A report on developing a data generator application to generate and send UDP datagrams at a specified rate.

Shah Dhruv Amitkumar

(Redid: 819653114)

Course Number: COMPE 571

Professor Name: Dr. Yusuf Ozturk

Date Submitted: 09-28-2015

Contents

Introduction

The data generator application concentrates on generating and sending UDP datagrams at a specified rate to gain specific bandwidth. It has been seen that one single process can achieve maximum upto 3 to 5 Gbits/sec without sleep time. A typical *sleep* system call takes a time value as a parameter, specifying the minimum amount of time that the process is to sleep before resuming execution. Sleep function places current process into an inactive state for a period of time. Without introducing sleep call, process will generate unexpected errors which results into unwanted bandwidth. Introduction to sleep call will decrease the unwanted errors. On the contrary it will decrease chances to achieve desired bandwidth with a single process.

One more problem which has to be addressed is drift. As explained earlier, when system will call sleep function process will sleep for some predefined time but in reality, it takes some more time to wake up than given in the parameter. There is no guarantee that sleep function will return in exact given parentheses value. Other processes are also running and they may control the processor when the sleep expires. Therefore, the process must wait a short period of time for the processor. This is called drift.

This single process application fails to generate packets at the specified rates when the data rate increases to beyond 300 Mbits/sec. The poor behavior was attributed to the indeterminate response time of the sleep() command when the sleeping period gets smaller and smaller. Also inaccuracies of timers and the inaccuracy of the sleep system call results into packet loss which results into inaccurate desired output. So to achieve desired output and to remedy these problems, multiple processes will be required to compensate the errors described above. By invoking multiple cooperating processes to achieve the same task will require each process supply data at a slower rate and cooperatively achieve the rate required.

Basically, this application focuses on generating and sending UDP datagrams to generate packets at predetermined rates.

Discussion on the methodology to solve the problem

Due to e inaccuracies of timers, the inaccuracy of the sleep system call, indeterminate amount of time taken by system calls and unexpected drift, one has to create multiple process to compensate with these issues to achieve desired bandwidth at specific data rate. Due to sleep function, one cannot achieve maximum achievable bandwidth and output bandwidth will be less than expected and if we decrease the sleep parameter a lot then, data loss will increase accordingly. Other problem that occurs is drift. Due to Drift output bandwidth will be fluctuating and to remedy this process should increase the sleep time. Multi processes will address to both these problems.

Multi-tasking is where different processes each get to run on the processor part of the time. For example, if three processes need to run, the operating system will let one run for a short amount of time, then the next process will run for a short amount of time, and then the third process will run for a short amount of time (this short period of time is called the processes time-slice). When we create multiple processes, the operating system chooses which process is going to run at any particular time. A scheduling algorithm and process priorities determine exactly what process will run next. Assuming that all processes have an equal priority, each process will be given a timeslice by the operating system. A process is free to run during its time-slice. At the end of its time-slice the operating system will pre-empt the process, task-out the process, and allocate a time-slice to another process. In this way, each process that needs to run will get to run.

Here, to make a process a server, one has to follow steps given below.

1. Create a socket with the socket() system call.
2. Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Measure 1 sec delay
4. Receive packets during that time and measure bandwidth.
5. Send and receive data using the read() and write() system calls.

To make a process a client, one has to follow steps below.

1. Create a socket with socket() system call.
2. Create processes with fork() call.
3. Start time.
4. Sends data to server.

5. Sleep for some defined amount of time.
6. End time.
7. Measure drift.
8. Close socket.

We will need multi processes to run at a time to achieve specific bandwidth. Here, to create a process one has to use fork() system call. The call to fork creates a new process. The new process is identical to the original except for one minor difference, the return value of the fork call. To understand what values can fork return, let us take a look at one of the basic example of fork call.

```
processID = 1;
for( count = 0; count < 7; count++ )
{
        if( processID != 0 )
        {
                processID = fork();             //creating a child
        }
        else
        {
                break;
        }
}
```

The call to fork returns some value which is stored in processID.

1. If fork call returns negative value, the creation of child process was unsuccessful.
2. If fork call returns equal to 0, the process is the new process and it is called the child.
3. If the value of the processID is greater than zero, the process is the original process and it is the parent.

For Parent process,

```
Main()
{
        Fork();
        next statements;
}
```

For parent process,

```
main()
{
    fork();
    next statements;
}
```

For child process,

```
main()
{
        fork();
        next statements;
)
```

In this example, when the parent process will execute the fork call an identical process will be generate. Only this new identical process will have different processID. This will make two identical copies of address space, one for child and one for parent. Both processes start their execution at the next statement following the fork call.

Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names. This way one can create multiple processes.

To calculate drift, one has to go through this piece of code to understand the logic.

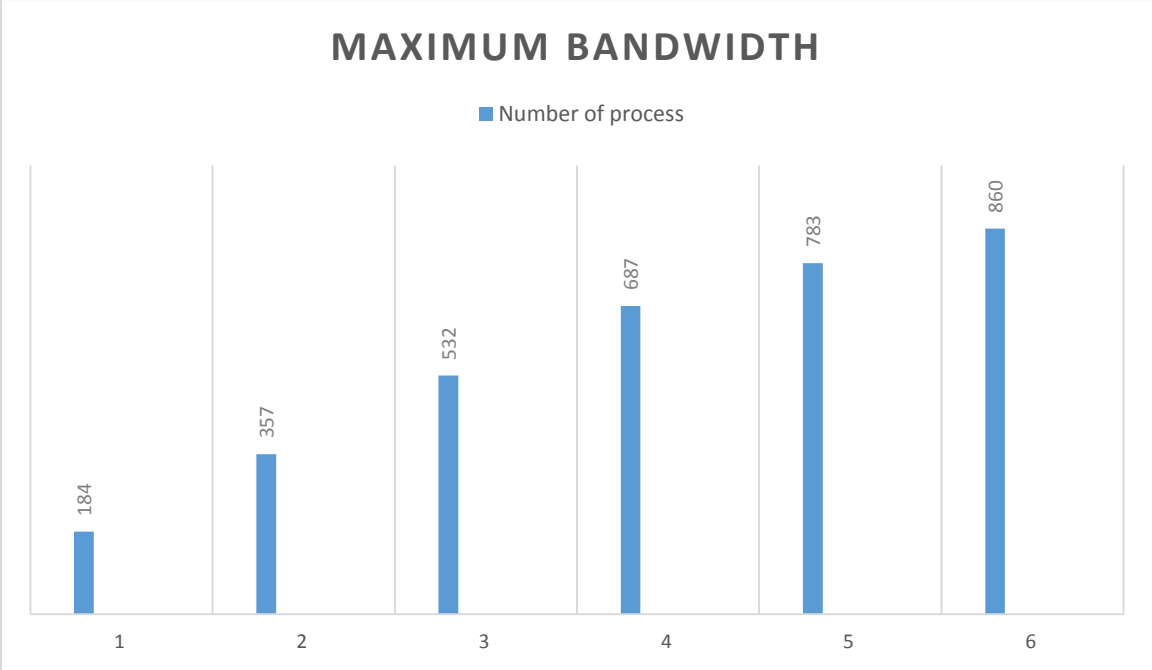
```
[...]
 gettimeofday( &start_time, NULL );
for( count = 0; count < NO_OF_ITERATIONS; count++ )
{
 usleep(SLEEP_TIME);
}
 gettimeofday(&stop_time, NULL );
 [...]
```

The start time is retreived using the gettimeofday() call. Next, a loop is started and executed NO_OF_ITERATIONS times. Each time through the loop, the process sleeps for SLEEP_TIME. (1000 microseconds or 1 millisecond). At the end of the loop, the end time is measured. In theory, the stop time minus the start time should be the NO_OF_ITERATIONS times the SLEEP_TIME. However, this is not the case, the for loop takes longer to execute and drift is appearent. The example program will display the total drift and the drift per iteration.

Results of running program

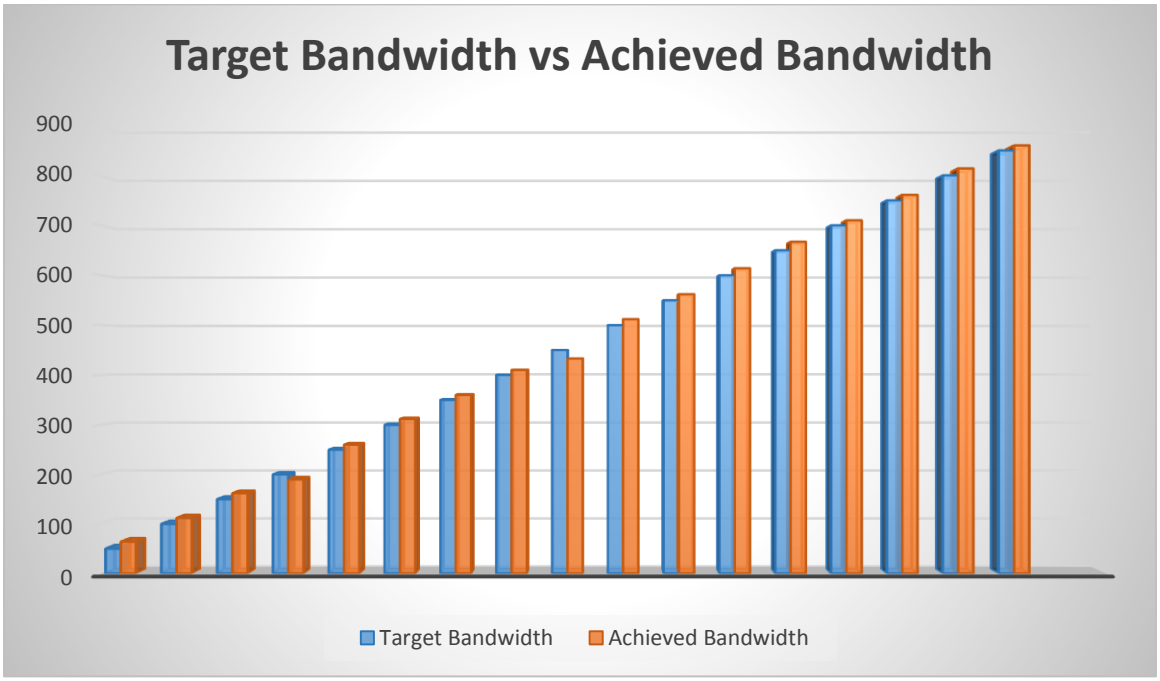
1. The number of processes versus maximum bandwidth achieved.

Number of Process	Maximum bandwidth achieved
1	184
2	357
3	532
4	687
5	783
6	837



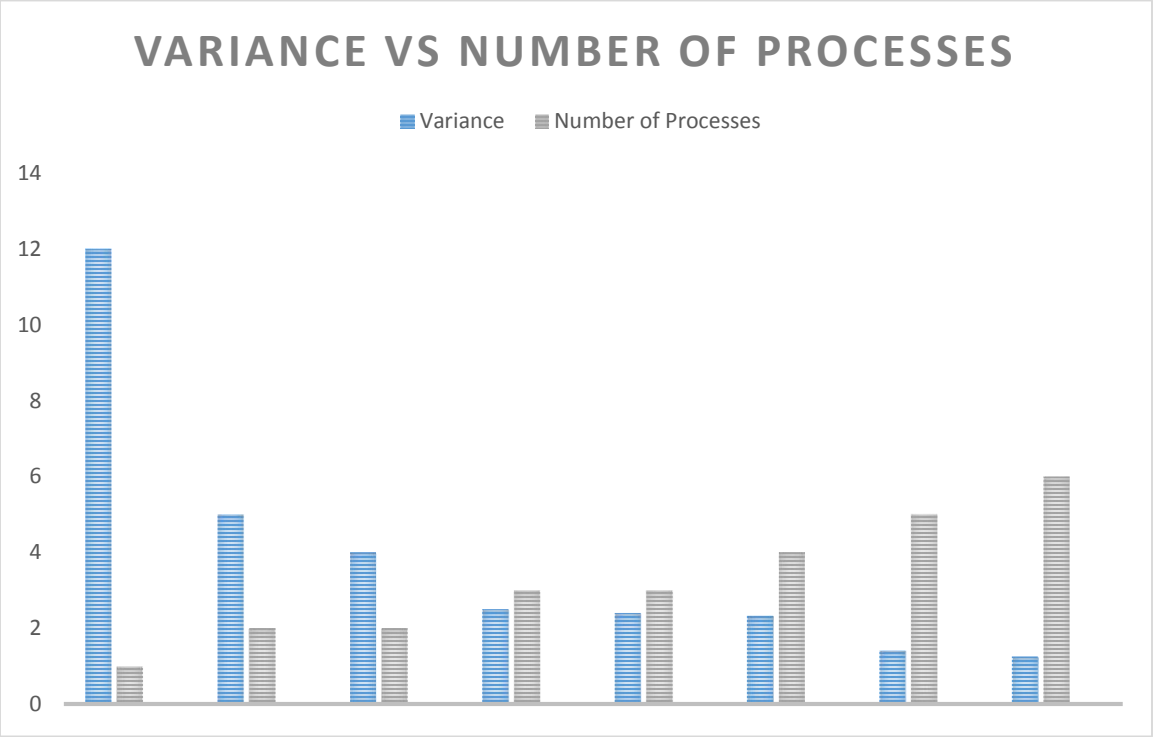
2. The target bandwidth versus achieved bandwidth

Target Bandwidth	Achieved bandwidth
50	64
100	112
150	162
200	190
250	260
300	312
350	360
400	410
450	433
500	512
550	562
600	614
650	667
700	710
750	761
800	810
850	860



3- The variance in the achieved bandwidth versus the number of processes to achieve a certain rate.

Variance	Number of processes
12%	1
5%	2
4%	2
2.5%	3
2.4%	3
2.33%	4
1.42%	5
1.25%	6



Analysis of Results

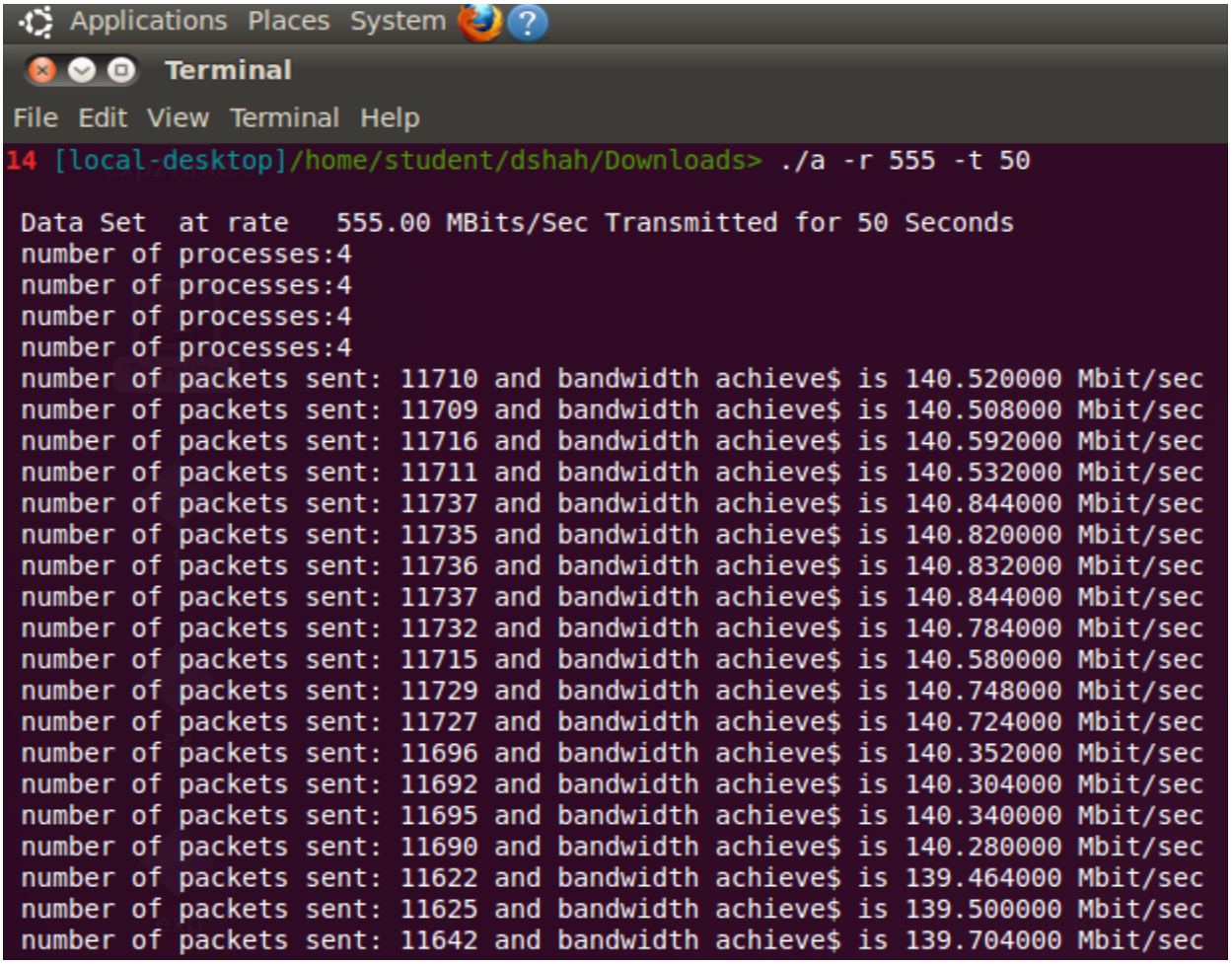
To achieve maximum bandwidth with least processes and least error rate I did this experiment repeatedly. I tried to find for how many processes and for how much sleep time, output bandwidth can be highest. With the experiments analyses I came to know that with decreasing the delay after one point error rate increased a lot.

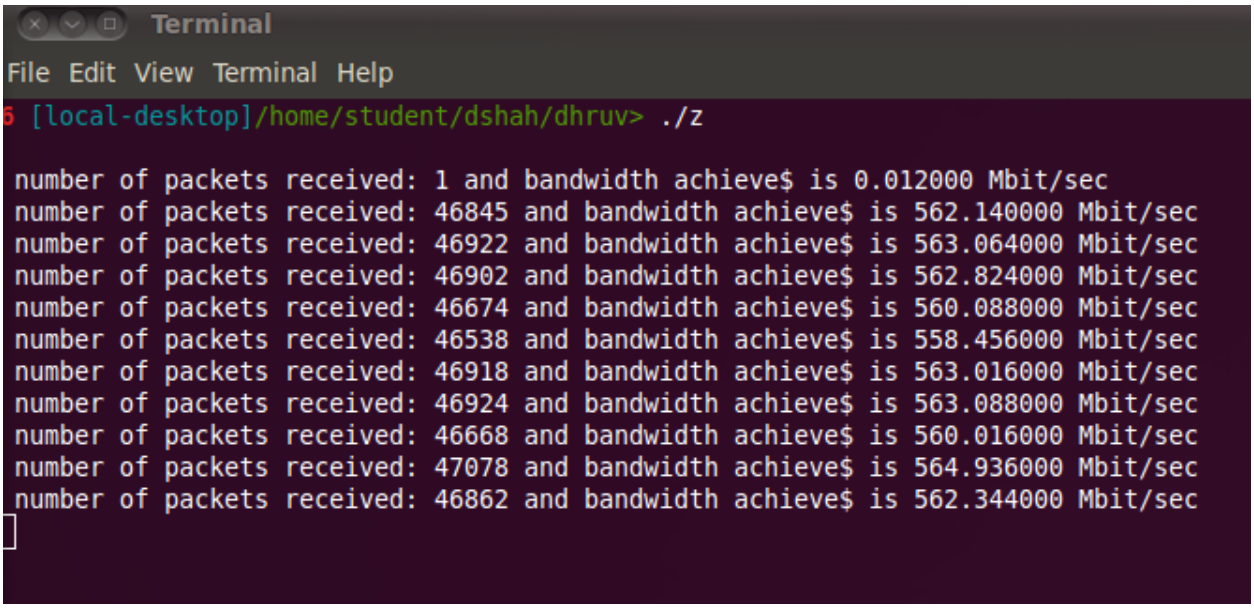
Also it was discovered that with less sleep time, more processes has to be created to achieve the specific bandwidth. So I made an array for the values I found for bandwidth with creating least process and least error rate which is shown below.

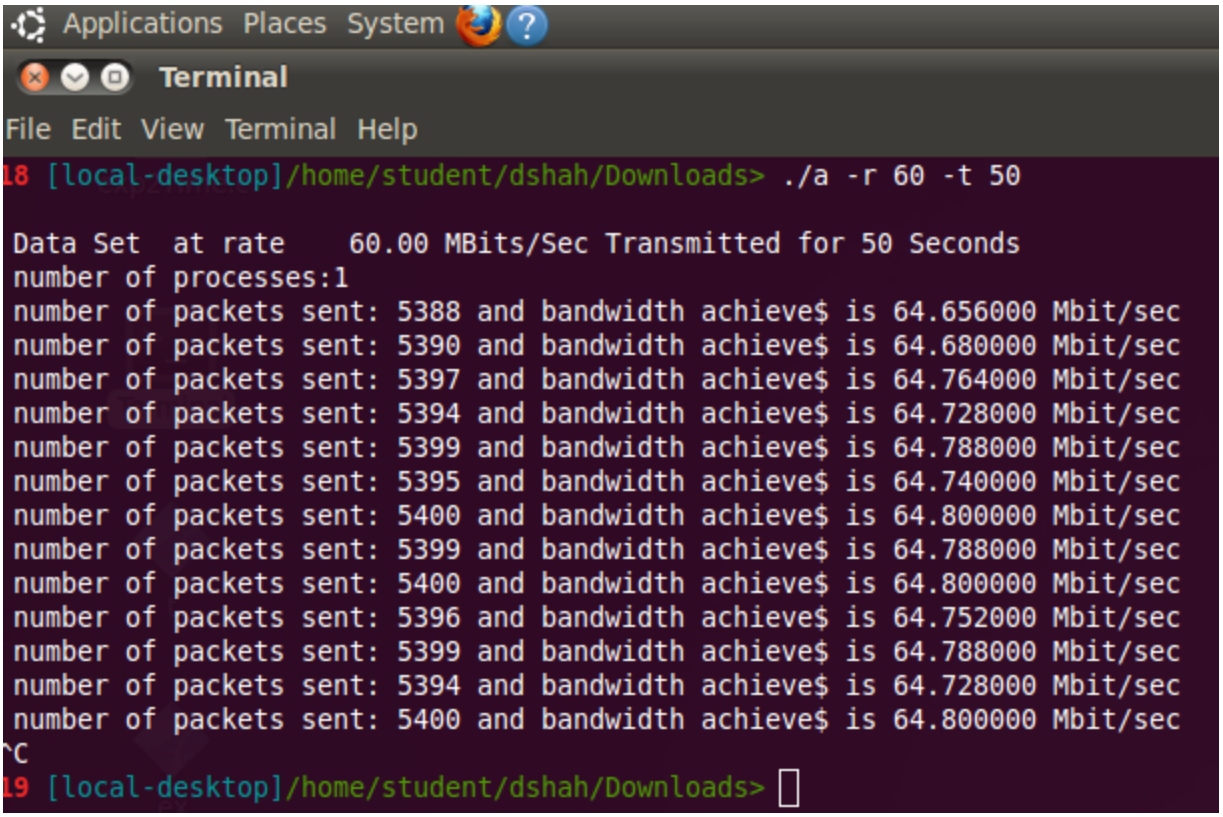
Here, from the experiments it can be seen that with the increase in bandwidth the error will be decreased. For example, if we consider for 100 mbps the output that can be achieved is 112 mbps so error rate = (112-100)*100% which will be around 12 % error rate and for 200 mbps it can be seen that the output will be around 190 mbps so error rate will be (|190-200|/200)*100% which will be 5% error so with the increase in the bandwidth the output error will decrease.

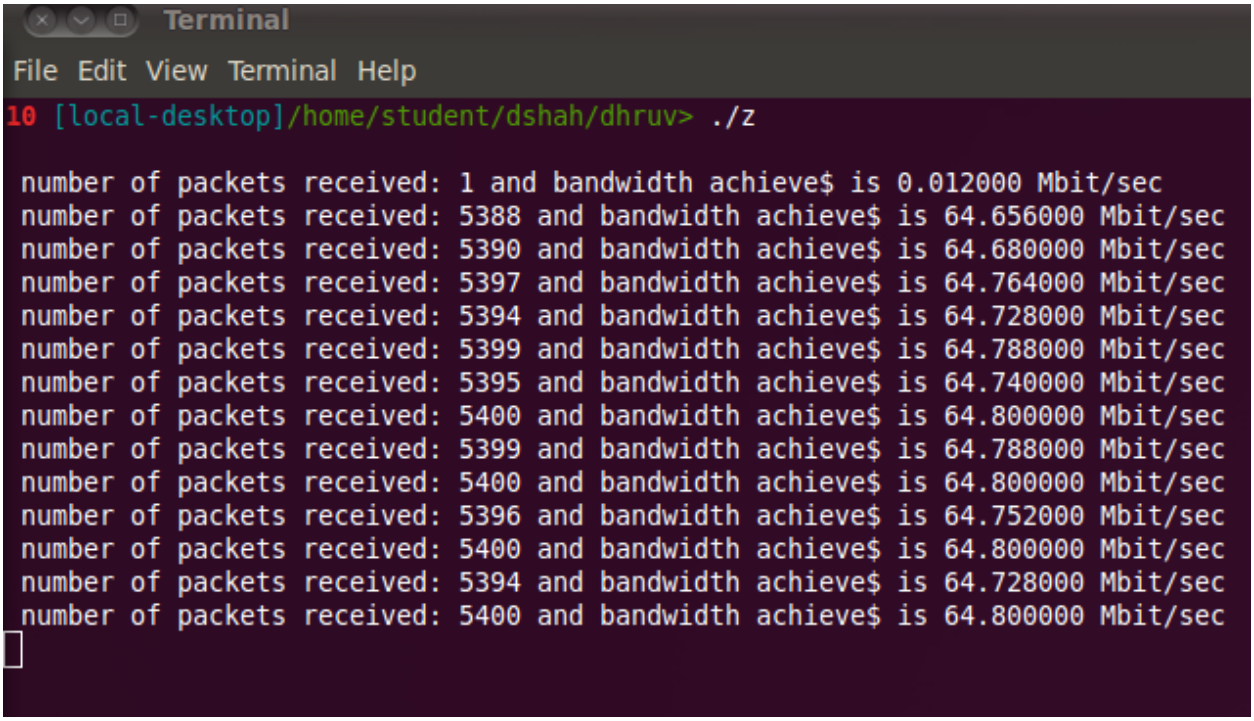
Here, I have attached 3 experiments screenshots. In first picture, each page will be client sending some number of bits. Client also lets you know how many processes are used to achieve given data rate and that data rate can be seen on the server side. In the second picture, it will be the server with number of bit receiving from the client.

Sending 555 Mbit/Sec by Client and Receiving 562.14 Mbit/Sec by Server



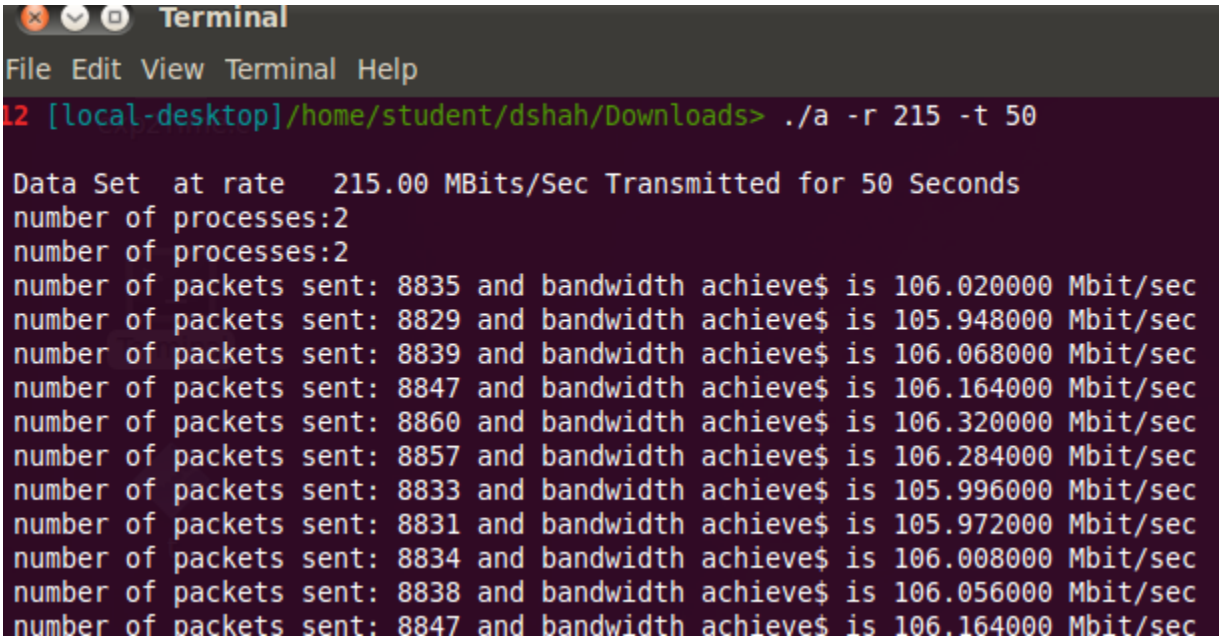


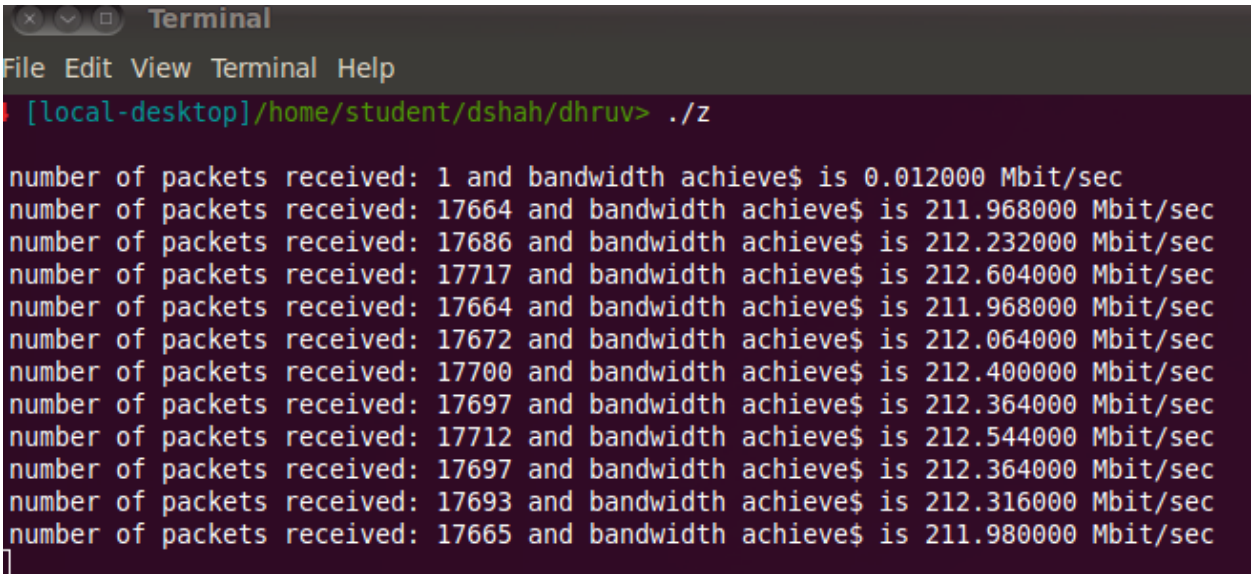




Sending 60 Mbit/Sec by Client and Receiving 64.65 Mbit/Sec by Server

Sending 215 Mbit/Sec by Client and Receiving 212.23 Mbit/Sec by Server





Conclusion

My assumption was that the system takes more time to start working after sleep system call but by increasing or adjusting processes and delay timings this can be overcome. My experiments stand true with my assumption. The experiments show that the precise data rate can be achieved with balance of processes and delay timings. While working on this project I gained knowledge about system calls, Process creation, Socket Programming, how to minimize effect of system limitations on application performance.

Appendix

Server Side Code:

```
#include <arpa/inet.h>
#include <stdio.h>       // perror
#include <stdlib.h>
#include <sys/socket.h>  // network
#include <unistd.h>      // exit
#include <string.h>      // memset
#include <time.h>


#define BUFLEN 1518
#define PORT 9001

long long
timeval_diff(struct timeval *difference,
             struct timeval *end_time,
             struct timeval *start_time
            )
{
  struct timeval temp_diff;

  if(difference==NULL)
  {
    difference=&temp_diff;
  }

  difference->tv_sec =end_time->tv_sec -start_time->tv_sec ;
  difference->tv_usec=end_time->tv_usec-start_time->tv_usec;

  /* Using while instead of if below makes the code slightly more robust. */

  while(difference->tv_usec<0)
  {
    difference->tv_usec+=1000000;
     difference->tv_sec -=1;
  }

  return 1000000LL*difference->tv_sec+
                   difference->tv_usec;

}


int main(int argc, char**argv)
{
```

```
struct sockaddr_in servaddr,cliaddr,si_other;
int sockfd,n;
socklen_t len;
 char mesg[BUFLEN];
 time_t start, end;
 double timedif;
  int packetCt,cnt=0;

  struct timeval earlier;
struct timeval later;
struct timeval interval;


   sleep(3);


   sockfd=socket(PF_INET,SOCK_DGRAM,IPPROTO_UDP);

 bzero(&servaddr,sizeof(servaddr));
 servaddr.sin_family = AF_INET;
 servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
 servaddr.sin_port=htons(8001); //port number
 if(bind(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr))==-1)
         {perror("bind");}

 if(gettimeofday(&earlier,NULL))
{
  perror("first gettimeofday()");

  exit(1);
}

  packetCt = 0;

 for (;;)
 {
    len = sizeof(cliaddr);
    n = recvfrom(sockfd,mesg,BUFLEN,0,(struct sockaddr *)&cliaddr,&len);
    packetCt++;
     if(gettimeofday(&later,NULL))
      {
      perror("second gettimeofday()");

      exit(1);
      }


      if(timeval_diff(NULL,&later,&earlier) >= 1000000)
      {
```

```
                printf("\n number of packets received: %d and bandwidth
achieve$ is %f Mbit/sec",packetCt, (double) packetCt*1500*8/1000000);
                packetCt = 0;
                if(gettimeofday(&earlier,NULL))
                        {
                        perror("first gettimeofday()");

                        exit(1);
                        }
        }
   }

}
```

Client Code:

```
#include <arpa/inet.h>
#include <stdio.h>       // perror
#include <stdlib.h>
#include <sys/socket.h>  // network
#include <unistd.h>      // exit
#include <string.h>      // memset
#include <time.h>


#define TARGET_IP "127.0.0.1"    // IP address of the destionation
computer
#define TARGET_PORT 9001
#define BUFLEN 1518

long long
timeval_diff(struct timeval *difference,
             struct timeval *end_time,
             struct timeval *start_time
            )
{
  struct timeval temp_diff;

  if(difference==NULL)
  {
    difference=&temp_diff;
  }

  difference->tv_sec =end_time->tv_sec -start_time->tv_sec ;
  difference->tv_usec=end_time->tv_usec-start_time->tv_usec;

  /* Using while instead of if below makes the code slightly more
robust. */

  while(difference->tv_usec<0)
  {
    difference->tv_usec+=1000000;
     difference->tv_sec -=1;
  }

  return 1000000LL*difference->tv_sec+
                   difference->tv_usec;
```

```
}


void usage()
{
 printf("\n");
 printf("\n -r : rate ");
 printf("\n -t : Total time duration  (seconds) ");
 printf("\n\n");

}

typedef struct packetheader
{
        int     packetcounter; // The packet sequence number
        int     clearcounters; // a signal to the receiver to clear
all counters
        int     expcounter;    // The experiment counter
        int     resetstat;     // a signal to the receiver to reset
the statist$
        int     IFG;  // the duration between release of two packets
        int     complete; // Experiments are complete , terminate
        int     rateseq; // The sequence number of the rate -- to
detect larger$
        int     expduration;

} packetheader ;

union datapacket{  // Union placing the packet header and data packets
into the
                    //same address space
        char data[1518];
        struct packetheader pkthdr;
        } datapacket;
int main(int argc,char **argv)

{
    struct sockaddr_in si_output;
    int s, i;
    long total=0;
     int no_of_Children;
    socklen_t slen=sizeof(si_output);
    time_t start,end;
    char *buffer;
    int l=0;
    int timesec;
```

```
    double rate,timedif;
    pid_t pid;
   int delay,children,array_num;
    int
time[45]={600,250,120,70,40,23,8,1,47,36,24,18,11,5,1,26,19,15,9,6,3,1
,18,15,11,8,6,2,15,13,11,5,15,1,1,1,1,1,1,1,1};
    int
processes[45]={0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,2,2,2,2,2,2,2,3,3,3,3,3,3
,4,4,4,4,5,5,5,5,5,5,5,5,5,5};
    int packetCt;
  struct timeval earlier;
  struct timeval later;
  struct timeval interval;
  struct timeval x;

   buffer = (char *) malloc(BUFLEN);
   memset(buffer,'\0',1518);

 if (argc < 2 )
        {
        usage();
        exit(0);
        }

else
        {
        for (i=1; i < argc ; i+=2)
        {
                if (argv[i][0] == '-')
                {
                        switch(argv[i][1])
                        {
                        case 'r':
                                rate = atoi(argv[i+1]);
                                break;
                        case 't':
                               timesec =  atoi(argv[i+1]);
                                break;
                 default:
                                return(0);
                                break;
                        }
                   }
           }
      }
```

```
      /* creating socket */

      if ((s=socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
           {perror("socket");}
             memset((char *) &si_output, 0, sizeof(si_output));

           si_output.sin_family = AF_INET;
           si_output.sin_port = htons(TARGET_PORT);
          if (inet_aton(TARGET_IP, &si_output.sin_addr) == 0) {
perror("inet_aton() $ton() failed"); _exit(1); }

union datapacket  buf;
     memset(buf.data,1500,0);

   printf ("\n Data Set  at rate %8.2f MBits/Sec Transmitted for %d
Seconds",rate,timesec);
    fflush(stdout);

if(rate<=1000)
{
array_num=(int)((rate/25));
children=processes[array_num];
delay=time[array_num];
}
else
{

children=5;
delay=1;
}

printf("\n number of processes:%d",children+1);
pid=1;
   for(l=0;l<children;l++)
{

        if(pid!=0)
                {

                     pid=fork();
                }
       else           {
                      break;
          }
```

```
}


if(gettimeofday(&earlier,NULL))
  {
    perror("first gettimeofday()");

    exit(1);
  }

if(gettimeofday(&x,NULL))
  {
    perror("first gettimeofday()");

    exit(1);
  }

while(timeval_diff(NULL,&later,&earlier) < 1000000*timesec)
{

     if( sendto(s, buf.data, BUFLEN, 0, (struct sockaddr
*)&si_output,slen)!=BUFLEN)
     {
          perror("Error: sendto()");
          _exit(1);
      }

        usleep(delay);           //delay of 'delay' microsecond


        packetCt++;

        buf.pkthdr.IFG =6;


 if(gettimeofday(&later,NULL))
        {
        perror("second gettimeofday()");

        exit(1);
        }

 if((double)timeval_diff(NULL,&later,&x) >= 1000000)
     //calculating final drift
        {
```

```
                printf("\n number of packets sent: %d and bandwidth
achieve$ is %f Mbit/sec",packetCt, (double) packetCt*1500*8/1000000);
                packetCt = 0;
                if(gettimeofday(&x,NULL))
                        {
                        perror("first gettimeofday()");

                        exit(1);
                        }
        }


}


close(s);             //closing socket
return 0;

}
```