Assingment-5.

Group Details:

1. Adarsh Dhakar → 22CS01040
2. Avik Sarkar → 22CS01060
3. Debargha Nath → 22CS01070
4. Soham Chakraborty → 22CS02002

Github Repository Link:

https://github.com/adarshdhakar/cn_lab_sheet5/

Images Directory Link:

https://github.com/adarshdhakar/cn_lab_sheet5/images

Report.pdf Link:

https://github.com/adarshdhakar/cn_lab_sheet5/Report.pdf

Demo Video Link:

https://github.com/adarshdhakar/cn_lab_sheet5/Demo.mp4

1.
Why C++ over C ?

- Use of **string** makes it easier to handle text messages instead of relying on character arrays where we would have used strcpy() and strcat().

- Use of **set<int>** to maintain list of active clients efficiently.
  What all libraries used ? Funtionalities provided by these. #include <bits/stdc++.h>

- Use of map**<string,int>** makes is easier to maintain mapping of active clients to their respective socketfd.This allows for quick lookups, insertions, and deletions, making client management more efficient and maintainable

- All of the above functionality of **C++** which significantly simplify memory management, data structures, and overall application logic as compared to **C**, is the reason to prefer **C++ over C**

What all libraries used ? Funtionalities provided by these. #include <bits/stdc++.h>

Functions used: string, set, iostream
#include <netinet/in.h>
Provides sturctures for internet addresses **(sockaddr_in)**
#include <netdb.h> (only in Client)
Contains **gethostbyname()** to resolve hostnames to IP addresses.
#include <pthread.h>
Used for multi-threading:

- Creating threads for handling multiple clients on server.

- Creating separate threads for reading and writing for the client.

#include <unistd.h>

Used for **Posix operating system API** function such as:

Read, Write, Exit etc

Explaination of Flow and Working:

Server:

Using Threads:

1. Main Function (Server Setup):

The server initiates by creating a **server socket as any typical socket**, Upon establishing this socket, the server will make a **thread** to handle that client's communication. This ensures the server can handle multiple clients simultaneously without blocking or waiting on individual connections

2. Client Handling Thread:

Each client connection is handled by its own dedicated thread. The **Client** function is invoked within this thread to manage the client's communication.

This function:

- **Listening for Messages:** Inside the ***Client*** function, the server continuously reads from the client's socket to receive messages. Once a message is received, it is processed and a response is sent back to the client.
- **Concurrent Communication:** Since each client has its own thread, the server can communicate with multiple clients simultaneously, without blocking on any single client's request. By using **mutex_lock().**
- **Timeout thread: Client** spawns a separate **timeout monitoring thread** for each client. This thread is responsible for periodically checking the client's inactivity time and ensuring that the client does not exceed the allowed timeout limit.

  handles all functionality based on the Client request Based on Receiver name input:

Functionality provided by Server:

Initialisation:

server first ask client to enter a unique name to be identified as a active client with a unique socketfd

Funcions:

It reads the input given by Client to Server

Format:

receiver_name, message to be sent

1. **Global Chat**: **receiver_name: All** then broadcast the **"message"** to all active client's socketfd.
2. **Peer2Peer chat**: **receiver_name: Specific receiver's name**
   Then the message is sent only to that specific receiver's socketfd
3. **Room Chat/Group Chat:**
   join:room_name: creates a **new room with name as room_name** (if already not created) and joins it
   leave:room_name: closes the connection of that client
   room:room_name: sends a message to all clients in that room
4. **Exit keyword:**
   To terminate the socket connection by client side

Client:

The **Client** is responsible for handling the bidirectional communication between the client and server. To efficiently manage the flow of messages, the client utilizes **two distinct threads**—one for reading messages from the server and another for writing messages to the server. This ensures smooth, concurrent communication without blocking either operation.

1. Read Thread:

- The **read thread** is dedicated to continuously listening for messages coming from the server.
- It constantly **reads data** from the server's socket and processes or displays it accordingly.
- The **read thread** is non-blocking with respect to the write operations, allowing the client to stay responsive and receive incoming messages in real time.

2. Write Thread:

- The **write thread** handles the **sending of messages** from the client to the server.
- It waits for the user to input messages and sends them through the socket connection (using write()).
- The **write thread** is responsible for gathering and formatting user input or system-generated messages and dispatching them to the server.
- Like the **read thread**, the **write thread** operates independently, ensuring that sending messages doesn't block the reading of incoming messages.