TM 11

ATM NETWORK PERFORMANCE SIMULATOR

BY

SCOTT ALLEN YENERICH

B.S., University of Illinois, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

To Maria, Mom, and Dad, for all of their love and support

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Professor Benjamin W. Wah, for his advice throughout this project. I would also like to thank Laurence Koehn for his assistance with the NWS application. Finally, I wish to thank Maria Yenerich for her constant support and motivation.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With the inevitable implementation of ATM (Asynchronous Transfer Mode) networks for use by many applications, it is important to understand the ability of a network to handle various traffic loads and, similarly, to understand the effects of various traffic conditions on the performance of applications. It is the purpose of this thesis to present a system which can be used to simulate certain traffic conditions that may be observed by applications. In addition, using a real application, we will examine the performance of a network with ATM-like transmission rates under various traffic loads.

The simulation system presented here is designed as a layer below the application layer. The simulator is passed the message or packet from the sending application. The simulation is performed, the packet is then sent to the simulator at the receiving end. This copy of the simulator forwards the packet to the receiving application.

The simulation itself consists of attempting to send the packet through the ATM-like network in the presence of other traffic. The characteristics of the traffic on the network are created with the use of several variable parameters. The simulation model is presented in Chapter 3. The measured result is the total time required to send the entire packet. This result has two components: the time the packet spends waiting to be sent, and the time it takes to transmit the entire packet.

The second chapter of this thesis provides some basic background on other analyses of traffic multiplexing in ATM networks. The third chapter presents the model used in this thesis. The fourth chapter describes the programs which compose the simulation system. The fifth chapter discusses the results obtained during simulations using a real application to produce the data packets. The final chapter discusses the results of the thesis.

CHAPTER 2

BACKGROUND ON ATM NETWORKS

2.1 ATM Networks

ATM networks are an emerging technology upon which B-ISDNs (Broadband Integrated Services Digital Networks) will be built. ATM networks utilize high-speed transmission rates. They are capable of handling all types of multi-media traffic, such as voice, video, and data.

The basic units of transfer in ATM networks are cells. Cells consist of a 48 byte information field and a five byte header. The cells of all connections are transported on the transmission path, which is composed of virtual paths. Each virtual path is further divided into virtual channels.

Cells belonging to the same connection are transmitted, in order, along the same virtual channel and, therefore, the same virtual path. Cells belonging to different connections, but which have the same network endpoints, are transmitted on different virtual channels within the same virtual path.

This implies that statistical multiplexing may be used to transport the cells from several connections along the same transmission path. ATM networks are regarded as an efficient way to integrate bursty types of heterogeneous traffic [1]. The traffic can include various classes with differences in bit rates and in quality of service (QOS).

2.2 Previous Work

Some work has been done to study cell multiplexing in ATM networks [1]-[6]. The performance has been evaluated using measures such as cell delay, cell loss rate, and cell blocking probability [1], [2], and [5]. The performance of a statistical multiplexer for heterogeneous, bursty traffic in an ATM network has been analyzed [1]-[3].

The performance of a finite queue, multi-server model with multiple output channels has been analyzed [1]. In this model, a Markov-modulated Poisson process was used to model the bursty traffic. The channels were fixed-rate, and the cells were transmitted at discrete-time instances through the channels, after being queued in a finite buffer.

It was found that the performance did not improve linearly with the number of channels [1]. In addition, it was found that performance decreased as burstiness increased, and it was suggested that traffic types of comparable burstiness be divided into groups and that bandwidth be allocated to the groups exclusively.

Other work indicates that when traffic classes with different burstiness characteristics are muliplexed, the class with the highest burstiness experiences the poorest QOS [2]. A third study also suggests that statistical multiplexing approaches can be improved by separating the treatment received by traffic from different classes and by using relatively large buffer sizes for the multiplexers [3].

Another approach uses a Dynamic Time-Slice (DTS) scheme to guarantee a required bandwidth for each traffic class [4]. This model sets up separate queues for different traffic classes. The server cyclically visits each queue for the time slice allocated to that queue.

The time slice allocated to each queue is proportional to the bandwidth required by that queue.

This scheme is dynamic in that it allows for the sharing of the bandwidth by the different traffic classes. The scheme is considered to be more efficient than statistical multiplexing as a way of combining traffic of various types. This work suggests that each traffic class be serviced separately, according to its own requirements.

A weighted round-robin scheme which improves upon an earlier algorithm has also been studied [5]. This algorithm allows different virtual circuits to be visited a different number of times during the server cycles. This scheme was presented in order to improve the smoothness of the multiplexer over the earlier algorithm. The weighted round-robin scheme is an effective and simple method for handling priority traffic.

CHAPTER 3

SIMULATION MODEL

3.1 The Model

The model used in this thesis is shown in Figure 3.1. It includes an infinite queue which stores the packets from the various connections. The packets are segmented into cells and are allocated an output channel on a First-Come, First-Served (FCFS) basis. The cells are sent via the output channels at a rate which is load-dependent.

The incoming packets have an inter-arrival time which is exponentially distributed. This is due to the fact that such bursty data packets can be modeled by a Poisson process [1]. In addition, the size of the packets is also exponentially distributed. The packet size is restricted to a range of 1 KB to 16 KB for purposes of practicality. The mean inter-arrival time and the mean packet size are system parameters.

The users in the simulation model are allocated bandwidth as a group, as has been suggested [1]-[4]. In the simulations shown in Chapter 5, the users are all assumed to be "reporters" from the NWS application, which is described in Chapter 4. Hence, the users all possess the same traffic characteristics and share the bandwidth made available to them.

The model assumes an infinite queue since it has been suggested that large output buffers are needed for bursty data traffic [3]. In addition, the assumption of an infinite queue is justifiable since the application used in the simulations uses a sliding-window protocol for flow control. This will insure that overflow would not be a problem, since adequate buffers could be provided once the level of users was known.

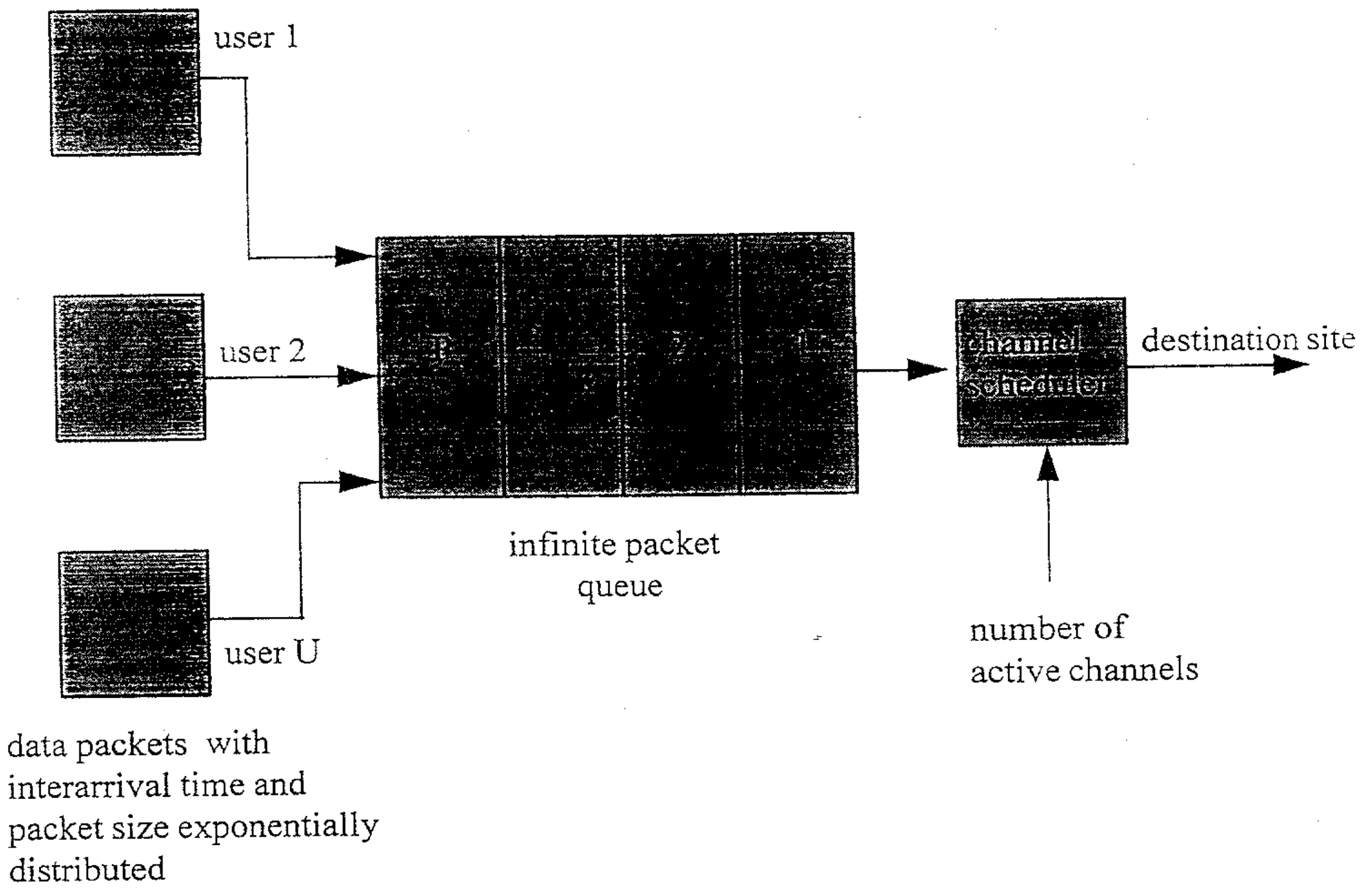
user 1
user 2
user U
infinite packet queue
channel scheduler
destination site
number of active channels
data packets with interarrival time and packet size exponentially distributed

Figure 3.1 The Simulation Model

distributed, but a maximum number of users is established. The parameter should be a floating-point value.

The fourth parameter is the average packet size, which other users on system may send. The size of the packets is exponentially distributed. This parameter has units of kilobytes, and should be a floating-point value. Again, a range is set for the packet size.

The final parameter is a floating-point number which represents the mean inter-arrival time of the packets on the system. This parameter is used to control the level of activity on the system.

CHAPTER 4

THE SIMULATION SYSTEM

4.1 The NWS Application

The application which was used to drive the simulation system is called NWS. It was developed by students at the University of Illinois. The application is a type of news wire service consisting of a news center, reporters, and clients. The reporters submit articles to the news center, which then distributes the articles to subscribing clients.

The application relies upon a program called "RTP" which coordinates the reliable transfer of the articles using Internet sockets. A copy of "RTP" runs at each node and can service multiple processes simultaneously. It is responsible for dividing a message into packets and sending them, and for receiving the packets and reconstructing the original message. "RTP" uses a packet size of one kilobyte.

This application was chosen for the simulation because it provides reasonably good characteristics of bursty data traffic. A good deal of the time the system is idle. However, when an article is submitted, there is a significant amount of traffic produced, in a relatively short amount of time.

4.2 The Interface Module

The first portion of the simulation system is the interface module, named "link.c". This module is shown in Appendix A. It is the interface between the simulation program and the application driving the simulation. This interface consists primarily of a function named "SENDTO" which takes parameters in the exact same format as the Unix function "sendto()". This module is linked with the application at compile time. The application then uses calls to "SENDTO()", rather than "sendto()", to send packets through Internet domain sockets.

The "SENDTO()" parameters include a socket descriptor for an Internet domain socket, a pointer to the packet to be sent, the length of that packet, some control flags, the address to which to send the packet, and the length of that address. When a call is made to "SENDTO", it gets the port number of the Internet domain socket and uses this number to create a unique Unix domain socket for communication with the simulation program. This is done so that multiple applications can use the simulation program simultaneously.

The interface module then sends all pertinent information to the simulation program. This information includes most of the parameters of the "SENDTO" function, the name of the Unix domain socket, and the port number of the Internet domain socket.

Next, the interface module waits for the simulation program to send back the time required to send the packet. This time is then written to a file. Finally, the module returns the size of the packet sent to the application, just as the Unix "sendto()" function does.

4.3 The Simulation Program

The second portion of the simulation system is the simulation program, named "sim1". This program is shown in Appendix A. It is responsible for receiving a packet from the interface, simulating the time required to send the packet, and sending that time back to the interface. In addition, the simulation program also sends the actual packet to the copy of the simulation program running at the receiving end of the application. This copy of the simulation program is responsible for forwarding the packet to the waiting application at the receiving end.

The simulation program sets up a socket in the Unix domain and a socket in the Internet domain. The Unix socket is used to receive packets from the local interface module, and the Internet socket is used to receive packets from remote copies of the simulation program. The local packets must undergo simulation and be sent to remote copies of the simulation program. The remote packets must be delivered to the local "RTP" program.

When a connection is made on the Unix socket, the program reads in the information sent by the interface module. It then "forks" a child process to handle the simulation, while the parent process waits for other connections.

The child calls the simulation module which generates the time to send the packet. Finally, the child sends the actual packet to the remote copy of the simulation program and sends the time to send the packet back to the interface module.

When a connection is made on the Internet socket, the program receives the packet. It then "forks" a child which sends the packet to the local "RTP" program, while the parent waits for other connections.

Some modifications had to made to the "RTP" program to handle the receipt of packets from the simulation program. Originally, "RTP" used the Unix function "recvfrom()" to receive packets from another copy of "RTP" at the sending node. This allowed the receiver to obtain the sender's Internet address. However, since the simulation program was sending the packets to the receiving copy of "RTP", the original sender's address had to be included in the packet. Thus, "RTP" had to be modified to handle packets of a slightly larger size. In addition, the original sender's address must be extracted from the received packet.

4.4 The Simulation Module

The final portion of the simulation system is the simulation module, named "sim2". This module is shown in Appendix A. It is an executable program created with the "CSIM" simulation package. This program requires the size of a packet in bytes and a seed for its random number generator. It then generates the time to send the packet.

The simulation program provides the inputs to the simulation module by writing the packet size and the seed to a file. The module reads the information and creates other traffic on the system. Some of this other traffic has arrived ahead of the real packet, and some of it has arrived after the real packet. The module simulates the sending of all of the traffic on the system. It then writes the time to send the real packet to a file, where it is read by the simulation program.

The characteristics of the created traffic on the system are determined by several parameters which are read from a file named "sim.init". The parameters include the following: 1) the total bandwidth available to any or all of the users, 2) the maximum number of channels available, 3) the average number of users on the system (in addition to the real application), 4) the average packet size sent by the other users, and 5) the average inter-arrival time of the packets on the system.

The simulation module services the packets on a first-come, first-served basis. When a packet arrives, it waits until it can be allocated a channel. The transmission rate of each channel is determined by the number of other users in the system.

Once the allocation of a channel has taken place, the transmission of the packet occurs. The total time to send the packet is the waiting time plus the transmission time.

CHAPTER 5

SYSTEM PERFORMANCE

5.1 Data Collection

The data presented in this chapter was collected using the NWS application running on Sun SPARC 2 workstations running SunOS 4.1.3. Specifically, the simulation times were recorded for packets sent from the "reporter" application to the "news center" application as part of a submitted article. A "dummy" article was chosen so that a single submission resulted in the sending of 50 packets by the "reporter". Each packet was one kilobyte in length.

Each of the figures in Appendix B shows plots of the time required to send an entire packet (in milliseconds) versus the maximum number of active channels, for three different available bandwidth values. Each point is the average of the send times for the 50 individual packets. Each of the eight figures represents a different combination of the average number of users on the system, the inter-arrival times of the packets, and the average packet sizes.

The maximum number of users is limited to 60, and the packet sizes are restricted to the range from 1 KB to 16 KB.

5.2 The Results

Figures B.1, B.3, B.5, and B.7 show plots for an average of 15 users, while Figures B.2, B.4, B.6, and B.8 show plots for an average of 30 users. Figures B.1, B.2, B.5, and B.6 show plots where the mean inter-arrival time for the packets is 2 ms, while Figures B.3, B.4, B.7, and B.8 show plots for a mean inter-arrival time of 1 ms. Finally, Figures B.1, B.2, B.3, and B.4 show plots where the average packet size is 4 KB, while Figures B.5, B.6, B.7, and B.8 show plots where the average is 8 KB.

For the cases where the available bandwidth is 10 Mbits/s, there seems to be significant improvement in the send time when the number of channels is increased. Having more channels will lower the waiting times for some messages, but will increase the waiting times for other messages since the send times will be much slower.

The send time is almost directly proportional to the average number of users on the system for the 10 Mbits/s cases. This seems to be true across all of the inter-arrival time and packet size combinations, especially for the cases involving smaller average packet sizes.

The effects of the inter-arrival times of the packets are more easily seen for the cases where the average packet sizes are smaller. For example, the differences between Figures B.1 and B.3 are greater than the differences between Figures B.6 and B.8.

Increasing the average packet sizes greatly increases the send time, especially when the inter-arrival times are larger. For example, a much greater increase is evident when comparing Figures B.1 and B.5, than when comparing Figures B.4 and B.8.

For the cases where the available bandwidth is 20 Mbits/s, the effects of varying the number of channels are similar to the 10 Mbits/s cases. For example, the effects of increasing the packet sizes are similar to the 10 Mbits/s cases. However, the overall effects of increasing the number of channels are not as noticeable as they were in the 10 Mbits/s cases.

The send times are still proportional to the average number of users on the system, as they were for the 10 Mbits/s cases. The effects of halving the average inter-arrival time seem to be greater than the effects of doubling the number of users for the smaller average packet size cases, while the reverse is true for the large packet size cases.

For the cases where the available bandwidth is 40 Mbits/s, the effects are much less apparent, but the overall trends are similar to the 10 Mbits/s and 20 Mbits/s cases.

CHAPTER 6

CONCLUSIONS

This thesis presented a scheduling scheme for ATM networks and the results of simulating the scheduling scheme using a real application. The performance of the system, with respect to the application used in the simulation, was examined for several combinations of available bandwidth and channels. The effects on packet delay were shown for traffic parameters which included the number of users on the network, the packet inter-arrival time, and the packet size. The results presented in Chapter 5 indicate that the various parameters have different effects upon packet delay, and that the effects are different under different circumstances.

APPENDIX A

SYSTEM CODE

This appendix lists the system programs.

Link.h

```
/* header file for link.c */

#include <stdio.h>
#include <signal.h>
#include <stddef.h>
#include <string.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/param.h>
#include <errno.h>

#define SIM_SOCK "__sim.sock"       /* Unix socket of sim1.c  */

extern int SENDTO();
```

Link.c

```
/* interface to simulator program */

#include "link.h"

/* clean up before exiting */

void link_quit(s1, s2, name)
  int s1, s2;
  char *name;
{
  close(s1);
  close(s2);
  unlink(name);
  return;
}

/* get port number of RTP Internet socket */

int get_port(s)
  int s;
{
  int port_number;
  struct sockaddr_in name;
  int namelen;

  namelen = sizeof(name);
  if ((getsockname(s, (struct sockaddr *)&name, &namelen)) < 0) {
    perror("Link: getting socket name");
    return -1;
  }
  port_number = ntohs(name.sin_port);
  return port_number;
}

/* SENDTO function */

int SENDTO(socket_descriptor, buffer, buffer_len, flags, to, tolen)
  int buffer_len, flags, socket_descriptor, tolen;
  struct sockaddr *to;
  char *buffer;
{
  int s1, s2, rdsock;
  struct sockaddr_un uaddr;
  int port_number;
  char sockname[32];
  int sock_len;
  int result;
  float timing;
  int i;
  fd_set readfds;
  struct timeval timeout, *to_ptr;
  FILE *fp;

 /* get port number of RTP Internet socket */

  if ((port_number = get_port(socket_descriptor)) < 0) {
    perror("Link: invalid port number");
    return -1;
  }

 /* set up sockets */

  sprintf(sockname, "__link.%d.sock", port_number);
  s1 = socket(AF_UNIX, SOCK_STREAM, 0);
  if (s1 < 0) {
    perror("Link: opening link socket1");
    return -1;
  }
  uaddr.sun_family = AF_UNIX;
  strcpy(uaddr.sun_path, sockname);
  if (bind(s1, (struct sockaddr *)&uaddr, sizeof(struct sockaddr_un)) < 0) {
    close(s1);
    perror("Link: binding");
    return -1;
  }
  if (listen(s1, 1) < 0) {
    close(s1);
    unlink(sockname);
    perror("Link: listening");
    return -1;
  }
  s2 = socket(AF_UNIX, SOCK_STREAM, 0);
  if (s2 < 0) {
    close(s1);
    unlink(sockname);
    perror("Link: opening link socket2");
    return -1;
  }
  uaddr.sun_family = AF_UNIX;
  strcpy(uaddr.sun_path, SIM_SOCK);
  if (connect(s2, (struct sockaddr *)&uaddr, sizeof(struct sockaddr_un)) < 0)
{
    link_quit(s1, s2, sockname);
    perror("Link: connecting");
    return -1;
  }

/* send info to sim */

  sock_len = strlen(sockname) + 1;
  if (write(s2, (char *)&sock_len, sizeof(sock_len)) < 0) {
    link_quit(s1, s2, sockname);
    return -1;
  }
  if (write(s2, sockname, strlen(sockname)) < 0) {
    link_quit(s1, s2, sockname);
    return -1;
  }
  if (write(s2, (char*)&port_number, sizeof(port_number)) < 0) {
    link_quit(s1, s2, sockname);
    return -1;
  }
  if (write(s2, (char *)&flags, sizeof(flags)) < 0) {
    link_quit(s1, s2, sockname);
    return -1;
  }
  if (write(s2, buffer, buffer_len) < 0) {
    link_quit(s1, s2, sockname);
    return -1;
  }
  if (write(s2, (char *)&tolen, sizeof(tolen)) < 0) {
    link_quit(s1, s2, sockname);
    return -1;
  }
  if (write(s2, (char *)to, tolen) < 0) {
    link_quit(s1, s2, sockname);
    return -1;
  }

 /* wait for response or timeout */

  FD_ZERO(&readfds);
  FD_SET(s1, &readfds);
  timeout.tv_sec = 30;
  timeout.tv_usec = 0;
  to_ptr = &timeout;
  i = select(FD_SETSIZE, &readfds, (fd_set *)NULL, (fd_set *)NULL, to_ptr);

 /* error */

  if (i < 0) {
    link_quit(s1, s2);
```

```
      perror("Link: selecting");
      return -1;
    }

/* timeout */

    if (i == 0) {
      link_quit(s1, s2);
      printf("Link: select timedout");
      return -1;
    }

/* accept connection from sim */

    if ((rdsock = accept(s1, (struct sockaddr *)0, (int*)0)) < 0) {
      link_quit(s1, s2, sockname);
      return -1;
    }

/* read info from sim */

    if (read(rdsock, (char *)&result, sizeof(result)) < 0) {
      close(rdsock);
      link_quit(s1, s2, sockname);
      return -1;
    }
    if (read(rdsock, (char *)&timing, sizeof(timing)) < 0) {
      close(rdsock);
      link_quit(s1, s2, sockname);
      return -1;
    }
    close(rdsock);
    link_quit(s1, s2, sockname);

/*   result = 1 -->  successful transmission      */
/*   result = 2 -->  packet sent, but lost        */
/*   result = 3 -->  packet sent, but corrupted   */

/* write timing info to file */

    if ((fp = fopen("timedata", "a+")) == NULL) {
      perror("Link: opening file");
      return -1;
    }
    fprintf(fp, "%f\n", timing);
    fclose(fp);

/* end of SENDTO */
  return buffer_len;
}
```

Sim1.h

```
/* header file for sim1.c */

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/param.h>
#include <netinet/in.h>
#include <netdb.h>
#include <signal.h>
#include <errno.h>

#define PKT_SIZE 1024           /* packet size sent by RTP                    */
#define RTP_PKT_SIZE 1044       /* packet size sent to RTP from simulator     */
#define SIM_PKT_SIZE 1052       /* packet size used by copies of simulator   */
#define SIM_SOCK "__sim.sock" /* local Unix socket name                      */
#define SIM_PORT 10000          /* simulator Internet socket port number      */
```

Sim1.c

```
/* simulator program */

#include "sim1.h"

float timing;
int result;
char linksock[32];
int link_len;
int usock, isock;
int rand_seed;
struct sockaddr sendaddr;
struct sockaddr_in recvaddr;
int saddr_len, raddr_len;
char msg[PKT_SIZE];
int port_number, flags;
int sim_pkt_size = SIM_PKT_SIZE;
int rtp_pkt_size = RTP_PKT_SIZE;
int pkt_size = PKT_SIZE;
char recvpkt[SIM_PKT_SIZE];

/* remove child processes upon exit */

void sig_child()
{
  union wait status;
  pid_t pid;

  while((pid = wait3(&status, WNOHANG, NULL)) > 0);
}

/* clean up and exit due to error */

void sim_quit()
{
  close(usock);
  unlink(SIM_SOCK);
  close(isock);
  exit(1);
}

/* create Unix and Internet sockets */
```

```
void socket_init()
{
  struct sockaddr_un uaddr;
  struct sockaddr_in iaddr;

/* create UNIX socket */

  usock = socket(AF_UNIX, SOCK_STREAM, 0);
  if (usock < 0) {
    perror("Sim: opening Unix socket");
    exit(1);
  }
  uaddr.sun_family = AF_UNIX;
  strcpy(uaddr.sun_path, SIM_SOCK);
  if (bind(usock, (struct sockaddr *)&uaddr, sizeof(struct sockaddr_un)) < 0)
{
    close(usock);
    perror("Sim: binding Unix socket");
    exit(1);
  }

/* create Internet socket */

  bzero((char *)&iaddr, sizeof(iaddr));
  iaddr.sin_family = AF_INET;
  iaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  iaddr.sin_port = htons(SIM_PORT);
  if ((isock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    close(usock);
    unlink(SIM_SOCK);
    perror("Sim: opening Internet socket");
    exit(1);
  }
  if (bind(isock, (struct sockaddr *)&iaddr, sizeof(iaddr)) < 0) {
    perror("Sim: binding Internet socket");
    sim_quit();
  }
  return;
}

/* construct packet and send to RTP */

void send_rtp()
{
  char rtp_pkt[RTP_PKT_SIZE];
  char packet[PKT_SIZE];
  int s4;
  int rtp_flags;
  int rtp_port;
  struct sockaddr_in rtpaddr;
  int rtpaddr_len;

  if ((s4 = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("Sim: opening dgram socket");
    exit(1);
  }

/* extract info from received packet */

  bcopy(recvpkt, (char *)&rtp_flags, sizeof(rtp_flags));
  bcopy(recvpkt + 4, (char *)&rtp_port, sizeof(rtp_port));
  bcopy(recvpkt + 8, (char *)&rtpaddr_len, sizeof(rtpaddr_len));
  bcopy(recvpkt + 12, packet, pkt_size);
  bcopy(recvpkt + (12 + pkt_size), (char *)&rtpaddr, rtpaddr_len);

/* construct rtp packet */

  recvaddr.sin_port = htons(rtp_port); /* change SIM port to RTP port */
  bcopy((char *)&raddr_len, rtp_pkt, sizeof(raddr_len));
  bcopy((char *)&recvaddr, rtp_pkt + 4, raddr_len);
  bcopy(packet, rtp_pkt + (4 + raddr_len), pkt_size);

/* send info to rtp */

  if (sendto(s4, rtp_pkt, rtp_pkt_size, rtp_flags, (struct sockaddr
*)&rtpaddr, rtpaddr_len) < 0) {
     close(s4);
     perror("Sim: sending to RTP");
     exit(1);
  }
  close(s4);
  return;
}

/* simulate via csim */

void simulation()
{
  int bytes;
  FILE *fp;

  bytes = pkt_size;

/* write size of message and seed to a file */

  if ((fp = fopen("simdata", "w")) == NULL) {
    perror("Sim: opening file1");
    exit(1);
  }
  fprintf(fp, "%d %d", bytes, rand_seed);
  fclose(fp);

/* run csim simulation program */

  system("sim2");

/* read result and timing from a file */

  if ((fp = fopen("simdata", "r")) == NULL) {
    perror("Sim: opening file2");
    exit(1);
  }
  fscanf(fp, "%d%f", &result, &timing);
  fclose(fp);
  return;
}

/* connect to link and send information */

void send_link()
{
  int s2;
  struct sockaddr_un linkaddr;

  s2 = socket(AF_UNIX, SOCK_STREAM, 0);
  if (s2 < 0) {
    perror("Sim: opening stream socket");
    exit(1);
  }
  linkaddr.sun_family = AF_UNIX;
  strcpy(linkaddr.sun_path, linksock);
  if (connect(s2, (struct sockaddr *)&linkaddr, sizeof(struct sockaddr_un)) <
0) {
    close(s2);
    perror("Sim: connecting to link");
    exit(1);
  }

/* send result and timing to link */

  if (write(s2, (char *)&result, sizeof(result)) < 0) {
    close(s2);
    exit(1);
  }
  if (write(s2, (char *)&timing, sizeof(timing)) < 0) {
    close(s2);
    exit(1);
  }
  close(s2);
  return;
}

/* corrupt the packet before sending to sim */

void corrupt()
{
  int corrupt_byte, location;

  corrupt_byte = rand();
  location = (rand() % pkt_size);
  bcopy((char *)&corrupt_byte, msg + location, sizeof(corrupt_byte));
  return;
}

/* create Internet socket and send information to sim */

void send_sim()
{
  int s3;
  char sim_pkt[SIM_PKT_SIZE];
  struct sockaddr_in simaddr;

/* construct address of other simulator */

  bzero((char *)&simaddr, sizeof(simaddr));
  bcopy((char *)&sendaddr, (char *)&simaddr, saddr_len);
  simaddr.sin_port = htons(SIM_PORT);
  if ((s3 = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("Sim: opening dgram socket");
    exit(1);
  }

/* construct packet to send */
```

```
  bcopy((char *)&flags, sim_pkt, sizeof(flags));
  bcopy((char *)&port_number, sim_pkt + 4, sizeof(port_number));
  bcopy((char *)&saddr_len, sim_pkt + 8, sizeof(saddr_len));
  bcopy(msg, sim_pkt + 12, pkt_size);
  bcopy((char *)&sendaddr, sim_pkt + (12 + pkt_size), saddr_len);

  if (sendto(s3, sim_pkt, sim_pkt_size, 0, (struct sockaddr *)&simaddr,
sizeof(simaddr)) < 0) {
    close(s3);
    perror("Sim: sending to sim");
    exit(1);
  }
  close(s3);
  return;
}

main ()
{
  int i, mode1, mode2;
  int childpid;
  int s1;
  fd_set readfds;
  struct timeval *to_ptr;

  signal(SIGCHLD, (void (*)())sig_child);

/* seed the random number generator */

  srand((int)time((long *)NULL));

  socket_init();

/* listen for connections from link */

  if (listen(usock, 5) < 0) {
    perror("Sim: listening");
    sim_quit();
  }

  do {
    mode1 = 0; mode2 = 0;
    FD_ZERO(&readfds);
    FD_SET(usock, &readfds);
    FD_SET(isock, &readfds);
    to_ptr = (struct timeval *)NULL;
    while (((i = select(FD_SETSIZE, &readfds, (fd_set *)NULL, (fd_set *)NULL,
to_ptr)) == -1) && (errno == EINTR));

/* error */

    if (i < 0) {
      perror("Sim: selecting");
      sim_quit();
    }

/* timeouts cannot occur, the select blocks indefinitely */

    if (i == 0) {
      printf("this should never happen!");
      sim_quit();
    }
```

```
/* receive Internet packet from other copy of simulator */

    if (FD_ISSET(isock, &readfds)) {
      raddr_len = sizeof(recvaddr);
      if (recvfrom(isock, recvpkt, sizeof(recvpkt), 0, (struct sockaddr
*)&recvaddr, &raddr_len) < 0) {
            perror("SIM: receiving dgram");
            sim_quit();
      }
      model = 1;
    }

/* accept Unix domain connection */

    if (FD_ISSET(usock, &readfds)) {
      if ((s1 = accept(usock, (struct sockaddr *)0, (int *)0)) < 0) {
            perror("Sim: accepting");
            sim_quit();
      }

/* read info from link */

      if (read(s1, (char *)&link_len, sizeof(link_len)) < 0) {
            close(s1);
            sim_quit();
      }
      if (read(s1, linksock, link_len) < 0) {
            close(s1);
            sim_quit();
      }
      if (read(s1, (char *)&port_number, sizeof(port_number)) < 0) {
            close(s1);
            sim_quit();
      }
      if (read(s1, (char *)&flags, sizeof(flags)) < 0) {
            close(s1);
            sim_quit();
      }
      if (read(s1, msg, pkt_size) < 0) {
            close(s1);
            sim_quit();
      }
      if (read(s1, (char *)&saddr_len, sizeof(saddr_len)) < 0) {
            close(s1);
            sim_quit();
      }
      if (read(s1, (char *)&sendaddr, saddr_len) < 0) {
            close(s1);
            sim_quit();
      }
      mode2 = 1;
      rand_seed = rand();     /* get seed for simulator */
      close(s1);
    }

/* fork child to handle incoming packet or simulation */

    if ((model) || (mode2)) {
      if ((childpid = fork()) < 0) {
            perror("Sim: forking");
            sim_quit();
            }
        }

    } while(childpid != 0);

/* CHILD PROCESS -- RECEIVE MODE */

    if (mode1) {
      close(isock);
      send_rtp();               /* forward packet to local RTP */
    }

/* CHILD PROCESS -- SIMULATION MODE */

    if (mode2) {
      close(usock);
      simulation();                   /* call to csim process       */
      if (result == 3) corrupt();     /* corrupt packet     */
      if (result != 2) send_sim();    /* send info to sim   */
      send_link();                    /* send info to link */
    }
    return(0);                        /* child exits */
}
```

Sim2.h

```
/* header file for sim2.c */

#include <stdio.h>
#include <stddef.h>
#include <sys/types.h>
#include <sys/time.h>
#include <errno.h>

#define CELL_SIZE 53        /* cell size in bytes  */
#define MIN_PKT 1024        /* minimum packet size */
#define MAX_PKT 16384       /* maximum packet size */
#define MAX_USERS 60        /* maximum users       */
```

Sim2.c

```
/* csim simulation module */

#include "lib/csim.h"
#include "sim2.h"

STORE s;           /* storage to simulate available bandwidth   */
FACILITY f;        /* facility to simulate server               */
EVENT done;        /* signals end of simulation                 */
int count;         /* counts the jobs left to service           */
int result;        /* 1 = normal, 2 = lost, 3 = corrupt         */
float timing;      /* record time to send the real packet in ms */

/* simulation parameters */
int band, corrupt, lost;

sim()
```

```
{
  int i, before, after, users;
  int sim_size, size, channels, cells, rand_seed;
  float net_users, pkt_size, iatm;
  FILE *fp;

  create("sim");
  f = facility("server");
  set_servicefunc(f, prc_shr); /* service function is load-dependent */
  done = event("done");

/* read in parameters */

  if ((fp = fopen("sim.init", "r")) == NULL) {
    perror("Sim2: opening init file");
    exit(1);
  }
  fscanf(fp, %d%d%f%f%f%d%d", &band, &channels, &net_users, &pkt_size, &iatm,
&corrupt, &lost);
  fclose(fp);
  s = storage("channels", channels);

/* read size and seed from file from sim1.c */

  if ((fp = fopen("simdata", "r")) == NULL) {
    perror("Sim2: opening data file");
    exit(1);
  }
  fscanf(fp, "%d%d", &sim_size, &rand_seed);
  fclose(fp);

  reset_prob(rand_seed);              /* reset random number generator   */

  before = (int)expntl(net_users);    /* other users ahead of real packet */
  if (before > MAX_USERS) before = MAX_USERS;
  after = (int)expntl(net_users);     /* other users behind real packet   */
  if (after > MAX_USERS) after = MAX_USERS;
  users = (before + after + 1);
  count = users;

  for(i = 1; i <= users; i++) {
    hold(expntl(iatm));                   /* inter-arrival time of the jobs  */
    if (i == (before + 1)) {
      cells = sim_size/44;
      if ((sim_size % 44) != 0) cells++;
      user(cells, 1);                     /* simulate the real packet      */
    }
    else {
      size = (int)expntl(pkt_size);       /* create size of other packets */
      if (size < MIN_PKT) size = MIN_PKT;
      if (size > MAX_PKT) size = MAX_PKT;
      cells = size/44;
      if ((size % 44) != 0) cells++;
      user(cells, 0);                     /* simulate the other jobs */
    }
  }
  wait(done);

  if ((fp = fopen("simdata", "w")) == NULL) {  /* write result and time to */
    perror("Sim2: opening data file");         /*      a file for sim1.c   */
    exit(1);
  }
  fprintf(fp, "%d %f", result, timing);
  fclose(fp);
                                              /* end of simulation */
  exit(0);
}

user(cells, id)                                /* the job process    */
     int cells, id;
{
  float t1, t2, t3, t4, transmission;

  create("user");
                                              /* set the result     */
  if (id == 1) {
    result = 1;
    outcome = random(0,99);
    if (outcome < lost) {
      result = 2;
    }
    else {
      outcome = random(0,99);
      if (outcome < corrupt) result = 3;
    }
  }

  t1 = simclock;                                  /* start the clock for the job  */
  allocate(1, s);                                 /* wait for available bandwidth */
  t2 = simclock;                                  /* waiting is over              */

 /* transmission time = (bits)/(bits per ms) */
  transmission = ((float)(CELL_SIZE * cells * 8.0))/((float)band * 1000.0);

  t3 = simclock;
  use(f, transmission);                        /* service the job    */
  t4 = simclock;
  deallocate(1, s);                            /* release bandwidth */

  if (id == 1) {
    timing = (t2 - t1) + (t4 - t3);            /* total time to send packet */
  }
                                               /* decrement job count */
  count--;
  if (count == 0) set(done);
}                                              /* end of job process  */
```

APPENDIX B

NUMERICAL RESULTS

This appendix contains the simulation results discussed in Chapter 5.

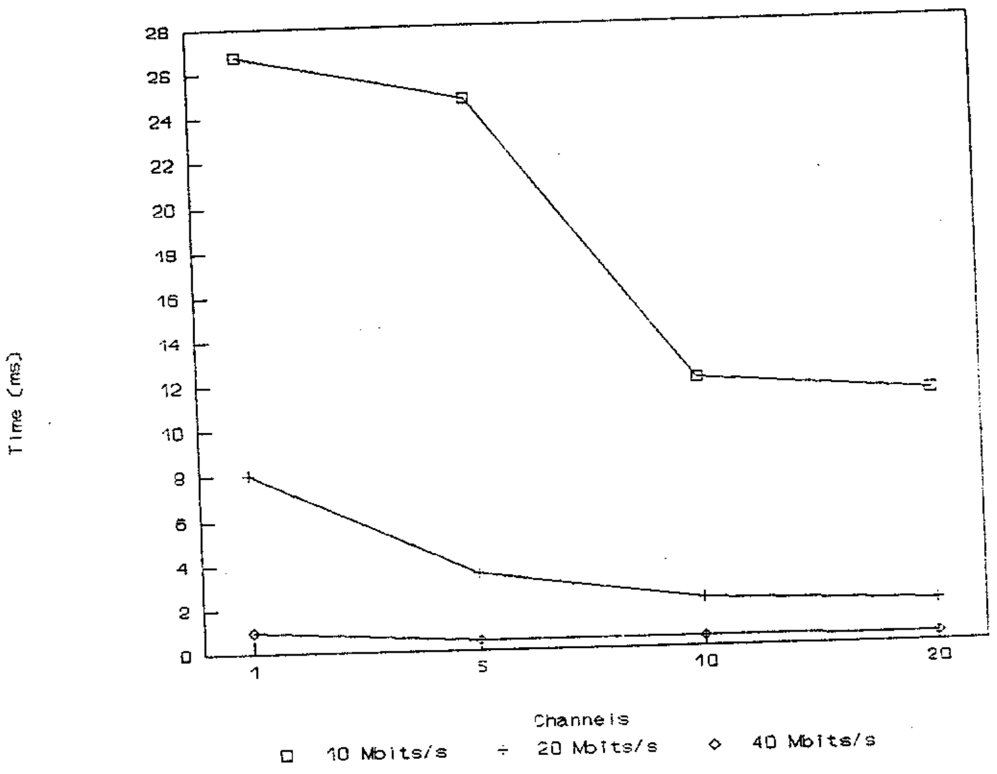

Figure B.1 Time versus Channels
average users = 15, average inter-arrival time = 2 ms, average packet size = 4 KB

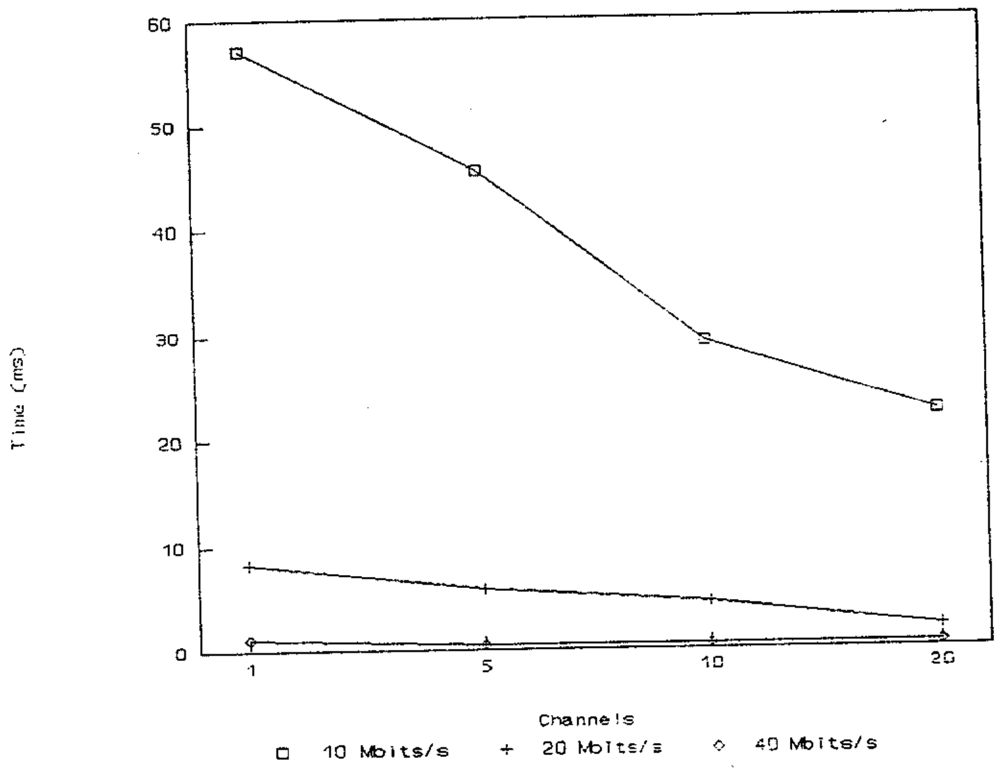

Figure B.2 Time versus Channels
average users = 30, average inter-arrival time = 2 ms,
average packet size = 4 KB

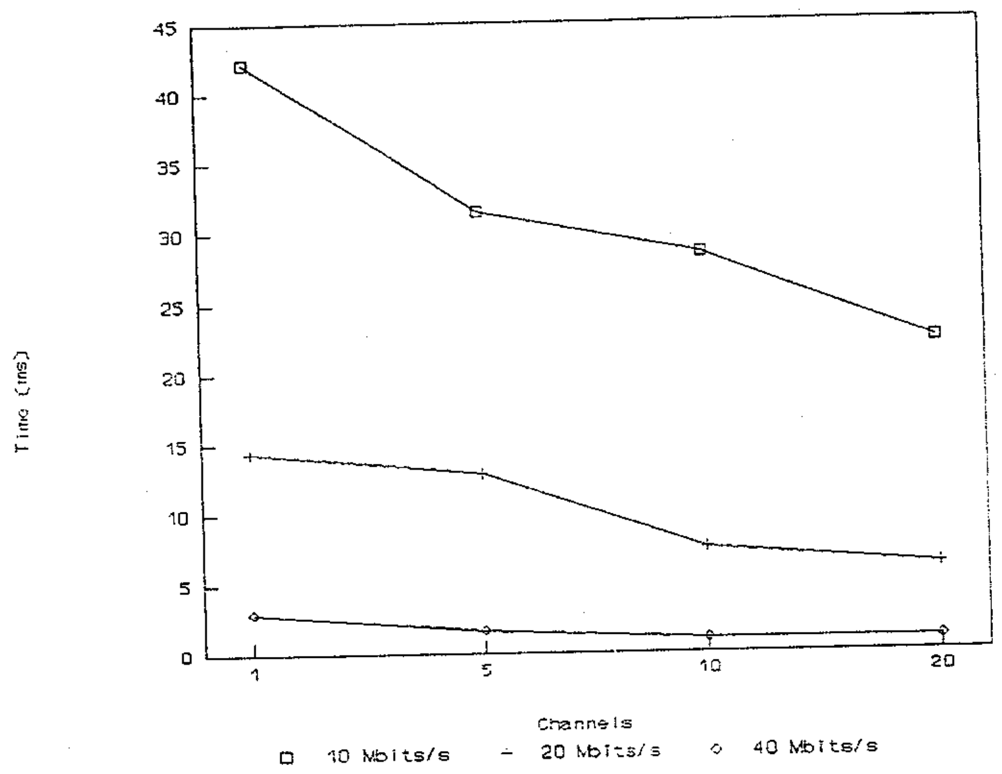

Figure B.3 Time versus Channels
average users = 15, average inter-arrival time = 1 ms,
average packet size = 4 KB

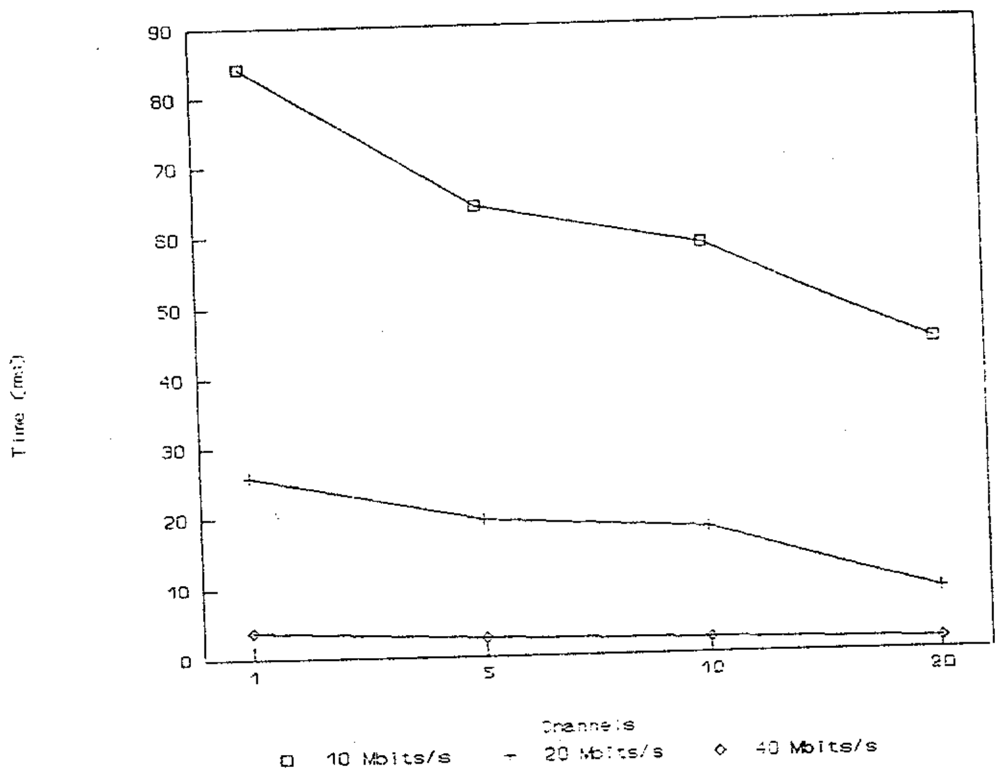

Figure B.4 Time versus Channels
average users = 30, average inter-arrival time = 1 ms,
average packet size = 4 KB

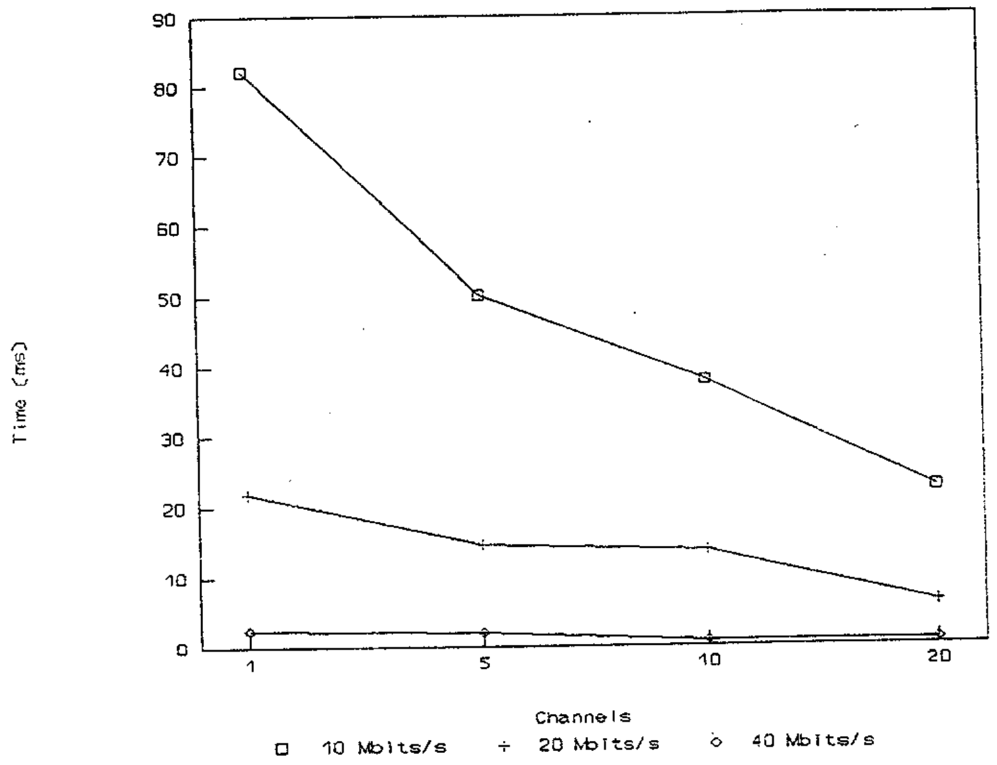

Figure B.5 Time versus Channels
average users = 15, average inter-arrival time = 2 ms,
average packet size = 8 KB

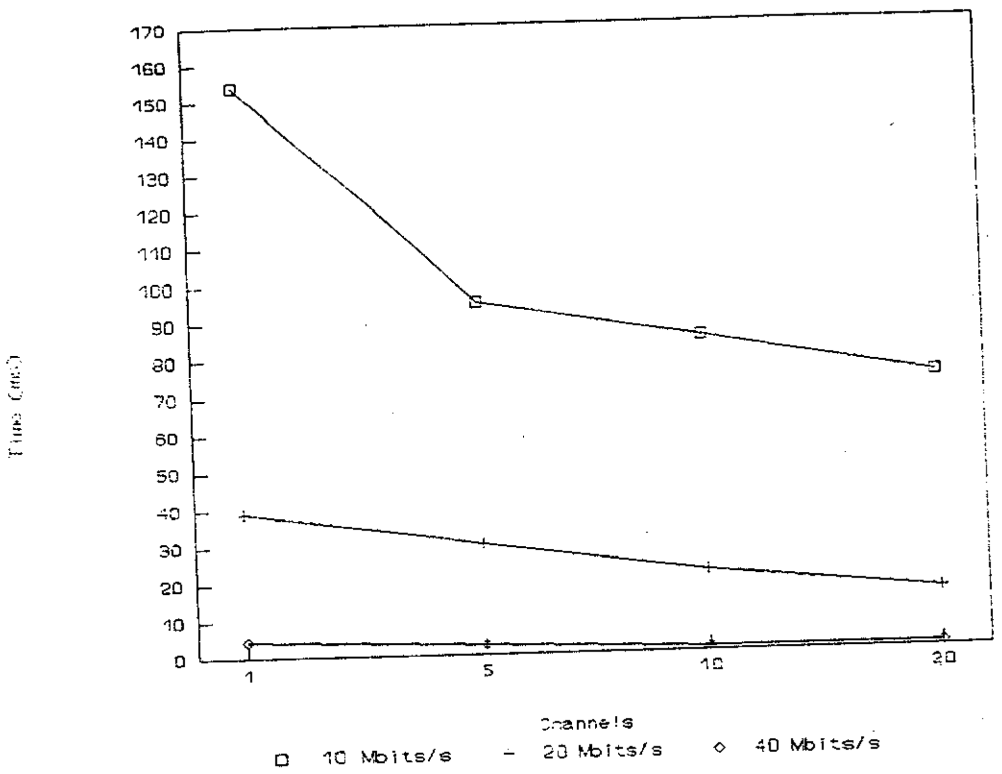

Figure B.6 Time versus Channels
average users = 30, average inter-arrival time = 2 ms,
average packet size = 8 KB

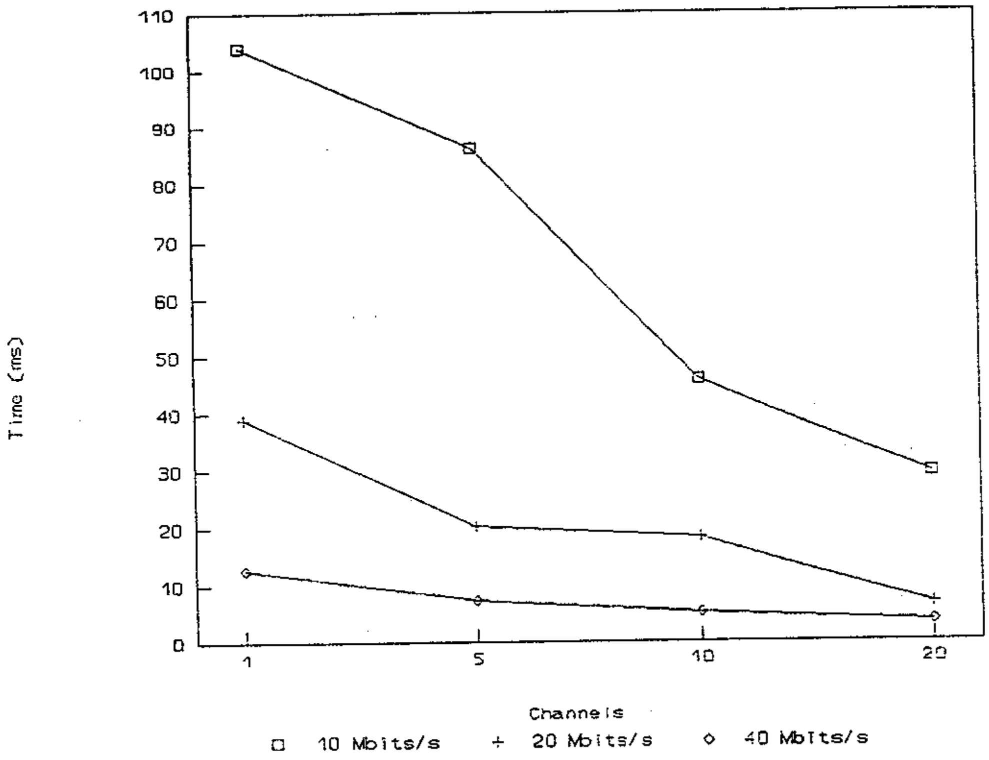

Figure B.7 Time versus Channels
average users = 15, average inter-arrival time = 1 ms,
average packet size = 8 KB

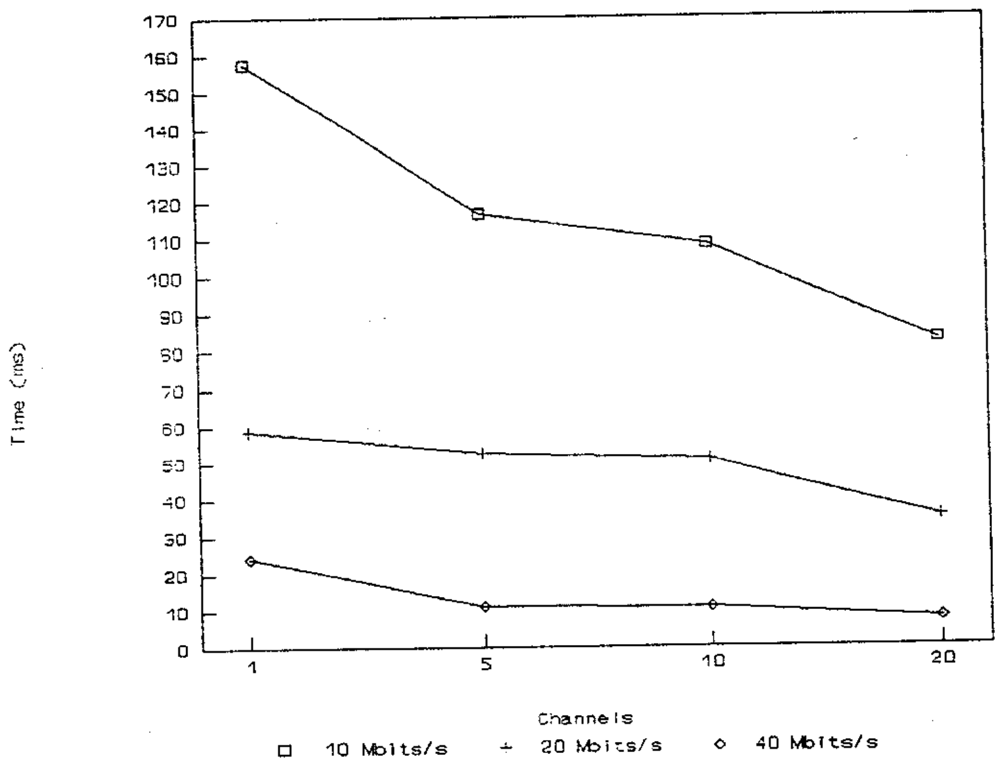

Figure B.8 Time versus Channels
average users = 30, average inter-arrival time = 1 ms,
average packet size = 8 KB

APPENDIX C

SYSTEM USERS' GUIDE

This appendix contains a guide to using the simulation system.

In order to use the simulation system, the file "link.c" must be linked with the application that will call the "SENDTO()" function. This function can replace the Unix "sendto()" function wherever desired, as it has exactly the same format. Before it returns, "SENDTO()" appends the time to send the packet to a file named "timedata". Hence, at the end of the simulation run, all "send" times can be found in this file.

Before the application is run, the program "sim1" must be running in the background in the same directory. This program is the executable form of "sim1.c". This program is set to run indefinitely, so when execution is halted, the process must be killed manually. In addition, it creates a socket, named "__sim.sock", which must also be removed manually before the program can be run again.

"Sim1.c" assumes that the application uses a constant packet size, which can be set in the file "sim1.h". If a variable packet size is possible, then the system should be modified so that the packet size is passed from "link.c" to "sim1.c".

The application on the receiving end may need to be modified to receive a packet of a larger size. This size is equal to the sender's packet size plus the size of a **struct sockaddr** plus the size of an **int**. This is due to the fact that the address of the original sender and the length of this address may need to be included in the packet, from where they can be extracted.

Finally, the copies of "sim1" use packets of the following size: the sender's packet size plus the size of a **struct sockaddr** plus three times the size of an **int**. Again, these sizes would need to be modified to include another integer (the original packet size) if variable-sized packets were allowed.

The user can set the traffic parameters discussed in Chapter 5. These parameters are found in the file "sim.init". In addition, the user can also allow for lost and/or corrupted packets in "sim.init". These parameters represent the percentage of packets that will be lost and/or corrupted. They should be integers in the range [0, 99] (if they are set to "0", they have no effect). Packets would be "sent" in simulated time and then either dropped or corrupted and sent in "sim1.c". All of the parameters should be typed on a single line in "sim.init", separated by a space.

REFERENCES

[1] Y. H. Kim and C. K. Un, "Performance Analysis of Statistical Multiplexing for Heterogeneous Bursty Traffic in an ATM Network," *IEEE Trans. Commun.*, vol. 42, no. 2/3/4, pp. 745-753, Feb./Mar./Apr. 1994.

[2] H. Saito, M. Kawarasaki, and H. Yamada, "An Analysis of Statistical Multiplexing in an ATM Transport Network," *IEEE J. Select. Areas Commun.*, vol. 9, no. 3, pp. 359-367, Apr. 1991.

[3] F. Bonomi, S. Montagna, and R. Paglino, "A further look at statistical multiplexing in ATM networks," *Computer Networks and ISDN Systems*, vol. 26, no. 1, pp. 119-138, Sep. 1993.

[4] K. Sriram, "Methodologies for bandwidth allocation, transmission scheduling, and congestion avoidance in broadband ATM networks," *Computer Networks and ISDN Systems*, vol. 26, no. 1, pp. 43-59, Sep. 1993.

[5] Y. Wang, T. Lin, and K. Gan, "An Improved Scheduling Algorithm for Weighted Round-Robin Cell Multiplexing in an ATM Switch," *Conference Record - IEEE International Conference on Communications*, vol. 2, pp. 1032-1037, 1994.

[6] M. Hirano and N. Watanabe, "Characteristics of a cell multiplexer for bursty ATM traffic," *Proc. ICC'89*, 1989, pp. 13.2.1-5.