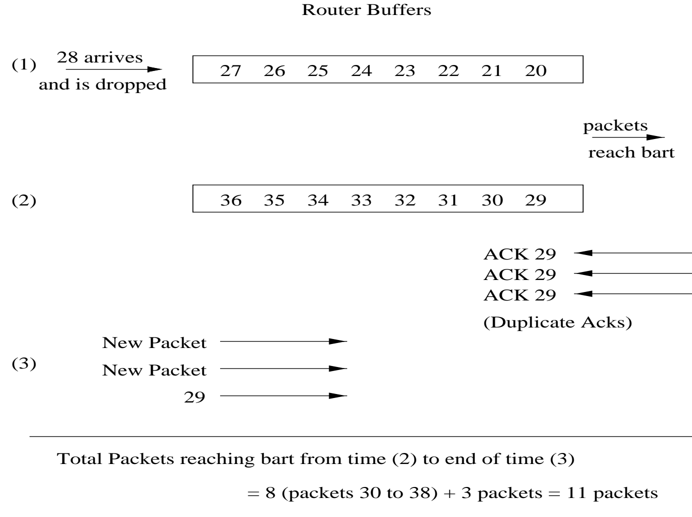

Figure 4.6 Triggering the flushing of packets in *bart*.

- Time (2): *RouterPC* buffers are filled with packets from 29 to 36. For our discussion, it does not make a difference if any packets are dropped after 36. Since packet 28 is missing, *bart* sends duplicate ACKs for 28.

- Time (3): *chattooga* sends 3 new packets on arrival of duplicate ACKs in addition to retransmitting the packet.

Totally buffer in *bart* has 11 packets. Since the driver allocates 9408 bytes for each packet, the total space occupied is equal to 103488 bytes whereas available space is 98000 bytes. This is the reason for overflowing the buffer space on *bart*. For the case of 7 buffers, the space occupied is 7 + 3 new packets = 10 packets = 94080 bytes. Hence, buffers are not overrun.

This discussion clearly demonstrates that queue pruning will occur in *bart* whenever *RouterPC* has 8 or more ATM transmit buffers. It does not, however, explain the slow start. Examination of the kernel logs on *bart* clearly show that blocks of up to 10 consecutive packets get pruned and this leads to the slow start. One possible explanation for the mass pruning is the loss of a retransmitted packet. However, the loss of a retransmitted packet should be more likely with 7 buffers on *RouterPC* than

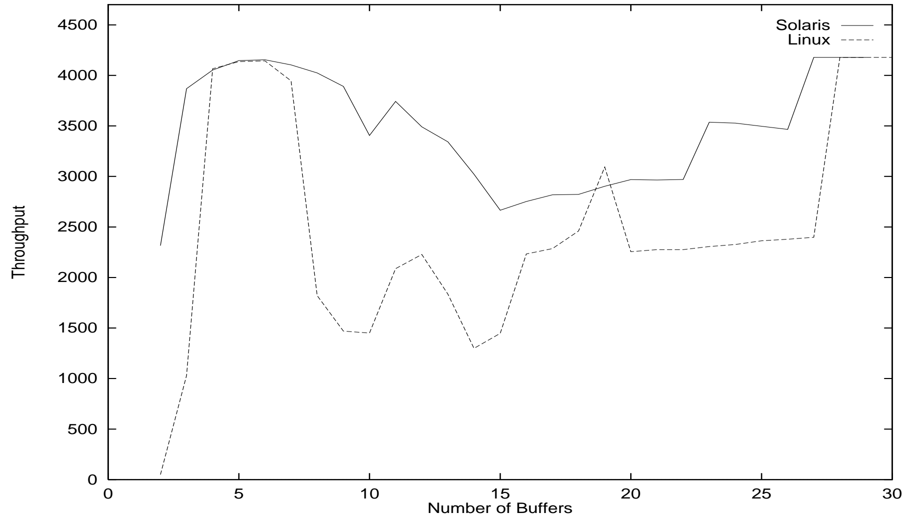

Figure 4.7 Effect of varying number of buffers on throughput for Solaris. ATM card transmission rate is 5000 Kbps, window size is 48 KB and segment size is 1448 bytes.

---

with 8, and the TCPDUMP logs fail to show any packet being transmitted three times in either configuration. Thus, unfortunately this problem remains open.

4.3 Solaris-TCP Investigation

We conducted an experiment to compare the performance difference between implementations of TCP in Solaris and in Linux. We used an Intel machine running Solaris 2.6 (PC version) and ran the sender application on this machine, instead of *chattooga*. A window size was set to 48 KB since the anomalous behavior noted earlier was more prominent at 48 KB window size. From Figure 4.7, we can make the following observations:

1. The maximum throughput obtained for large buffer counts in both the cases are same(4178 Kbps).

2. Performance in Solaris doesn't degrade too badly when packets are dropped. We observe that the throughput drops from 4178 Kbps to only 3465 Kbps as compared to Linux which dropped from 4178 Kbps to 2398 Kbps.

3. As observed in Linux implementation, Solaris also does a fast retransmission when the buffer count is 7 and throughput increases. Looking at the overall graph, Solaris TCP performs better than Linux and the throughput doesn't drop below 2666 Kbps for any buffer count where as throughput in Linux fluctuates and goes extremely low in some cases.

Chapter 5
Concluding Remarks and Future Research

In this paper we described the architecture and implementation of the TIGER testbed for network performance studies. TIGER is a flexible testbed designed to support a variety of experiments, and we demonstrated its use in our study of the effects of limiting the number of intermediate router buffers on the performance of TCP. We demonstrated that buffer constraints can lead to unexpected behavior and explained some aspects of the anomalous behavior that occurred at buffer counts of 7 and 8. We also showed the effect of variation of window size and transmission rate on throughput and queue length. Some other research areas that can be investigated with TIGER are identified in the following discussion.

5.1 Selective Acknowledgments (SACKs)

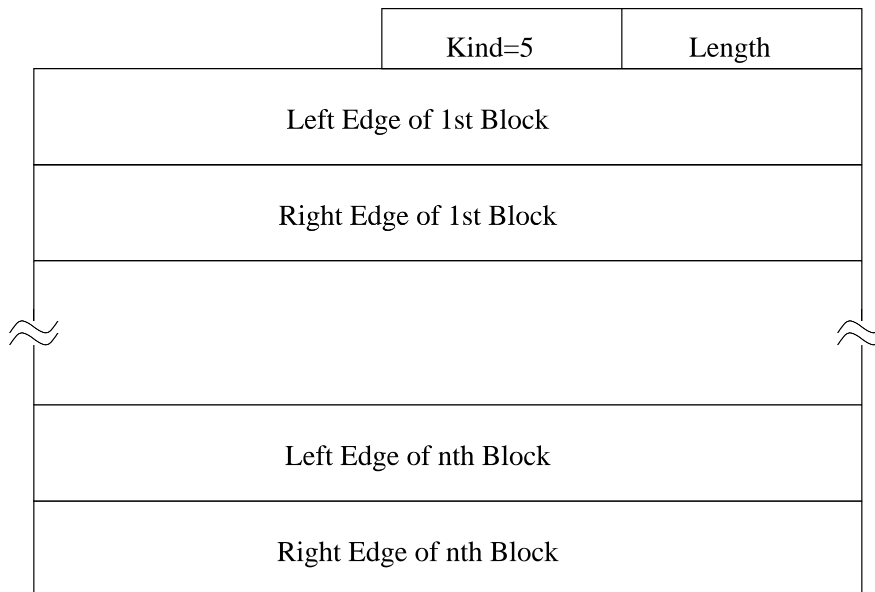

Figure 5.1 TCP SACK Options

Multiple packet losses from a window of data can have a catastrophic effect on TCP throughput. Selective Acknowledgment (SACK) is a strategy which corrects this behavior in case of multiple dropped segments. With selective acknowledgments, the data receiver informs the sender about all segments that have arrived successfully. So the sender retransmits only the segments that have actually been lost. The SACK option, as shown in Figure 5.1, is used to convey extended acknowledgment information from the receiver to the sender over an established TCP connection. In Linux, SACKs are enabled by default. It will be an interesting project to see the performance loss by disabling SACKs. The *RouterPC* can be configured to drop multiple packets in a window to study its effect.

5.2 Validated Performance Modeling

A key to the design of high-performance networks is the ability to model and estimate performance parameters. The designer needs to be able to estimate future traffic volume and characteristics on the basis of observed traffic. The statistical characteristics of a traffic flow affect a wide variety of design and configuration issues, including routing protocols, queuing disciplines used at routers and ATM switches, and buffer sizes. Padhye et al.[8] present a model for TCP throughput and its validation. Analytic models can be validated using the results obtained by actual experimentation on the TIGER testbed. For example, an analytic model to characterize the steady state throughput of a bulk transfer TCP flow as a function of loss rate and round trip time could be validated on TIGER.

5.3 Self-Similar Traffic Models

A number of recent studies [2, 9] indicate that much of the traffic on high-speed networks does not exhibit the random character required for the queuing theory equations to be valid. Rather, such traffic has a self-similar, or fractal, character. The question arises as to how prevalent such traffic patterns are and under what

conditions performance analysis is critically dependent on taking self-similarity into account. A study involving the traffic types such as Poisson, exponential, and self-similar can be readily carried out on the TIGER testbed by suitably configuring the transmitting processes. The transmit rate on *RouterPC* can also be dynamically varied to simulate the effect of competing sources of varying intensity.

References

1. Almesberger Werner, ATM on Linux - The 3rd year. 4th International Linux Kongress, Würzburg.

2. Crovella M., and Vestavros A. Self-Similarity in World-Wide Web Traffic: Evidence and Possible Causes. *Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1994.

3. Flower Alice, Integration and Performance Testing of IP over ATM, Clemson University, 1998.

4. IBM redbooks - Campus Network Configuration, *http://www.redbooks.ibm.com*

5. Matthew M, Mahdavi Jamshid, Floyd Sally, Romanow Allyn, TCP Selective Acknowledgment options, Request for Comments 2018, October 1996.

6. Jacobson V, Congestion Avoidance and Control. *Proceedings of ACM SIGCOMM '88.* August, 1988.

7. Mankin Allison, Random Drop Congestion Control. *Proceedings of ACM SIGCOMM '90.* 1990.

8. Padhye Jitendra, Firoiu Victor, Towsley Don, Kurose Jim, Modeling TCP Throughput: A Simple Model and its Empirical Validation. *Proceedings of ACM SIGCOMM '98.* 1998.

9. Leland W, Taqqu M, Willinger W, Wilson D. On the Self-Similar Nature of Ethernet Traffic (Extended Version).*IEEE/ACM Transactions on Networking*, February 1994.

10. Paxson V, End-to-End Internet Packet Dynamics. *Proceedings of ACM SIGCOMM '97.* September, 1997.

11. Romanow Allyn, Floyd Sally, Dynamics of TCP traffic over ATM Networks, *IEEE JSAC*, V.13 N.4, May 1995, p. 633-641.

12. Semke Jeffrey, Mahdavi Jamshid, Matthew Mathis, Automatic TCP Buffer Tuning. *ACM SIGCOMM '98.*

13. Stevens W, *TCP/IP Illustrated, Vol I The Protocols* Addison-Wesley, 1994.

14. Stevens W, TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, Request for Comments 2001, March 1996.

15. Westall J.M, Geist R.M., Validated Modeling of Network Component Performance, Proposal to IBM, Research Triangle Park, NC, 1998.

16. Westall, J.M., Geist, R.M., Bringing the High End to the Low End: High Performance Device Drivers for Linux PC. In *Proceedings 36th Annual ACM Southeast Conference.* ACM, 1998.

17. Westall, J.M., Geist, R.M., Flower, A.L., ATM Device Driver Development in Linux, Clemson University, 1998.

APPENDIX A
Modification to TCP on chattooga

A.1 Header file

A 10000 element array of the structure, _seqdata, is used for capturing data from packets. start_flag is used for enabling and disabling the capture. seq_count is for counting 10000 packets.

```
[/usr/src/linux/our_syscalls/seqdata.h]

/* SEQDATA */
#ifndef _SEQDATA_H
#define _SEQDATA_H
#include<linux/time.h>

typedef struct _sqdata {
        unsigned long seq;
        struct timeval time;
        unsigned long cwnd;
        unsigned long ssthresh;
        unsigned long ack;
        int flight;
} _seqdata;

extern unsigned short int seqdata_port;
extern int start_flag;
extern unsigned long seq_count;
extern _seqdata seqdata[10000];

#endif //_SEQDATA_H
```

A.2 Copy from kernel to user space

This routine copies kernel data to user space using the function copy_to_user(). Capturing is disabled since data is copied to user space.

```
[/usr/src/linux/our_syscalls/get_seqdata.c]

# include "seqdata.h"
#include <asm/uaccess.h>

void sys_get_seqdata(_seqdata *sq)
{
        start_flag = 0;
        copy_to_user(sq, &seqdata[0], sizeof(seqdata));
}
```

A.3 Reset kernel values

This function sets the value in the array to zero and port number to the argument of the function. Capturing is enabled.

```
[/usr/src/linux/our_syscalls/set_seqdata.c]

# include "seqdata.h"

void sys_set_seqdata(unsigned short int portnum)
{
        int i;
        seqdata_port = portnum;
        for(i=0;i<10000;i++)
        {
                seqdata[i].seq=0;
                seqdata[i].time.tv_sec=0;
                seqdata[i].time.tv_usec=0;
                seqdata[i].cwnd=0;
                seqdata[i].ssthresh=0;
                seqdata[i].ack=0;
                seqdata[i].flight=0;
        }
        seq_count =0;
        start_flag = 1;
}
```

A.4 Makefile

An entry is also needed in the files `/usr/src/linux/arch/i386/kernel/entry.S` and `/usr/src/linux/include/asm/unistd.h` for the system call.

```
[/usr/src/linux/our_syscalls/Makefile]

OBJS = set_seqdata.o get_seqdata.o

all: our_syscalls.o

our_syscalls.o: $(OBJS)
        ld -r -o our_syscalls.o $(OBJS)
        sync
dep:
        sync
fastdep:
        sync
```

A.5 Capturing the data

The data is captured in the output sending routing `tcp_transmit_skb()`. The `if` condition checks if `start_flag` is set, if count is less than 10000 and if port numbers are same. The code, that is added, begins with the comment, `// SEQDATA --->` and ends with `// <--- SEQDATA`

```
[/usr/src/linux/net/ipv4/tcp_output.c]

//SEQDATA --> INSERTED AT LINE 64 in original tcp_ouput.c
#include <linux/time.h>
#include "/usr/src/linux/our_syscalls/seqdata.h"

_seqdata seqdata[10000];
unsigned short int seqdata_port = 33456; // default is 33456
int start_flag = 1;
unsigned long seq_count = 0;
int p_flight=0;
// <-- SEQDATA

void tcp_transmit_skb(struct sock *sk, struct sk_buff *skb)
{
   if(skb != NULL) {
      struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
      struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);
      int tcp_header_size = tp->tcp_header_len;
      struct tcphdr *th;

      if(tcb->flags & TCPCB_FLAG_SYN) {
           tcp_header_size = sizeof(struct tcphdr) + TCPOLEN_MSS;
           if(sysctl_tcp_timestamps)
                tcp_header_size += TCPOLEN_TSTAMP_ALIGNED;
           if(sysctl_tcp_window_scaling)
                tcp_header_size += TCPOLEN_WSCALE_ALIGNED;
           if(sysctl_tcp_sack && !sysctl_tcp_timestamps)
                tcp_header_size += TCPOLEN_SACKPERM_ALIGNED;
      } else if(tp->sack_ok && tp->num_sacks) {
      /* A SACK is 2 pad bytes, a 2 byte header, plus
      * 2 32-bit sequence numbers for each SACK block.
      */
           tcp_header_size += (TCPOLEN_SACK_BASE_ALIGNED +
                           (tp->num_sacks * TCPOLEN_SACK_PERBLOCK));
      }
      th = (struct tcphdr *) skb_push(skb, tcp_header_size);
      skb->h.th = th;
      skb_set_owner_w(skb, sk);
```

```
/* Build TCP header and checksum it. */
th->source              = sk->sport;
th->dest                = sk->dport;
th->seq                 = htonl(TCP_SKB_CB(skb)->seq);

// SEQDATA --> INSERTED AT LINE 107 in original tcp_output.c

if( start_flag == 1 && seqdata_port == th->dest
                                        && seq_count < 10000)
{
     seqdata[seq_count].seq  = ntohl(th->seq);
     seqdata[seq_count].cwnd = tp->snd_cwnd;
     seqdata[seq_count].ssthresh = tp->snd_ssthresh;
     seqdata[seq_count].ack = tp->snd_una;
     seqdata[seq_count].flight = tcp_packets_in_flight(tp);
     do_gettimeofday(&seqdata[seq_count++].time);
}
          // <-- SEQDATA
```

A.6 User level get program

This user level program gets the value from the kernel using the system call `get_seqdata()` and prints the data in column format.

```
[/home/sridatta/seq/get_data.c]

#include "/usr/src/linux/include/asm/unistd.h"
#include <linux/time.h>

typedef struct _sqdata1 {
        unsigned long seq;
        struct timeval time;
        unsigned long cwnd;
        unsigned long ssthresh;
        unsigned long ack;
        int flight;
} _seqdata1;

int errno;
_syscall1(void, get_seqdata, _seqdata1* , data)

main(int argc, char **argv)
{
        _seqdata1 data[10000];
```

```
        int i;
        unsigned long start_seconds;
        unsigned long start_seqnum;
        get_seqdata(&data[0]);

        start_seconds=data[0].time.tv_sec;
        start_seqnum=data[0].seq;
        printf("%f %lu %lu %lu %lu %d\n",
            (float)((float)data[0].time.tv_usec/1000000),
            data[0].seq-data[0].seq, data[0].cwnd,
            data[0].ssthresh, data[0].ack, data[0].flight);

        for (i=1 ; i<10000 ; i++)
        {
             data[i].time.tv_sec-=start_seconds;
             printf("%f %lu %lu %lu %lu %d\n",
                 (float)((float)data[i].time.tv_sec+
                 ((float)data[i].time.tv_usec/1000000)),
                 data[i].seq-start_seqnum, data[i].cwnd,
                 data[i].ssthresh, data[i].ack, data[i].flight);
        }
}
```

A.7 User level reset program

```
[/home/sridatta/seq/reset.c]

#include "/usr/src/linux/include/asm/unistd.h"
#include <netdb.h>
#include <netinet/in.h>
#include <sys/types.h>

int errno;
_syscall1(void, set_seqdata, unsigned short int, portnum)

main(int argc, char **argv)
{
        unsigned short int p=htons(33456);
        if(argc == 1)
                p=(htons(atoi(argv[0])));
        set_seqdata(p);
}
```

APPENDIX B
Barnes Lab 8285 configuration

B.1 Device Specification

```
8285> show device
8285 Nways ATM Workgroup Switch
Name : 8285
Location :

For assistance contact :
Sridatta, Ishraq (656-5866, 656-2841)

Manufacture id: VIM
Part Number: 51H4119 EC Level: E59245
Serial Number: 8269
Boot EEPROM version: v.1.5.2
Flash EEPROM version: v.1.5.2
Flash EEPROM backup version: v.1.3.0
Last Restart : 16:16:41 Sun 27 Sep 98 (Restart Count: 19)

A-8285
-----------------------------------------------------------------------------
 ATM address: 47.02.03.04.05.06.07.08.09.00.00.02.01.99.99.99.99.99.99.01

 > Subnet atm: Up
 IP address: 130.127.201.35. Subnet mask: FF.FF.FF.00

 > Subnet lan emulation ethernet/802.3
   Not Started
   Name :""
   MAC Address: 000000000000
   IP address : 0.0.0.0. Subnet mask: 00.00.00.00
   ATM address    :47.02.03.04.05.06.07.08.09.00.00.02.01.99.99.99.99.99.99.00
   Config LES addr:none
   Actual LES addr:00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00
   BUS ATM address:00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00
   Config LECS add:none
   Actual LECS add:00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00
   LEC Identifier: 0. Maximum Transmission Unit: 0
 > Subnet lan emulation token ring
   Not Started
   Name :""
   MAC Address: 000000000000
   IP address : 0.0.0.0. Subnet mask: 00.00.00.00
```

```
   ATM address    :47.02.03.04.05.06.07.08.09.00.00.02.01.99.99.99.99.99.01
   Config LES addr:none
   Actual LES addr:00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00
   BUS ATM address:00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00
   Config LECS add:none
   Actual LECS add:00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00
   LEC Identifier: 0. Maximum Transmission Unit: 0

Default Gateway : OK
----------------------------------------------------------------------------
 IP address: 130.127.201.3

ARP Server:
----------------------------------------------------------------------------
 ATM address: 47.02.03.04.05.06.07.08.09.00.00.01.01.11.11.11.11.11.11.11

Dynamic RAM size is 8 MB. Migration: off. Diagnostics: disabled.
```

B.2 Port Status

```
8285> show port all

      Type Mode      Status
------------------------------------------------------------------
 1.01:UNI enabled  UP-NO ACTIVITY
 1.02:UNI enabled  UP-NO ACTIVITY
 1.03:UNI enabled  UP-NO ACTIVITY
 1.04:UNI enabled  UP-NO ACTIVITY
 1.05:UNI enabled  UP-NO ACTIVITY
 1.06:UNI enabled  UP-NO ACTIVITY
 1.07:UNI enabled  UP-OKAY
 1.08:UNI enabled  UP-OKAY
 1.09:UNI enabled  UP-NO ACTIVITY
 1.10:UNI enabled  UP-NO ACTIVITY
 1.11:UNI enabled  UP-NO ACTIVITY
 1.12:UNI enabled  UP-NO ACTIVITY
 1.13:NNI enabled  UP-OKAY
```

B.3 Status of port 7: *bart*'s port

```
8285> show port 1.07 verbose

      Type Mode      Status
------------------------------------------------------------------
```

```
 1.07:UNI enabled  UP-OKAY

Signalling Version   : with ILMI
Flow Control         : Off
VPI.VCI range        : 0.63 (0.6 bits)
Connector            : RJ45
Media                : copper twisted pair
Port speed           : 25600 kbps
Remote device is active
IX status            : IX OK
```

B.4 Status of port connected to F-D IBM 8260

```
8285> show port 1.13 verbose

      Type Mode      Status
-------------------------------------------------------------------
 1.13:NNI enabled  UP-OKAY

VPI.VCI range        : 15.1023 (4.10 bits)
Connector            : SC DUPLEX
Media                : multimode fiber
Port speed           : 155000 kbps
Remote device is active
IX status            : IX OK
Logical links indexes: 1

Frame format         : SONET STS-3c
Scrambling mode      : frame and cell
Clock mode           : internal
```

B.5 Static route to F-D IBM 8260

```
8285> show static_route
Index Acn Static route
-----------------------------------------------------------------------------
   1   01 47020304050607080900000 1

63 empty entries.
```

B.6 End System Identifiers (ESI) of registered machines

```
8285> show atm_esi all
```

```
Port   ATM_ESI           Type
-------------------------------------------------------------------------------
 1.07 00.04.AC.6C.2B.7F dynamic
 1.08 00.04.AC.6C.28.EF dynamic
```

APPENDIX C

Acronyms

AAL	ATM Adaptation Layer
APE25	ATM Protocol Engine
ATM	Asynchronous Transfer Mode
CLIP	Classical IP over ATM
ELAN	Emulated Local Area Network
ESI	End System Identifier
FIFO	First In First Out
FTP	File Transfer Protocol
HTTP	HyperText Transport Protocol
IETF	Internet engineering Task force
IISP	Interim Inter-switch Signaling Protocol
ILMI	Integrated Local Management Interface
IP	Internet Protocol
LAN	Local Area Network
LIS	Logical IP Subnetwork
LLC	Logical Link Control
MTU	Maximum Transmission Unit
MSS	Maximum Segment Size
NIC	Network Interface Card
NNI	Network Network Interface
NSAP	Network Service Access Point
PNNI	Private Network-Network Interface
PVC	Permenent Virtual Circuit
QoS	Quality of Service
RFC	Request For Comments
SACK	Selective Acknowledgement
SMTP	Simple Mail Transfer Protocol
SNAP	Subnetwork Attachment Point
SVC	Switched Virtual Circuit
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VBR	Variable Bit Rate
VCI	Virtual Circuit Identifier
VPI	Virtual Path Identifier
WAN	Wide Area Network
WWW	World Wide Web