Version 42.00 – 29/12/2019

Final Project: AsciiFlix Streamer App

Due date: Jan 24, 2020

Contents

1 Introduction

You will be implementing a “simple” Internet Video streaming service that streams "video" using multicast in a single AS. You will also implement a simple client that will connect to the server, play movies, receive data from the service server, and even connect to the server directly to have direct control over the service (we'll refer to this as the premium service). The purpose of this assignment is for you to become familiar with sockets and threads, and to get you used to thinking about network protocols. To get a good feel for the project you should use the provided demo programs and look at the class presentation.

2 Protocol

This assignment has two main parts:

1. The server, which handle connected clients and streams videos to multicast groups.
2. The client connects to the server and receives 'video'. The client programs will be referred as "*Client*".

There are two kinds of data being sent between the server and the client:

1. “control” data, data that contains information about the videos, connection information, etc.
2. “video” data, which the server reads from .txt files and streams to a multicast group or a dedicated TCP socket.

You will be using TCP for the control data and UDP for the video data. In the more advance part of the project you will also send video data using TCP.
The client-server “control” communication, is done by passing messages over a TCP connection, this is basically the protocol.
The following sections describe the messages involved, what is the role of each message in the protocol, and the timeouts between each event and message.

2.1 Client to Server Commands

The client sends the server messages called ***commands***. There are five commands the client can send to the server, in the following format.
In each message the first field is a single byte that describes the message type, it is **constant** in all messages of the same type.

Hello:

```
uint8_t     commandType = 0;
uint16_t    reserved = 0;
```

AskFilm:

```
uint8_t     commandType = 1;
uint16_t    stationNumber;
```

GoPro:

```
uint8_t     commandType = 2;
uint16_t    reserved=0;
```

SpeedUp:

```
uint8_t     commandType = 3;
uint8_t     speed;
```

***Release**:*

```
uint8_t     commandType = 4;
uint16_t    reserved = 0;
```

A ***uint8_t***[1] is an unsigned 8-bit integer. A ***uint16_t*** is an unsigned 16-bit integer.

2.1.1 A Brief Overview for each Message Type in the Client

- A ***Hello*** command is sent when the client connects to the server.
  - ***reserved**:* This field is always set to zero.
- An ***AskFilm*** command is sent by a client to inquire about a video that is played in a station.
  - ***stationNumber***: The number of the station/channel we inquire about.
- A ***GoPro*** command is sent by the client when it wants to start receiving the video stream using a dedicated TCP socket.
  - ***reserved**:* This field is always set to zero
- A ***SpeedUp*** command is sent by a client that is connected to the TCP video service, when it wants to increase the speed of the movie.
  - ***Speed***: a number from 0 to 100 that will increase the video speed by 0% to 100%.
- A ***Release*** command is sent by a client that is connected to the TCP video service, and wants to leave back to the regular service.
  - ***reserved**:* This field is always set to zero

2.2 Server to Client Replies

There are **five** possible messages[2] called replies the server may send to the client:

***Welcome**:*

```
uint8_t     replyType = 0;
uint16_t    numStations;
uint32_t    multicastGroup;
uint16_t    portNumber;
uint8_t     row;
uint8_t     col;
```

***Announce**:*

```
uint8_t     replyType = 1;
uint8_t     row;
uint8_t     col;
uint8_t     filmNameSize;
char        filmName [filmNameSize];
```

***PermitPro**:*

```
uint8_t     replyType = 2;
uint8_t     permit;         // 1 || 0
uint16_t    portNumber; //if permit =1, else portNumber=NULL
```

[1] You can use these types from C if you `#include <inttypes.h>`.

[2] These messages appear here in a form similar to *c structs* for convenience only, you don't have to use structs!

Ack:

```
uint8_t     replyType = 3;
uint8_t     toWhat;
```

InvalidCommand:

```
uint8_t     replyType = 4;
uint8_t     replyStringSize;
char        replyString[replyStringSize];
```

2.2.1 A Brief Overview for each Message Type in the Server

- A ***Welcome*** reply is sent in response to a ***Hello*** command.
  - ***numStations:*** The number of stations at the server. Stations are numbered sequentially from 0, so a "***numStations"*** =30 means that stations from 0 through 29 are valid.
  - ***multicastGroup*****:** Indicates the multicast IP address used for station 0. Station 1 will use the multicast group ***multicastGroup*** +1, and so on.
  - ***portNumber***: The multicast port indicates the port number to listen to. All stations will be using the same port!

- An ***Announce*** reply is sent after a client sends a ***AskFilm*** command about a ***stationNumber***.
  - ***row & col***: the number of rows and columns in a film's frame.
  - ***filmNameSize:*** Represents the length of the filename, in bytes.
  - ***filmName:*** The name of the film itself. It is a string that contains the filename itself in ASCII.[3]

- A ***PermitPro*** is sent as a reply for an ***GoPro*** message. It either approves or disapproves the client to join the premium service.
  - ***Permit***: The answer can either be a positive reply (set to 1) or a negative reply (set to 0). The server first test if it has room for another user.
  - ***portNumber****:* if permit is zero the reply will include a port number that the client will use to connect to the server on the premium TCP socket. Otherwise, this field will not exist.

- An ***ACK*** is sent as a reply for a ***SpeedUP*** or ***Release*** message. It signals the client that messages have been processed by the server successfully
  - ***toWhat***: This byte denotes the type of command this ack is meant to acknowledge

- An ***InvalidCommand*** reply is sent in response to invalid conditions from the client.
  - ***replyStringSize:*** Represents the length of the ***replyString***, in bytes.
  - ***replyString:*** A message string which contains the error string itself. The string must be formatted in ASCII and must **not be null-terminated**.

[3] The string needs to be formatted in ASCII and must **not be null-terminated**. To ***announce*** a film called "*Roar*.txt", your client would send the ***replyType*** byte, followed by two bytes for a the row and col and a byte whose value is 8 followed by the 8 bytes who are the ASCII character values of the film name *"Roar.txt"*.

2.3 Server-Client Messages, field Specifications

We can't just send bytes to the network in any way we want, we need to agree on how each field is formatted as well. Each message is sent in a single buffer.
The order in which the fields are set in the message is the same as the order presented in this document.
The length of each field is according to the field size, i.e. uint16_t is two bytes long, etc...
For each of the **integer fields** in each message in your programs you **MUST** use network byte order[4], that is, change your message fields to the network byte order and back again when you receive them. Your programs will send exactly the number of bytes as in the command format. So, to send a ***Hello*** command, your client would send exactly three bytes to the server. Any other amount is a violation of the protocol.
For example, to send an ***Announce*** message using a buffer "buff" we set the first byte to 1(this message type), the next two bytes are filled with the number of stations in the server(the number of different films) after we used htons() on it. The next four are entered using htonl(); The next two with hotons(), and the last two bytes are set regularly. The message length will be 12 bytes. We can summarize this briefly in the following **pseudo** code:

```
buff[0] =0;//message type
buff[1 to 2] =hotns(numStations) );// this field is more than one byte long
buff[3 to 7] =htonl(multicastGroup);// this field is more than one byte long
buff[7 to 9] =htons(portNum);//
buff[10] =row;
buff[11] =col;
sendToClient(buff,12)
```

2.4 Server-Client Protocol basics

In our project, the server and client program have a basic protocol with which they interact. It operates under a TCP connection.
Whenever a client sends a control message(command) to the server, the server should (under the protocol) always send a proper reply. There is always one single proper replay type it can send, any other type is a violation of the protocol. The commands that a client sends must be related to its state in server, for example a freshly connected client must start with a Hello not an AskFilm! Only premium users can send premium commands, etc.
Let us divide the protocol into *three phases*.

1. The first, *initial connection*, or *handshake* when a client first tries to connect to a server.
2. second, *established connection*, after the connection had been established.
3. The third, the *premium connection* phase, after a client has entered a premium channel.

2.4.1 Initial Connection:

Here, the client can(and needs to) send **one** type of command normally:

1. The client will send a ***Hello***. ←→ The server should replay with a ***Welcome***.
    a. The client parses the message and the protocol moves into to the *established connection* phase.

2.4.2 Established Connection

Here, the client can send **two** types of commands normally:
1) An ***AskFilm*** command. ←→The server will reply with an ***Announce***.
    a) The client prints the film name on the screen and changes the station.

[4] Use the functions htons, htonl, ntohs and ntohl to convert from network to host byte order and back.

b) The channel ***AskFilm*** command specified in the must be in the range of the valid channels.

2) A ***GoPro*** command ←→ The server will replay with a ***PermitPro***.
   a) If the permit is negative, the client will remain in the *established connection* phase.
   b) If the permit is positive, the client connects to the premium TCP socket and the protocol transitions into the *premium connection* phase.
   c) The film transmitted on the TCP socket must be the last film the client was watching.

2.4.3 Premium Connection

Here, The client can send **three** types of commands normally:

1) An ***AskFilm*** command ←→ The server will reply with an ***Announce***.
   a) The client prints the film name on the screen and the server will changes the film.
   b) The channel ***AskFilm*** command specified in the must be in the range of the valid channels.
   c) The new film always starts form the beginning of the file!
2) A ***SpeedUp*** command. ←→ The server will reply with an ***ACK*** , with its *toWhat* field set to the command type of ***SpeedUP***.
   a) The server should change the speed of the sent film.
   b) The valid speed range is 0 to 100.
3) A ***Release*** command. ←→ The server will reply with an ***ACK*** , with its *toWhat* field set to the command type of ***Release***.
   a) The server will close TCP connection for the premium socket and allow new users to the premium channel.
   b) The client returns to the *established connection* phase. It will watch last channel it watched before sending the Release command.

In any case of an error, that is, a message is sent out of order, wrong format or a timeout occurred, the server will send an ***InvalidCommand*** message to the client. Consequently, the client needs to terminate and the server closes the connection with the client and removes it from the server. For example, a ***Hello*** message is sent during the *established connection* phase. See the following figure for an example (Fig 2.5.1).

Whenever the server detects that the client terminated, it will close the connection and remove it from the server. On the other hand, when the client detects connection errors/protocol errors, the client closes the connection and terminates, without directly sending any message to the server. The server still need to detect that the client is disconnected.

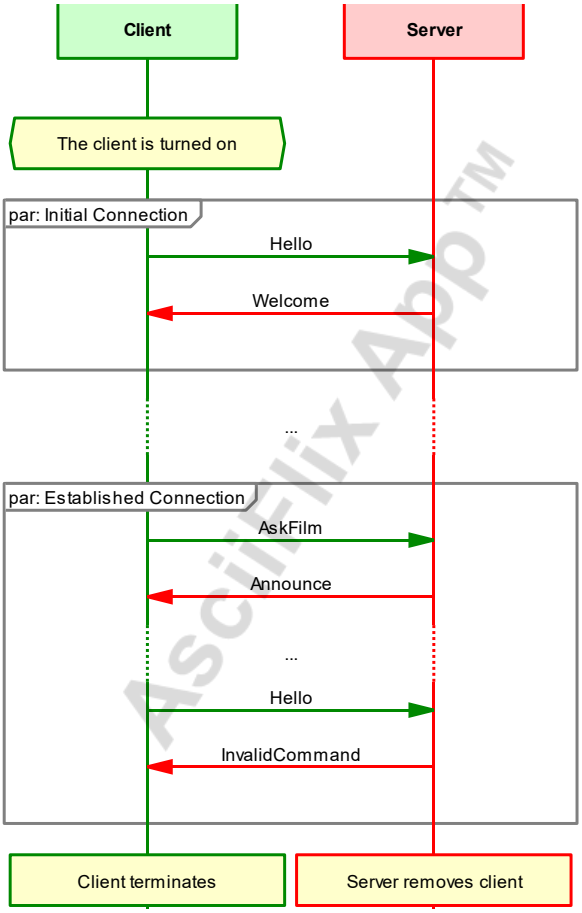

Fig2.5.1: Sample of the server-client protocol (ignore the word “par”).

2.7 Invalid Conditions

Since neither the client nor the server may assume that the program with which it is communicating is compliant with this specification, they must both be able to behave correctly when the protocol is used incorrectly.

2.7.1 Server

On the server side, an ***InvalidCommand*** reply is sent in response to any invalid command. ***replyString*** should contain a brief error message explaining what went wrong. Give helpful strings stating the reason for failure. If an ***AskFilm*** command was sent with 1729 as the ***stationNumber***, a bad ***replyString*** is “Error, closing connection.”, while a good one is “Station 1729 does not exist”.
To simplify the protocol, whenever the server receives an invalid command, it **MUST** reply with an ***InvalidCommand*** and then close the connection to the client that sent it.

An Invalid command may happen in the following situations:

- ***AskFilm***
  - The station given does not exist.
  - The command was sent before a ***Hello*** command was sent. The client must send a ***Hello*** command before sending any other commands.
- ***SpeedUp***
  - The speed value is greater than 100.

- ***Hello***
  - More than one ***Hello*** command was sent. Only one should be sent, at the very beginning.
    - Note that the *best practice* is to disconnect the client on any error, that is, disconnect the client whenever any message is sent out of place, however, you will only be tested on these situations.
- **Any case where a message is sent to the server when not in its proper protocol phase.**
- ***Any timeout.***

2.7.2 Client

On the client side, invalid uses of the protocol **MUST** be handled simply by disconnecting and printing the appropriate reason. This could for example happens in the following situations:

- ***Announce***
  - The server sends an ***Announce*** before the client has sent an ***AskFilm***.
  - The server answers an ***AskFilm*** with a ***PermitPro*** message.
- ***Welcome***
  - The server sends a ***Welcome*** before the client has sent a ***Hello***.
  - The server sends more than one ***Welcome*** at any point (not necessarily directly following the first ***Welcome***).
- ***InvalidCommand***
  - The server sends an ***InvalidCommand***. This may indicate that the client itself is incorrect, or the server may have sent it out of error. In either case, the client **MUST** print the ***replyString*** and disconnect.
- ***PermitPro***
  - The server sends a ***PermitPro*** to the client before the client has sent an ***GoPro*** to the server.
  - The server answers a ***PermitPro*** with an ***Announce*** message.
- An unknown response was sent (one whose ***replyType*** was not 0, 1, 2,3 or 4).
- A command that sent with unexpected length (not in the format we mentioned before).
- **Any case where a message is sent to the server when not in it's proper protocol phase.**
- ***Any timeout*** .
- Note that the *best practice* is to exit on any error, that is exit whenever any message is sent out of place.

2.8 Timeouts

Sometimes, a host you're connected to may misbehave in such a way that it simply doesn't send any data. In such cases, it's imperative that you can detect such errors and reclaim the resources consumed by that connection. Considering this, there are a few cases in which you will be required to time-out a connection if data isn't received after a certain amount of time, in this project, **your typical timeout should set to 300 milliseconds**. Use #Define to include it in your code.
These timeouts should be treated as errors just like any other I/O or protocol errors you might have and handled accordingly. They must be taken to only affect the connection in question, and not unrelated connections (this is obviously more of a problem for the server than for the client, that also needs to send a suitable invalid command message). Some examples of timeouts are:

- A timeout MAY occur, for example, in any of the following circumstances:

- If a client connects to a server, and the server does not receive a ***Hello*** command within some preset amount of time, the server SHOULD time out that connection.
- If a client connects to a server and sends a ***Hello*** command, and the server does not respond with a ***Welcome*** reply within the preset amount of time, the client will time out that connection.
- If a client has completed a handshake (the initial connection phase) with a server, and has sent an ***AskFilm*** command, and the server does not respond with an ***Announce*** reply within some preset amount of time, the client SHOULD time out that connection.
- For an established connection between a client and a server, if the client sends an ***GoPro*** message and the server does not respond within 300 milliseconds.
- For an established connection between a client and a server, if the client sends an ***GoPro*** message and the server respond with a positive ***PermitPro***, but the client does not connect to the premium socket within 300 milliseconds.

A timeout **has to occur** in any circumstance where either the client or the server program expects a certain type of message (replay or command) and does not receive it within 300 milliseconds.

A timeout **should not occur** in due to **film data** messages not arriving for some time.

Work alongside your FSM and planning papers when implementing the protocol!

3 Implementation Requirements and Details

to help you become familiar with the Berkeley sockets API, you must implement this project in *C* The project MUST work on the laboratory computers using your multicast topology (GNS3) and using the virtual box machines (pc1, …, pc4).
Your programs MUST work well with the programs which we will supply, these will be a part of your defense.

3.1 Correctness

You will write two separate programs, each of which will interact with the user to varying degrees. It is your responsibility to sanitize all input. Your programs **MUST NOT** do anything which is disallowed by this specification, even if the user asks for it. The choice of how you deal with this is yours, but you must display an error message to the user and an implementation which behaves incorrectly, even if only when given incorrect input by the user, will be considered **incorrect**.
It is **highly recommended** that your programs contain as many informative prints as possible (you can use the example programs and see some examples) this will allow us to better evaluate your code, when your code is graded at the ‘defense, it is harder to give a good grade for a program that is uninformative.

3.2 Clients

You will write one client program, called "*Client Control*".
The *Client Control* handles the control and film data from the server. The executable **MUST** be called flix_control. Its command line **MUST** be:

```
flix_control  <server_name>  <server_port>
```

<server_name> represents the IP address (e.g. 132.72.38.158) or hostname (e.g. localhost) which the control client should connect to, and <server_port> is the TCP port to connect to.
The *Client Control* program should do the following:

- Communicate with the server according to the protocol.
  - Handel any exception from the protocol by the server by an orderly exit.
- Contain exactly two threads.
  - The first thread is responsible to handle movie data, to receive it using either a TCP or a UDP socket, and print it to the screen.
  - The second thread handles (without blocking!) data from both the user, (i.e. stdin) and the server control data (a TCP socket). It both sends requests to the server, when asked to by the user, receives and handles replays from the server. This must be implemented using select().
- When the program starts, it will start the handshake with the server, and display the film in channel 0, without further input from the user.
- Handel user input according to the specifications in the user input section.
- When sending an ask film, the program will start receiving the and playing new film.
- When changing a station, you will not close either the thread or the UDP socket.
  - If you are receiving the film via multicast, refresh the row and col parameters, leave the current multicast group, then join the new multicast group.
  - When using the TCP socket, you will only update the row and col parameters.
  - The transition should be smooth and without too much delay.
- The *Client Control* program will use no more than three sockets.
  - A UDP socket to receive film data
  - A TCP the send and receive control data.
  - And, only if needed, another TCP socket for film data, which might be closed and reopened.
- When the user quits the program, you must exit in orderly manner from the program.
  - Close the connection.
  - Print an appropriate error message for the user.
  - Release all system resources(malloc, threads, etc.).
    Remember to always close all sockets upon termination.
- The client must know when the server closes the TCP connection(s) abruptly and act accordingly(close).
- The client **will** print whatever information the server sends to it (e.g. the ***numStations*** in a ***Welcome***). Print replies in real time.

3.3 Server

The *server* executable MUST be called film_server. Its command line MUST be:
film_server <tcp_port> <tcp_prim_port> <multicast_ip> <udp_port> <file1 > <file2 > …

<tcp_port> is a port number on which the server will listen. <multicast_ip> is the IP on which the server send station number 0. < udp_port > is a port number on which the server will stream the films, <tcp_ prim _port> is the port of the premium socket. followed by a list of one or more files.

The *Server* program should do the following:

- The server will support multiple clients simultaneously .
  - The server MUST support connection and data requests from multiple clients

simultaneously.
    - These clients will be handled in accordance with the protocol. Handel any exception from the protocol by a client by sending it a proper invalid command and then removing it form the server.
    - The number of clients that *could* be connected simultaneously to the server should be no less than 50.
    - The server will start with two welcome sockets.
        - With the port <tcp_prim_port> for premium users.
        - With the port <tcp_port> for all users, and the initial handshake.
    - The number of possible premium users should be specified by ……
- The server MUST never crash, even when a misbehaving client connects to it. The connection to that client MUST be terminated, and not affect other clients.
- The server will start streaming the films to multicast immediately upon starting.
    - If no clients are connected, the current position of the film will still progress. The TV doesn't stop when no one is watching.
    - The rate of the stream is described in the Rate Control section.
    - There MUST be no hard-coded limit to the number of film channels your server can support.
    - The first station belongs to the <multicast_ip> group. The second channel is transmitted via the following group <multicast_ip>+1, and so on for each channel.
    - Each station **will** loop its film indefinitely. Forever.
- Handel user input according to the specifications in the user input section.
- When the user quits the program, you must exit in orderly manner from the program.
    - Close the connections.
    - Release all system resources (malloc, threads, etc.). Remember to always close all sockets upon termination.
- The server must know when a client closes a TCP connection(s) abruptly and act accordingly(remove said client from the server).
- It ***your choice*** on how to implement these features.
    - With threads or select()?
    - What databases to use? What should it contain?
    - How and where to handle premium user transmission?
    - Think before you program!

3.3 Film, and Frame Format

Each film file starts with two integers in it's first line in the format:
<row>,<col>\n
<#1 frame>
<#2 frame>
….
These should be read once to infer the number of rows and columns that comprise each frame in the film.
The number of rows includes first row which contains the frame duration number.
The first row of each frame contains a single integer. This is the frame duration. This number should not be sent out to the client! You will use it in your rate control.

3.4 Printing a Frame

A ***frame*** is set of *row*(the number) rows, each row is *col* characters long. In this project we will **assume that each frame is transmitted whole**, as a single line of text. To print it, you may reassemble it as a matrix print each line, wait for the following frame before deleting the last from the screen. This can be done in a manner *similar* to this partial code snippet:

```
char frame[row][col];
... //(after the first frame)
for ( i = 0 ; I < row ; i++ )
        printf( "\033[IA\033[2K\r" ); //delete a row and move the 'cursor' up
for ( i = 0 ; I < row ; i++ ) //print the frame
       printf( frame[i] );// print a row
```

3.5 Rate Control

Normally a frame is transmitted once each 1/24th of a second. That means your program should read a single frame from the file, and transmit it once a 1/24th of a second had passed. The receiver client doesn't need to know the rate, it will merely print the frame once it is available.
If a frame has a duration parameter greater than 1, such as, for example 6, you will wait 6/24th of a second before transmitting the frame.

3.6 User Input

To have a clear way to compare and test your programs, we offer these guidelines as to how to build the user interface with each of your two programs. You should also look at our programs to see how the menus work. You do not have to emulate the exact wording of the menus.

3.6.1 Server User Input

When it's turned on, the server will prompt the user for input and wait. Upon the user entering ***"q\n".*** (that is, *'q'* followed by the "*Enter*" key), the server will exit the program in an orderly manner. Otherwise, it will ignore this input and prompt the user for input again.

3.6.2 Client User Input

When it's turned on the client will start showing the current film (after the handshake). Upon hitting the "Enter" key, the client will **stop printing** the movie to the screen and prompt the user for input than wait. It should print a menu similar to this:

```
Please enter your choice:
1.Change Movie (send AskFilm)
2. Ask For Premium (send GoPro)
3.Return to watch
4.Quit Program
```

Each option should be self-explanatory. When entering the ***AskFilm*** sub menu (1), the client will prompt the user to enter a station.
When given **premium** access to the server, the menu should be slightly different, such as this example:

```
Please enter your choice:
1.Change Movie (send AskFilm)
2.Go Faster (send SpeedUp)
3.Leave Pro (send Release)
4.Return to watch
5.Quit Program
```

Each option should be self-explanatory. When entering the ***AskFilm*** sub menu (1), the client will prompt the user to enter a station. When entering the ***Speedup*** sub menu (2), the client will prompt the user to enter a number between 0 and 100.
It is *your choice* on how exactly to handle wrong input by the user, however, it should not make your program quit unintentionally. We recommend emulating the sample programs.
recall: if the client program waits for user input, it must still be ready to handle any input by the server.

4 Testing

A good way to test your code at the beginning is to stream text. Once you're more confident of your code, you can test your client using the executable files provided to you in the Moodle site. To test film streaming you may want to slightly change your UDP listener from lab 5 so that is able to pares UDP packets into frames and print them to the screen independently.

```
./UDP_listener <multicast_group> <port>
```

You should use the binaries we provided to you to test your code and your programs, remember that these programs are not perfect, however your programs have to work with our programs. And your defense would be based on how your programs function with our programs on the lab PC.

5 Handin

- ✓ Hand in your project in the following format:
  `AsciiFlix_<ID1>_<ID2>.zip`
- ✓ Submission via Moodle.
- ✓ We should be able to rebuild your programs by running the `make` command in the terminal.
- ✓ The program **MUST** run well on your multicast topology (at GNS3) and using the VirtualBox machines (pc1, …, pc4) .
- ✓ Your programs must to work well with the executable programs, which we supplied.
- ✓ File names must be as defined in this document.
- ✓ **Due dates:**

  Design papers: during the lab hours up to 6 or 7/01/2020, (according to your hours)

  Entire project due date: ?? ??, 2020

7 Grading

Design paper: (5%)

- Client FSM and short Client design paper. **(50%)**
- Short server design paper. **(50%)**

The following are the main issues we expect your program to handle *in order of their importance*.

AsciiFlix Client: (47.5%)

1. Creating and maintaining a TCP connection, and a proper implementation of the Protocol
2. Treatment for messages outside the protocol scope and the handling of timeouts.
3. Handling film data, a proper implementation of the "*listener"* part of the client
4. Proper implementation of the premium service
5. Proper use of "select()" and threads.
6. Proper handling of user input.
7. Proper termination of the client.

AsciiFlix Server: (47.5%)

1. Maintaining active connections with multiple concurrent clients and a proper implementation of the protocol.
2. Treatment for messages outside the protocol scope and the handling of timeouts.
3. Proper implementation of the premium service.
4. Proper management and implementation of the 'film stations'(streaming over multicast).
5. Proper use of "select()" and threads.
6. Proper handling of user input, with a proper termination of the server.

8 Useful Hints/Tips

- ✓ Read this before you finish your project.
- ✓ For the TCP connection, use `recv()` and `send()` (or `read()` and `write()`). For the UDP connection, use `sendto()` and `recvfrom()`. Don't send more than 1400 bytes with one call to `sendto()`[5].
- ✓ Always read the exact number of bits you expect to read from the socket, not more.
    - o If your socket is in "blocking" mode, this might not work, consider making a none blocking socket where it is appropriate.
- ✓ To control the rate that the server sends film data at, use the `nanosleep()`function.
- ✓ Don't send a *struct* as it is. Compilers might insert padding bytes between structure members (invisible to you program, but they take space in the structure) to conform some alignment rules. Put each individual field of a structure to a raw byte-buffer manually.
- ✓ If you implement the program using more than one .c file (recommended), we encourage the use of *extern* declaration.
- ✓ Use `pthread_join()` to wait for an open thread to terminate. To exit a thread use `pthread_exit()` and not `exit()`.
- ✓ Use `select()` to implement timeouts, remember that all file descriptors that select tests will have a single timeout. For the TCP connection, timeouts *may* also be set on the socket using `setsockopt()`.
- ✓ Remember to use `perror()` to handle error messages.
- ✓ Use `memcpy()` to copy messages into arrays.
- ✓ Use `strlen()` to find the length of a string (Please don't use sizeof !)
- ✓ Source for original ascii films is from: https://www.asciimation.co.nz/asciimation/ascii_faq.html

Please let us know if you find any mistakes, inconsistencies, or confusing language!

[5] This is because the MTU of Ethernet is 1440 bytes, and we don't want our UDP packets to be fragmented.