Simulation of Fast Ethernet

Computer Networks and Communications

Programming Project
CS5310

Student: Soto Alvarez del Castillo, Maria Adela
Student ID: A04944834

San Marcos TX, Fall 2019

Introduction

This project simulates the Fast Ethernet. The goal of the project is to practice basic socket API programming through a client/server application.

Background

There are two primary areas for concern when talking about networks, one is upgrading bandwidth and the other one is cabling, and hubs.

Many of the designs for WAN, LAN, and MAN networks have been standardized under the name of IEEE 802.

This brings us that along with the computational development there have been two kinds of Ethernet.

Classic Ethernet, which solves the multiple access problem working at 10Mbps speed; and switched Ethernet, in which devices called switches are used to connect different computers.

The problems associated with finding breaks or loose connections drove it toward a different kind of wiring pattern, in which each station has a dedicated cable running to a central hub, that simply connects all the attached wires electrically as if they were soldered together.

At that time the switches were becoming popular, these only output frames to the ports for which those frames are destined, the speed of 10-Mbps Ethernet was not enough.

Classic Ethernet evolves away from the single long cable architecture to the Fast Ethernet.

The Fast Ethernet works at the speed of 100Mbps. Referend on the IEEE standard 802.3u fast ethernet/100BASE-T specified in May 1995. The features of this type of fast ethernet are as follows:

- Includes multiple PHY layers.
- It uses original ethernet MAC but operates at 10 times higher speed.
- It needs a star wired configuration with a central hub.

Project Specification

Problem Analysis

We simulate a fast Ethernet with a collection of processes that execute on a collection of UNIX machines in the CS department Linux Lab.

The project programming language is **C** language.

All interprocess communication is using UNIX sockets.

The underliving transport protocol is TCP.

The simulation is composed of the following processes:

1. Station Process (SP)

One station process for each simulated station.

Each SP has an associated simulation data file that will read only by process of simulation input data files.

The 10 different station processes allowed are in separate text files named (`StationProcess1.txt` to `StationProcess10.txt`).

2. Communication Switch Process (CSP)

There is only one such process, which simulates the function of a switch in a fast Ethernet.

The project contains the program `CBP.c` that acts as server in the client/server implementation and implements the functionalities of a Communication Bus process.

It contains 10 station process `StationProcess1.txt`, all through `StationProcess10.txt` which acts as input files.

For every station, the sequence of frames to be sent, together with the destination station number, is specified as simulation input data file.

Solution Design

SP Functionality

- Its functionality depends on whether there is an incoming data frame, receive it.
- The file `SP.c` reads an input data line from its simulation input data file, by example: `StationProcess.txt` and format it as a data frame. The contents of the input file are read into a buffer line by line. The destination station to which the data frame is to be sent is mentioned in the input file itself.
- Then it will be sending a request frame to the CSP asking permission to send its data frame just formatted and wait for reply.

- If a reject reply comes from then CSP, it re-transmits the request frame.
- If it a positive reply comes from then CSP, it will then proceed to send data frame to the CSP.
- With the `Socket Function`, a socket is created once the buffer has read a line from the input file.
- Connection is initiated with the connect function from the socket with file descriptor `sockfd` to the destination socket whose address is specified by the `serv_addr` and `sizeof(serv_addr)` arguments.
- Finally, it writes the part of frame into the socket `sockfd`.

CSP Functionality

CSP functions as the switch of the fast Ethernet.

This switch gives permission to a SP to send a frame, accept frames from a source station and forward each frame received to its destination station.

Every time a request frame is received, the CSP will either send a reply to the SP so that the SP will send a frame or hold it in its waiting queue if it is busy. When it receives a data frame it will inspect the destination address. It will then send out the frame to the destination station.

A data structure named `sockaddr_in` is used to store the addresses of the client and server.

- Function `sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)`

  The function socket creates a socket.

  Arguments:

  - AF_INET specifies the namespace
  - SOCK_STREAM specifies the communication style
  - IPPROTO_TCP is the protocol.

  The return value from socket is the file descriptor for the new socket named sockfd.

- Function `bind (sockfd, (struct sockaddr*) (&serv_addr), sizeof(serv_addr))`

  This function assigns an address to the socket sockfd.

  The serv_addr and sizeof(serv_addr) arguments specify the address.

  The return value is 0 on success and -1 on failure.

- Function `listen(sockfd, 10)`

The listen function enables the socket sockfd to accept connections, thus making it a server socket.

The argument 10 specifies the length of the queue for pending connections

- The CSP maintains two queues: a data frame queue that contains data frames to be forwarded; and a request frame queue that contains request from SPs for accepting data frames. Each queue has a fixed size.

  The CSP will send a reply to allow the SP to send its data frame, if data frame queue is not full once a request frame is received. If the data frame queue is full, and the request frame queue is not full, then the CSP will place that request frame into the request frame queue. If the request frame queue is full as well, a reject reply will to send back the SP.

SP Functionality

SP functionality is comparatively easier than CSP. If there is an incoming data frame, receive it.

`SP.c` reads an input data line from its simulation input data file (`StationProcess.txt`) and format it as a data frame.

The contents of the input file are read into a buffer line by line.

The destination station to which the data frame is to be sent is mentioned in the input file itself.

It then sends a request frame to the CSP asking permission to send its data frame just formatted and wait for reply.

If a reject reply comes from then CSP, it re-transmits the request frame.

If it a positive reply comes from then CSP, it will then proceed to send data frame to the CSP.

With the help of the `socket` function, a socket is created once the buffer has read a line from the input file.

Connection is initiated with the `connect` function from the socket with file descriptor `sockfd` to the destination socket whose address is specified by the `serv_addr` and `sizeof(serv_addr)` arguments. Finally, it writes the part of frame into the socket `sockfd`.

Multiplexing

Multiplexing is implemented on the CSP and SP which runs multiple input events simultaneously.

The CSP is able to receive request/data frames from multiple SP, send replies, and forward data frames, at the same time.

The SP on the other terminal sends data/request frames, read from input file, and receive data frames sent by other SPs, at the same time using `sendFrame(char *rdbuff, int part, int I)`

Binary Exponential Backoff (BEBO) algorithm

BEBO algorithm is implemented in this project.

When collision occurs (when two or more frames from different stations collide on the bus). It can occur either during the sending period, or the uncertain period (in which the station has sent out the frame but is not sure if the frame has arrived safely to its destination station).

A station will stop transmission immediately if it detects the collision during the sending period. It then waits for a random period of time before trying re-transmission.

After the first collision, the time is divided into discrete slots whose length is equal to the worst-case round-trip propagation time.

After the first collision, a station waits for either 0, or 1 time slots before at-tempting its first re-transmission.

If the first re-transmission collides too, it will wait for either 0, 1, 2, or 3 time slots before the second re-transmission.

In general, after the i th collision, where $i \leq 10$, a station will wait for a random number of slot times chosen from the interval from 0 to 2 i − 1. However, after 10 collision, the random interval is fixed between 0 and 2 10 − 1 = 1023 time slots.

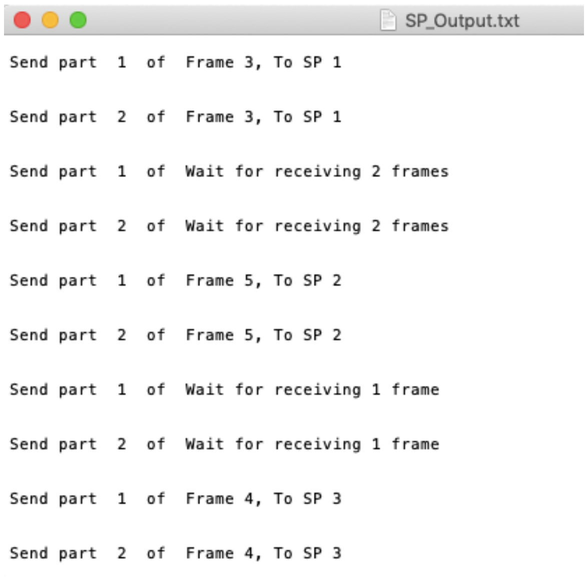
After 16 collisions, the Ethernet controller for a station suspends re-transmissions and reports the failure to the data link layer.

Implementation

Files included

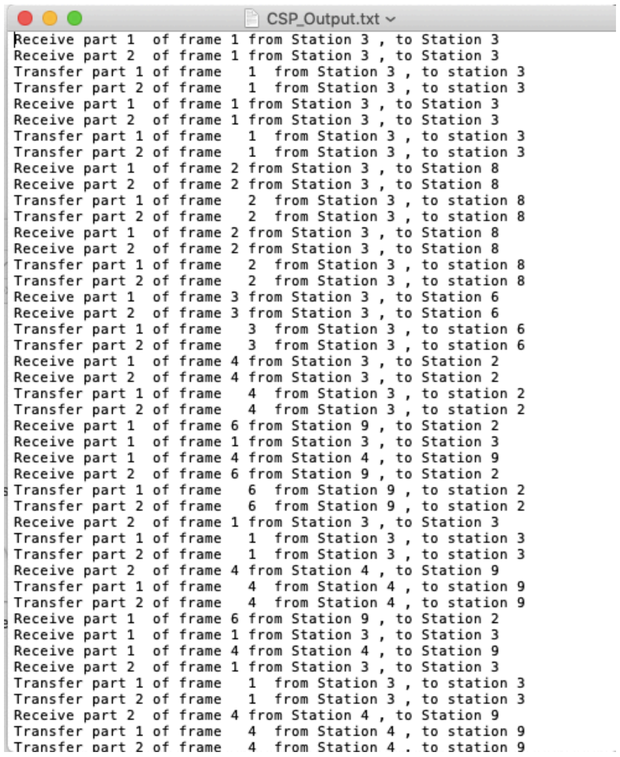
- CSP.c (Server Code)
- CSP (executable file)
- CSP_Output.txt (CSP activity log file)
- SP.c (Client Code)
- SP (executable file)
- SP_Output.txt (SP activity log file)
- Station Process input text files
    - StationProcess1.txt
    - StationProcess2.txt
    - StationProcess3.txt
    - StationProcess4.txt
    - StationProcess5.txt
    - StationProcess6.txt
    - StationProcess7.txt
    - StationProcess8.txt
    - Stationprocess9.txt
    - Stationprocess10.txt

The result of the simulation that is the activity log files is maintained by each SP with the name `SP_Output.txt` and CSP by `CSP_Output.txt`



```
Send part  1  of  Frame 3, To SP 1

Send part  2  of  Frame 3, To SP 1

Send part  1  of  Wait for receiving 2 frames

Send part  2  of  Wait for receiving 2 frames

Send part  1  of  Frame 5, To SP 2

Send part  2  of  Frame 5, To SP 2

Send part  1  of  Wait for receiving 1 frame

Send part  2  of  Wait for receiving 1 frame

Send part  1  of  Frame 4, To SP 3

Send part  2  of  Frame 4, To SP 3
```

FIGURE 1. SP OUTPUT



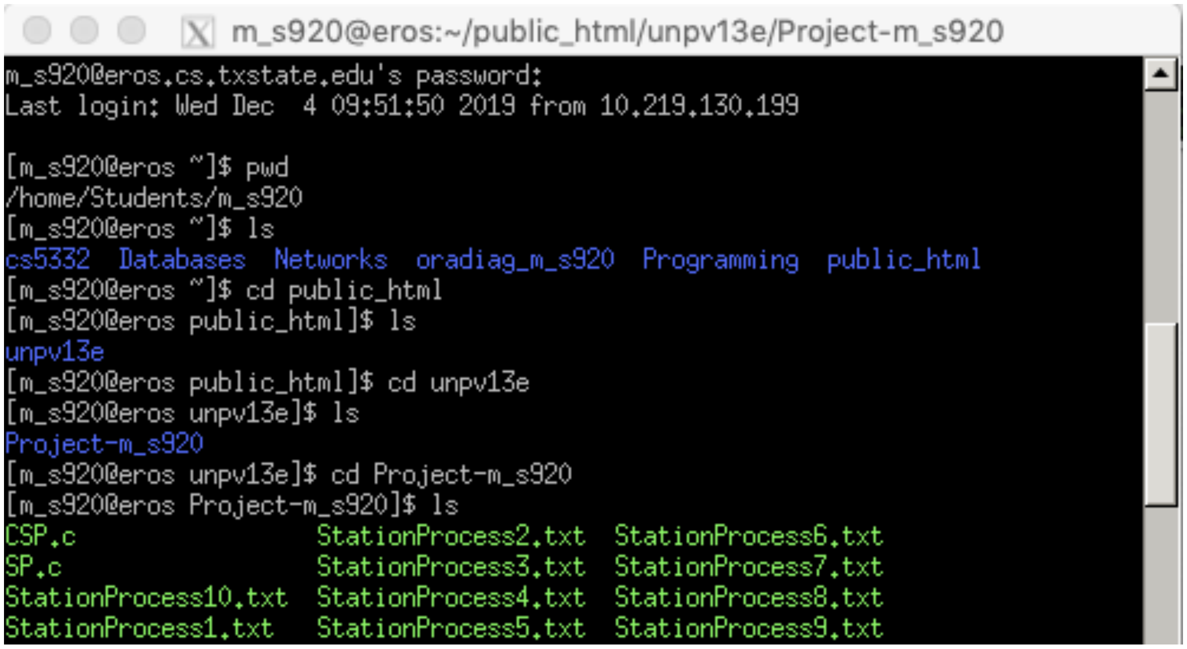
```
Receive part 1  of frame 1 from Station 3 , to Station 3
Receive part 2  of frame 1 from Station 3 , to Station 3
Transfer part 1 of frame   1  from Station 3 , to station 3
Transfer part 2 of frame   1  from Station 3 , to station 3
Receive part 1  of frame 1 from Station 3 , to Station 3
Receive part 2  of frame 1 from Station 3 , to Station 3
Transfer part 1 of frame   1  from Station 3 , to station 3
Transfer part 2 of frame   1  from Station 3 , to station 3
Receive part 1  of frame 2 from Station 3 , to Station 8
Receive part 2  of frame 2 from Station 3 , to Station 8
Transfer part 1 of frame   2  from Station 3 , to station 8
Transfer part 2 of frame   2  from Station 3 , to station 8
Receive part 1  of frame 2 from Station 3 , to Station 8
Receive part 2  of frame 2 from Station 3 , to Station 8
Transfer part 1 of frame   2  from Station 3 , to station 8
Transfer part 2 of frame   2  from Station 3 , to station 8
Receive part 1  of frame 3 from Station 3 , to Station 6
Receive part 2  of frame 3 from Station 3 , to Station 6
Transfer part 1 of frame   3  from Station 3 , to station 6
Transfer part 2 of frame   3  from Station 3 , to station 6
Receive part 1  of frame 4 from Station 3 , to Station 2
Receive part 2  of frame 4 from Station 3 , to Station 2
Transfer part 1 of frame   4  from Station 3 , to station 2
Transfer part 2 of frame   4  from Station 3 , to station 2
Receive part 1  of frame 6 from Station 9 , to Station 2
Receive part 1  of frame 1 from Station 3 , to Station 3
Receive part 1  of frame 4 from Station 4 , to Station 9
Receive part 2  of frame 6 from Station 9 , to Station 2
Transfer part 1 of frame   6  from Station 9 , to station 2
Transfer part 2 of frame   6  from Station 9 , to station 2
Receive part 2  of frame 1 from Station 3 , to Station 3
Transfer part 1 of frame   1  from Station 3 , to station 3
Transfer part 2 of frame   1  from Station 3 , to station 3
Receive part 2  of frame 4 from Station 4 , to Station 9
Transfer part 1 of frame   4  from Station 4 , to station 9
Transfer part 2 of frame   4  from Station 4 , to station 9
Receive part 1  of frame 6 from Station 9 , to Station 2
Receive part 1  of frame 1 from Station 3 , to Station 3
Receive part 1  of frame 4 from Station 4 , to Station 9
Receive part 2  of frame 1 from Station 3 , to Station 3
Transfer part 1 of frame   1  from Station 3 , to station 3
Transfer part 2 of frame   1  from Station 3 , to station 3
Receive part 2  of frame 4 from Station 4 , to Station 9
Transfer part 1 of frame   4  from Station 4 , to station 9
Transfer part 2 of frame   4  from Station 4 , to station 9
```

FIGURE 2. CSP OUTPUT

Testing

The project is implemented on a CS department Linux lab computer, in order to test the project should be direct to the path

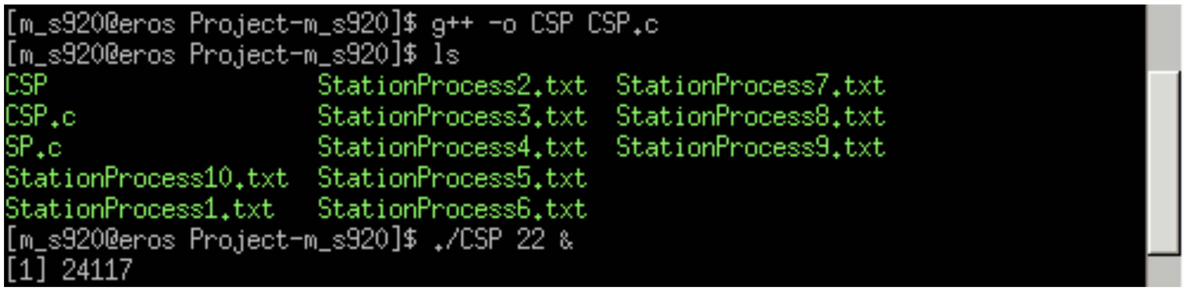
/home/Students/m_s920/public_html/unpv13e/Project-m_s920



```
m_s920@eros:~/public_html/unpv13e/Project-m_s920
m_s920@eros.cs.txstate.edu's password:
Last login: Wed Dec  4 09:51:50 2019 from 10.219.130.199

[m_s920@eros ~]$ pwd
/home/Students/m_s920
[m_s920@eros ~]$ ls
cs5332  Databases  Networks  oradiag_m_s920  Programming  public_html
[m_s920@eros ~]$ cd public_html
[m_s920@eros public_html]$ ls
unpv13e
[m_s920@eros public_html]$ cd unpv13e
[m_s920@eros unpv13e]$ ls
Project-m_s920
[m_s920@eros unpv13e]$ cd Project-m_s920
[m_s920@eros Project-m_s920]$ ls
CSP.c                  StationProcess2.txt  StationProcess6.txt
SP.c                   StationProcess3.txt  StationProcess7.txt
StationProcess10.txt   StationProcess4.txt  StationProcess8.txt
StationProcess1.txt    StationProcess5.txt  StationProcess9.txt
```

FIGURE 3

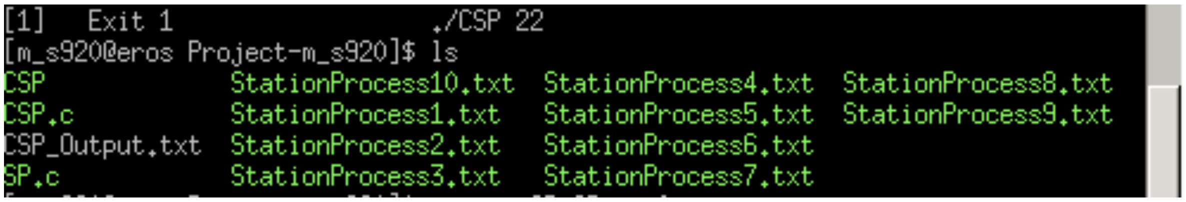
Then you should run the CSP.c file, executing it gives out the executable file named 'CSP', which is already copied under the folder.



```
[m_s920@eros Project-m_s920]$ g++ -o CSP CSP.c
[m_s920@eros Project-m_s920]$ ls
CSP                    StationProcess2.txt  StationProcess7.txt
CSP.c                  StationProcess3.txt  StationProcess8.txt
SP.c                   StationProcess4.txt  StationProcess9.txt
StationProcess10.txt   StationProcess5.txt
StationProcess1.txt    StationProcess6.txt
[m_s920@eros Project-m_s920]$ ./CSP 22 &
[1] 24117
```

FIGURE 4ª

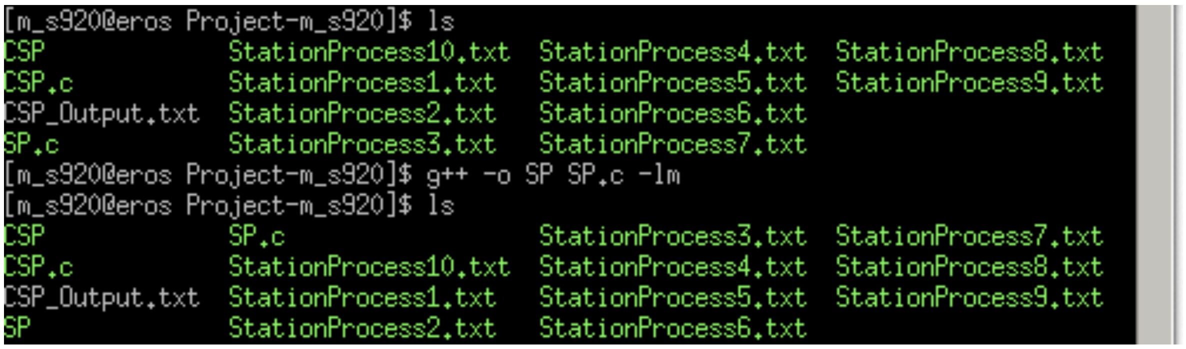
The next step is to run the CSP.c file, to get the server ready



```
[1]    Exit 1                    ./CSP 22
[m_s920@eros Project-m_s920]$ ls
CSP            StationProcess10.txt  StationProcess4.txt  StationProcess8.txt
CSP.c          StationProcess1.txt   StationProcess5.txt  StationProcess9.txt
CSP_Output.txt StationProcess2.txt   StationProcess6.txt
SP.c           StationProcess3.txt   StationProcess7.txt
```

FIGURE 4B

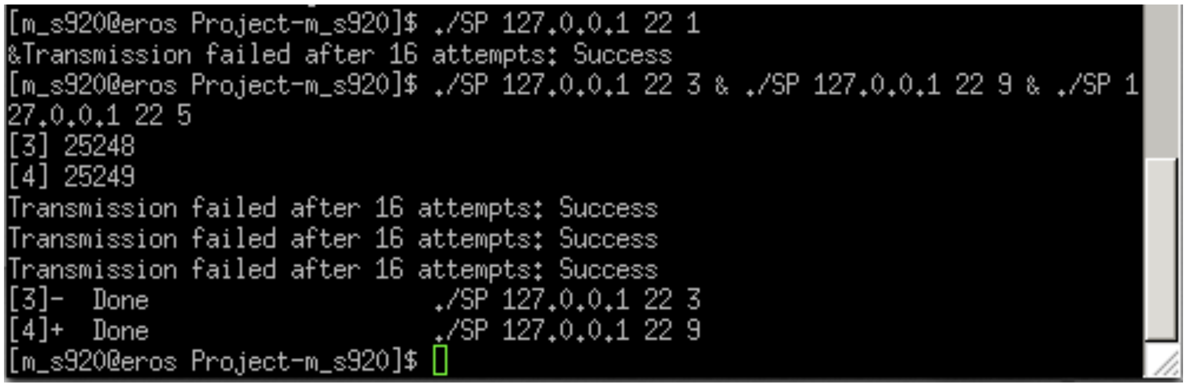
The you must need to compile SP.c, executing it gives out the executable file named 'SP', which is already copied under the folder



```
[m_s920@eros Project-m_s920]$ ls
CSP             StationProcess10.txt  StationProcess4.txt  StationProcess8.txt
CSP.c           StationProcess1.txt   StationProcess5.txt  StationProcess9.txt
CSP_Output.txt  StationProcess2.txt   StationProcess6.txt
SP.c            StationProcess3.txt   StationProcess7.txt
[m_s920@eros Project-m_s920]$ g++ -o SP SP.c -lm
[m_s920@eros Project-m_s920]$ ls
CSP             SP.c                  StationProcess3.txt  StationProcess7.txt
CSP.c           StationProcess10.txt  StationProcess4.txt  StationProcess8.txt
CSP_Output.txt  StationProcess1.txt   StationProcess5.txt  StationProcess9.txt
SP              StationProcess2.txt   StationProcess6.txt
```

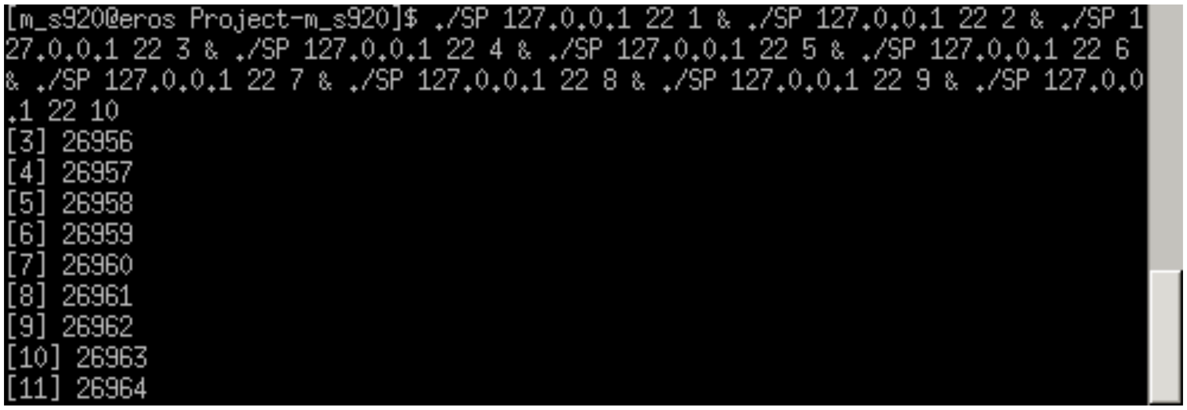
FIGURE 5ª

And then run the SP.c file



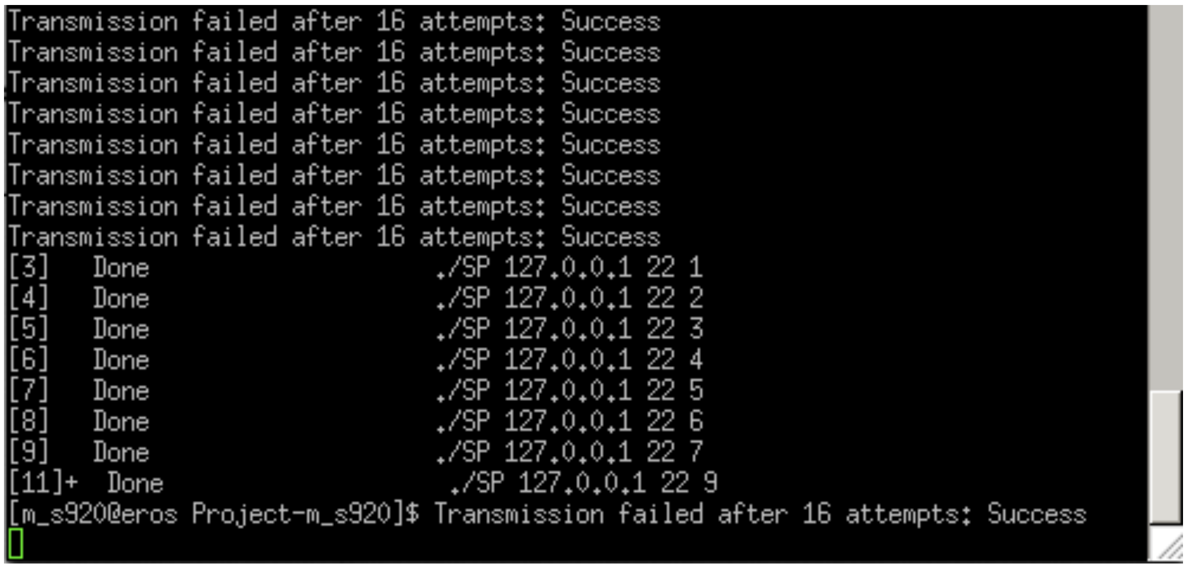
```
[m_s920@eros Project-m_s920]$ ./SP 127.0.0.1 22 1
&Transmission failed after 16 attempts: Success
[m_s920@eros Project-m_s920]$ ./SP 127.0.0.1 22 3 & ./SP 127.0.0.1 22 9 & ./SP 1
27.0.0.1 22 5
[3] 25248
[4] 25249
Transmission failed after 16 attempts: Success
Transmission failed after 16 attempts: Success
Transmission failed after 16 attempts: Success
[3]-  Done                    ./SP 127.0.0.1 22 3
[4]+  Done                    ./SP 127.0.0.1 22 9
[m_s920@eros Project-m_s920]$ 
```

FIGURE 6



```
[m_s920@eros Project-m_s920]$ ./SP 127.0.0.1 22 1 & ./SP 127.0.0.1 22 2 & ./SP 1
27.0.0.1 22 3 & ./SP 127.0.0.1 22 4 & ./SP 127.0.0.1 22 5 & ./SP 127.0.0.1 22 6
& ./SP 127.0.0.1 22 7 & ./SP 127.0.0.1 22 8 & ./SP 127.0.0.1 22 9 & ./SP 127.0.0
.1 22 10
[3] 26956
[4] 26957
[5] 26958
[6] 26959
[7] 26960
[8] 26961
[9] 26962
[10] 26963
[11] 26964
```

FIGURE 6ª



```
Transmission failed after 16 attempts: Success
Transmission failed after 16 attempts: Success
Transmission failed after 16 attempts: Success
Transmission failed after 16 attempts: Success
Transmission failed after 16 attempts: Success
Transmission failed after 16 attempts: Success
Transmission failed after 16 attempts: Success
Transmission failed after 16 attempts: Success
[3]   Done                    ./SP 127.0.0.1 22 1
[4]   Done                    ./SP 127.0.0.1 22 2
[5]   Done                    ./SP 127.0.0.1 22 3
[6]   Done                    ./SP 127.0.0.1 22 4
[7]   Done                    ./SP 127.0.0.1 22 5
[8]   Done                    ./SP 127.0.0.1 22 6
[9]   Done                    ./SP 127.0.0.1 22 7
[11]+  Done                    ./SP 127.0.0.1 22 9
[m_s920@eros Project-m_s920]$ Transmission failed after 16 attempts: Success
```

FIGURE 6B

References

1. A.S.Tanenbaum and D.J.Wetherall, Computer Networks (5th ed.). Prentice-Hall, 2011. ISBN13: 978-0-13-212695-3.

2. W. R. Stevens, Bill Fenner, and Andrew M. Rudoff. UNIX Network Programming – Networking APIs: Sockets and XTI (3rd ed.). Addison-Wesley, 2004. ISBN: 0-13-141155-1.

3. Material taken from class lectures