COMPUTER NETWORK REPORT

1. Introduction

1.1 Multicast

Multicast is a method of routing data on a computer network that allows one or more senders to communicate efficiently with a group of receivers. Multicasting supports one-to-many routing, in which a single device sends data to a group of devices. It also supports many-to-many routing, in which a group sends data to a group.

Multicasting is one of the five major techniques for routing data on a computer network. The others are unicast, broadcast, anycast, and geocast. Multicasting is more efficient than broadcasting because it can target specific devices, rather than an entire network segment. It reduces unnecessary network and processor resource usage.

In IP multicasting, devices use IGMP messages to join or leave a multicast group. Any data packets sent to the multicast address for that group are distributed by the network hardware to all members of the group.

1.2 Internet-Radio

Internet radio (web radio, net radio, streaming radio, e-radio, IP radio, online radio) is a digital audio service transmitted via the Internet. Broadcasting on the Internet is usually referred to as webcasting since it is not transmitted broadly through wireless means. It can either be used as a stand-alone device running through the internet, or as a software running through a single computer.

Internet radio is generally used to communicate and easily spread messages through the form of talk. It is distributed through a wireless communication network connected to a switch packet network (the internet) via a disclosed source. Internet radio services offer news, sports, talk, and various genres of music—every format that is available on

traditional broadcast radio stations. The first Internet radio service was launched in 1993.

2. Design Decisions

2.1 Server

a. Each station is being executed on a different pthread:

- Why do we need this : As all say n number of stations need to transmit data parallelly and also we need to keep a TCP server for providing information about various radio stations. Hence we need either parallelly running threads or processes.

- Alternate option : Make diff fork() processes and make different stations work parallelly.

- Reason/argument for preferring this over other options:

If we implement this using pthread then it will be much more neater, reusable and easy to modify/add feature in main application because all information about a station will be passed to thread by making it from main function.

Hence we can easily add/remove any station at any point of time dynamically if we want to as it is running on a different thread.

We can protect and control the flow of code using mutex lock provided by pthread.

Whereas, if we use a fork then either it will be tough to add/remove them dynamically and also code will have to either repeated many times or it won't be as neat as pthread implementation.

We can easily kill a thread using main function as and when required.

Also there will lot of if else conditions to be checked in each process if we use fork.

b. Each station uses fork for running data port and information port parallely:

- Why do we need this : As we are sending information port every one second it will be better to make a different thread/process instead of checking every time if 1 sec has passed after sending previous info port packet. (Will be faster)

- Alternate option : Use pthread and make two different threads for each thread of station.

- Reason/argument for preferring this over other options:

We can make two different threads but we do not have dynamic/more amount of threads here (unlike for each station case).

We don't really need re-usability here as this function will be called multiple times for each station and we won't be calling any one part (info port) of it without calling the other part (data port)

We don't need protection about data access or flow control here.

It was easy for us to make communication b/w two forks instead of two threads (by sharing resources).

c. Should we make a different thread for TCP or keep it in main function as main function is not doing anything else as of now:

- Although it is not required as of now for our code but its always better to create a thread for it because it might be helpful later on.

For example if we plan to provide a menu for adding/removing/modifying each station then we need to use the

main function for this because the main function will be calling all threads of all stations.

2.2 Client

a. Again same as point 1 of server part we made pthread for data port ,info port and playing video instead of using fork. (better control , neatness , readability , easy to stop and start it again and again (when user clicks on stop/change station).

b. As TCP part was sequential we don't need any thread/fork for the same as it required in sequence execution with main function menu.

c. We don't need separate thread/process for menu and tcp here because main function is not supposed to do anything else on client side.

3. Client-Server Multicast

3.1 Server

3.1.1 Important Implementations

a. **Making the file streamable:** Using the following function we have made a file streamable. It's important to make the file streamable because it is a mandatory requirement for playing the file at the receiver, while simultaneously receiving it.

ffmpeg -i <inputfile.mp4> -f mpegts <streamable_output.mp4>

b. **Altering the bit-rate at runtime according to input:** Using the following function we have made the bit rate of all the input files to a

constant same value (<bps>). Also, we have set the buffer size as approximately 3/5th of the maximum bit rate in order to get proper streaming, minimizing the glitches.

ffmpeg -i <inputfile.mp4> -c:v libx264 -b:v <bps> -maxrate <1M> -bufsize <700k> <bps_output.mp4>

c. **Handling multiple stations dynamically:** We have created multiple threads throughout the server code, assigning one thread to each station. This enables every station to send data simultaneously, maintaining multiple stations.

d. **Handling multiple songs for each station dynamically:** For each station, we have given the flexibility to enter the number of songs to the user. Each song would be played sequentially and the list will rewind, once exhausted.

3.1.2 Declarations

a. Buffer Size: 4096
b. Bit Rate: Variable as per station

3.1.3 Functions Used

a. **Function Name:** station()

Description: It is a thread which takes number of songs as input from the user and validates the availability of the file.

b. **Function Name:** to_stringa()/to_unit32_t()

Description: Functions for Converting Multicast Address(unit32_t) to string and vice-versa.

3.2 Client:

3.2.1 Important Implementations

a. **Increase the receiver buffer size to the store more data :** Using setsockopt() function which set receiver buffer size so that receiver can allow for bookkeeping overhead.

b. **Joining and Withdrawing from Multicast group :** We can join multicast group by passing IP_ADD_MEMBERSHIP parameter in setsockopt() and leave a multicast group by passing IP_DROP_MEMBERSHIP.

c. **Using mutex lock to pause and resume a thread:** We are using mutex lock to pause and resume a thread and also avoiding the deadlock situation by using flags.

d. **Simultaneously writing and playing to the same file :** We are using wb+ mode so that we can create an empty file or open it for update (both for input and output).

e. We are using the three threads one thread named udp_data to receive the song on data port and second thread udp_info is used to receive information like song name, renaming time, next song name and third one is use to play the song.

3.1.2 Declarations

a. Buffer Size: 4096
b. Bit Rate: Fixed by sender
c. Receiver Buffer Size: 1 Byte

3.1.3 Functions Used

Function Name: pthread_create()

Description:

pthread_create() takes 4 arguments.

The first argument is a pointer to thread_id which is set by this function.

The second argument specifies attributes. If the value is NULL, then default attributes shall be used.

The third argument is the name of a function to be executed for the thread to be created.

The fourth argument is used to pass arguments to the function, myThreadFun.

Function Name: setsockopt()

Description:

Parameter

Socket

The socket descriptor.

Level

The level for which the option is being set.

Option_name

The name of a specified socket option.

Option_value

The pointer to option data.

Option_length

The length of the option data.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate

options at the socket level, the level parameter must be set to SOL_SOCKET as defined in sys/socket.h. To manipulate options at the IPv4 or IPv6 level, the level parameter must be set to IPPROTO_IP as defined in sys/socket.h or IPPROTO_IPV6 as defined in netinet/in.h. To manipulate options at any other level, such as the TCP level, supply the appropriate protocol number for the protocol controlling the option. The getprotobyname() call can be used to return the protocol number for a named protocol.

The option_value and option_length parameters are used to pass data used by the particular set command. The option_value parameter points to a buffer containing the data needed by the set command. The option_value parameter is optional and can be set to the NULL pointer, if data is not needed by the command. The option_length parameter must be set to the size of the data pointed to by option_value.

Function Name:pthread_mutex_lock(pthread_mutex_t **mutex*)

Description: The mutex object referenced by *mutex* is locked by calling *pthread_mutex_lock()*. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

Function Name:pthread_mutex_unlock(pthread_mutex_t **mutex*)

Description: The *pthread_mutex_unlock()* function releases the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock()* is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex

3.1.4 Challenges Faced

a. How to receive to song and information simultaneously on different ports?

   Solution: By creating two threads so that we can receive song and information from two different ports

b. How to play a song while writing in file?

   Solution: wb+ mode in opening the file in update mode

c. How to pause song?

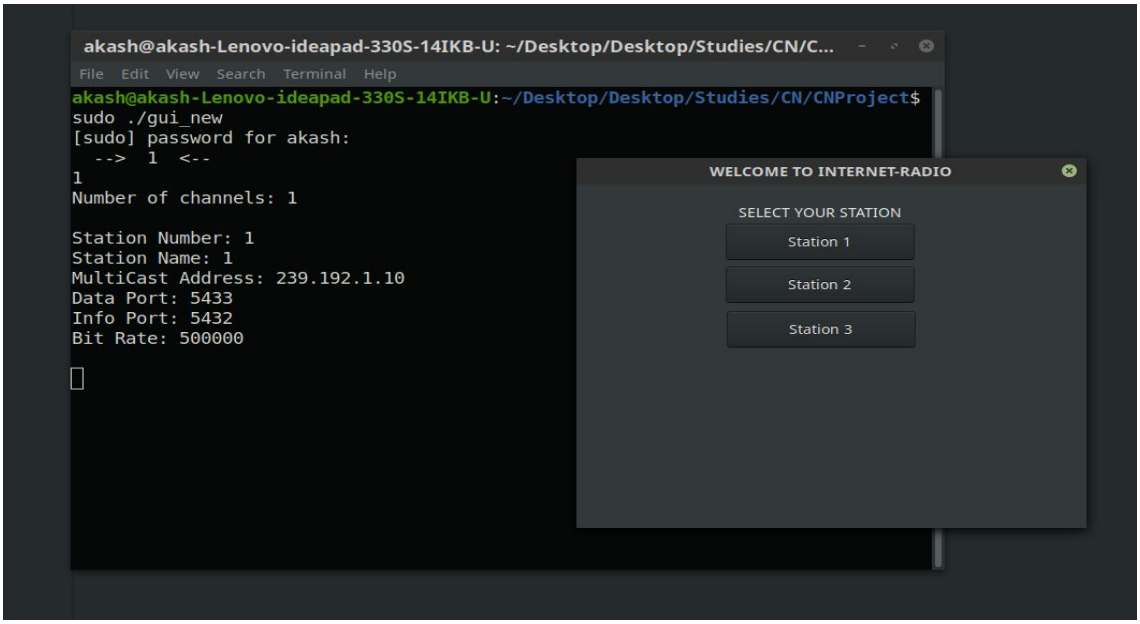
   Solution: By pausing a thread using mutex lock

d. How to change the station?

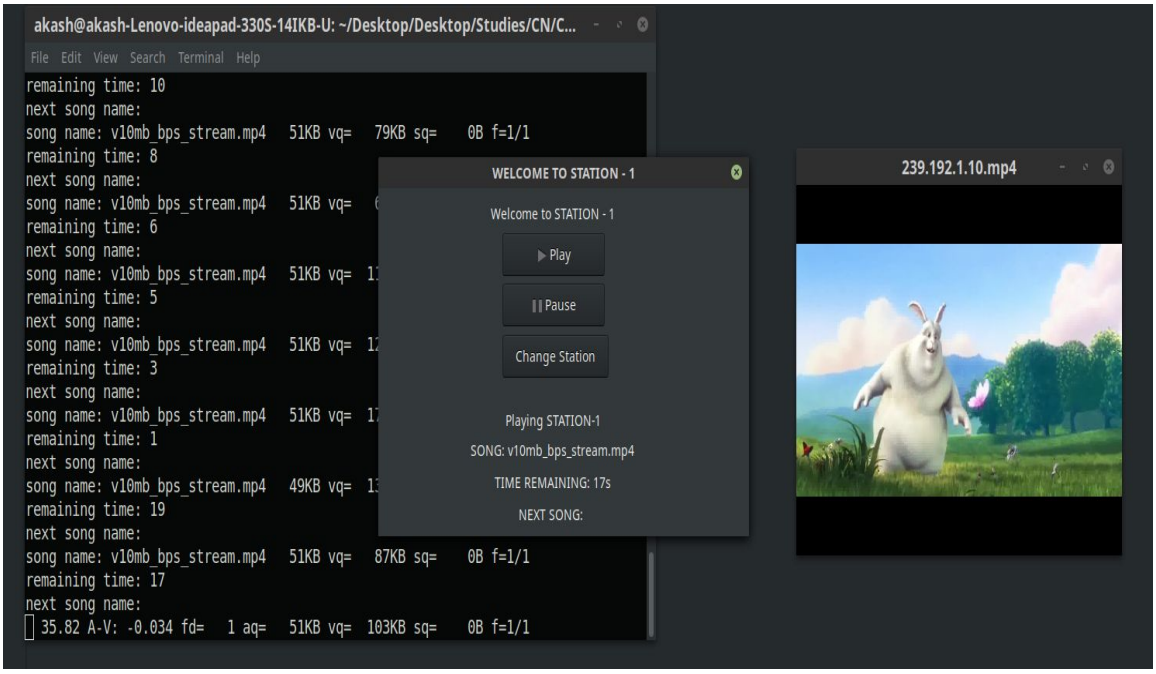
   Solution : By setsockopt() we can drop membership from one multicast group and join another multicast group.

4. Screen-shots

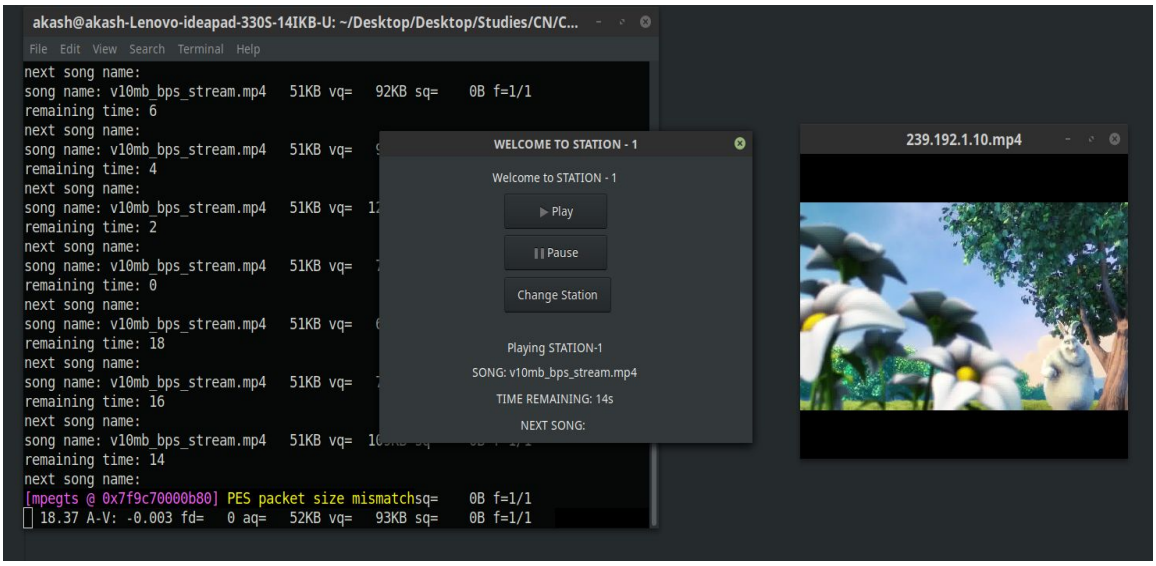
a. Main Menu of GUI displaying the stations



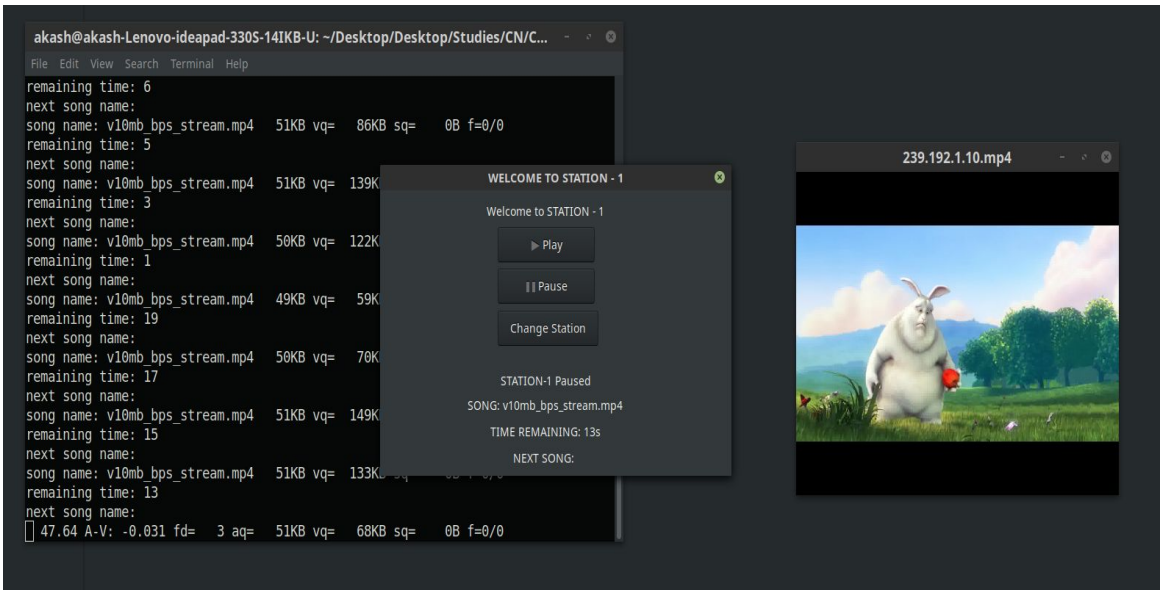
b. Entering Station-1 and clicking on Play. Time remaining = 17s (Time is being dynamically changed)



c. Time remaining = 14s



d. Station-1 is paused



5. Contributions

TASK	MEMBER
Designing of Server and Receiver and debugging of client	Smit Mandavia
Server : TCP	Smit Mandavia
Server : UDP	Akash Tike
Client	Parth Maniyar
Streamable File & Constant Bit Rate Concept	Kausha Vora
GUI	Kausha Vora
Integrating GUI with Client	Akash Tike