California State University, Fresno

PROJECT REPORT

Experiment Title: Breakout Room Program

Course Title: CSCI 156

Date Submitted: 12/14/2024

Prepared By:	Sections Written:
Ricardo Navarro	**1-3, 4.1, 6,**
Mauricio Romero	**4.2, 4.3, 5, 7, 8**

Final Grade: _____

Table of Contents

1. Statement of Objectives

The objective of this project was to implement a "Breakout Room" program consisting of a server, instructor client, and multiple student clients, all developed in C++. The system facilitates communication and breakout room management in a classroom-like setting, with the server handling client connections and interactions using sockets and threading. The instructor can create and manage breakout rooms, send messages, and respond to student requests, while students can join the main room or breakout rooms, send messages, and communicate with the instructor.

The server was designed to run on a Windows machine using Visual Studio, while the instructor and student clients were tested using both Visual Studio on Windows and Xcode with the terminal on macOS. This project demonstrates the use of TCP sockets for reliable communication, threading for handling multiple clients, and command-line interaction for seamless program operation. It provides a practical implementation of client-server architecture in a distributed environment.

2. Theoretical Background

This project uses a client-server setup to manage communication between an instructor and multiple student clients. The server uses TCP (Transmission Control Protocol) to make sure messages are delivered reliably and in the right order. Each client gets its own thread, allowing multiple clients to connect and communicate at the same time. Clients must log in using a password and send commands to the server to create, join, or leave breakout rooms. Key features, like sending messages and managing breakout rooms, are handled using sockets and safe methods to share data between threads.

SERVER.CPP

SOCKET server_socket = socket(AF_INET, SOCK_STREAM, 0);

- Creates the server's TCP socket for communication.

bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)); listen(server_socket, SOMAXCONN);

- Binds the socket to port 8080 and starts listening for incoming connections.

SOCKET client_socket = accept(server_socket, NULL, NULL); thread(handle_client, client_socket, client_counter).detach();

- Accepts client connections and spawns a new thread for each.

if (strcmp(buffer, PASSWORD) != 0) { send(client_socket, "Incorrect password.\n", 22, 0); closesocket(client_socket);return; }

- Ensures only authorized clients can connect to the server.

string room_name = "breakoutRoom" + to_string(room_counter); breakout_rooms[room_name] = vector<Client>();

- Allows the instructor to create a new breakout room.

breakout_rooms[room_name].push_back(*student);

- Moves a student to a specified breakout room.

breakout_rooms.erase(room_name);

- Deletes a breakout room and moves its students back to the main chat.

broadcast_to_all_clients("Global message from Instructor: " + message);

- Sends a message from the instructor to all connected clients.

broadcast_to_room(room_name, "Student " + to_string(client_id) + ": " + message, client_socket);

- Sends a message to clients in a specific breakout room.

broadcast_to_main_chat("Student " + to_string(client_id) + ": " + message, client_socket);

- Sends a message to all clients in the main chat.

3. Experimental Procedure

3.1. Equipment Used

Hardware:

- A Windows computer for running the server, as it uses Winsock for managing sockets and client connections
- Any Mac or Windows computer with a C++ compiler installed for running the instructor and student clients.

Software:

- Server:
  - A C++ compiler (e.g., Visual Studio) for compiling and running the server code.
- Client:
  - A C++ compiler (e.g., Visual Studio on Windows or Xcode on macOS) for compiling and running the instructor and student client applications.
  - Terminal or command-line interface for executing the compiled programs.

3.2. Procedure

1. Set Up the Server:

- Windows:
  - open CSCI156_server.cpp in visual studio.
  - compile and run the server.
  - ensure it listens on port 8080 and note the IP address.

2. Set Up the Client:

- Windows:
  - open CSCI156_instructor.cpp or CSCI156_student.cpp in visual studio.
  - compile and run the program.
- macOS:

Compile the client in the terminal using:

g++ CSCI156_instructor.cpp -o instructor -pthread

g++ CSCI156_student.cpp -o student -pthread

Run the program with:

./instructor
./student

3. Update the Server IP Address (if necessary):

- Open the client file (CSCI156_instructor.cpp or CSCI156_student.cpp).

Find this line:

const char* server_ip_address = "192.168.x.x";

- Replace 192.168.x.x with the actual IP address of the server machine.

4. Connect the Client to the Server:

- Windows and macOS:
    - Start the server on the Windows machine.
    - Run the client program on any device.
    - Verify the client connects using the IP and port 8080.
    - Only one instructor can join. A second attempt will show: "An instructor is already connected."

The password is sent automatically in the client code:

send(client_fd, PASSWORD, strlen(PASSWORD), 0);

5. Test Multi-Client Communication:

- Run multiple student clients and one instructor client.
- Messages sent by one client are received by all others.
- Instructor commands like /sendGlobal work.
- Messages from students show twice for the instructor.

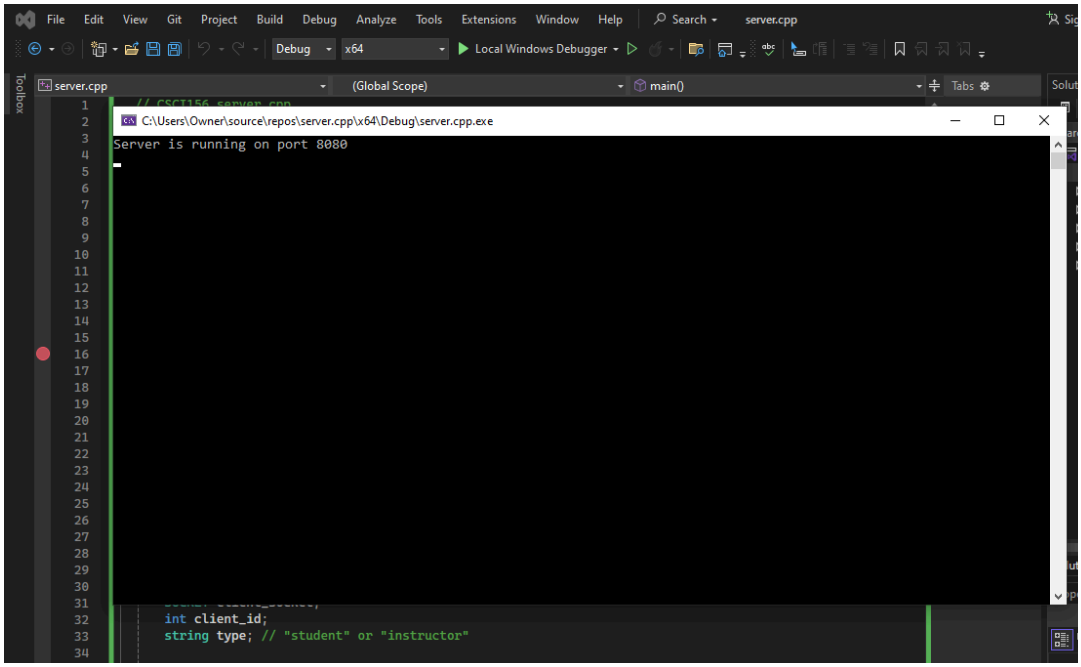
4. Analysis

4.1 Output Results

Here is the demo of the program, the scenario involves 1 instructor and 3 students. The instructor runs a class session with three students. Student 2 needs a private breakout session, Student 3 asks a direct question, and Student 4 participates in the main chat.

STEP 1: The instructor runs the CSCI156_server.cpp program to start the server on a Windows machine.

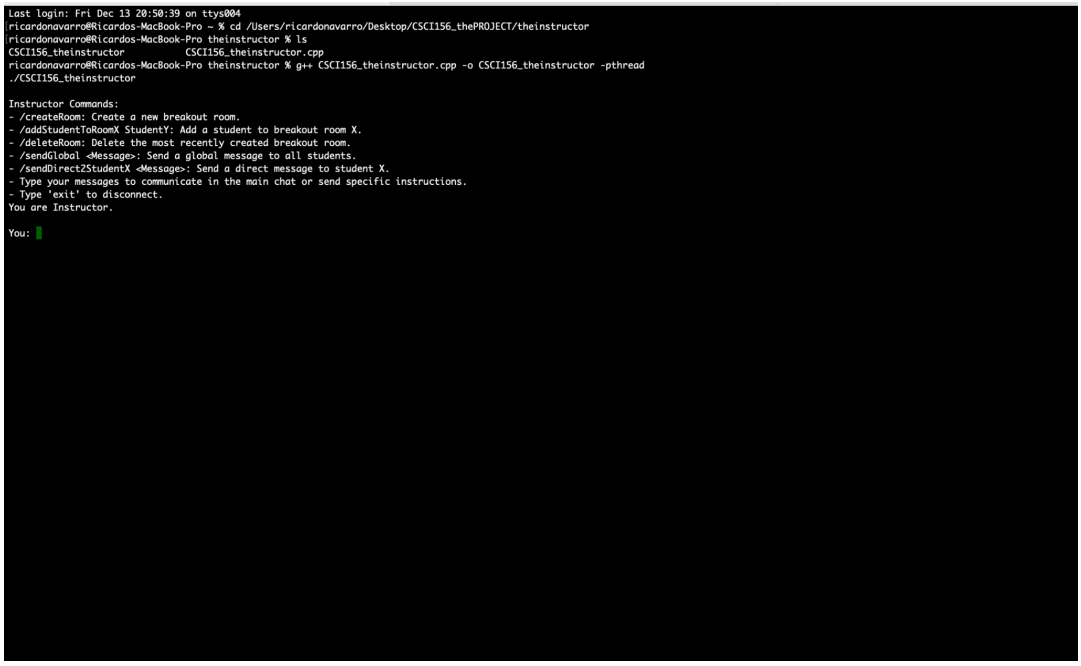
- The server begins listening for incoming connections on port 8080.



STEP 2: The instructor connects to the server using the CSCI156_theinstructor.cpp program.

- The server recognizes the instructor and displays:
  "You are Instructor."
- If another instructor tries to connect, they see:
  "An instructor is already connected."

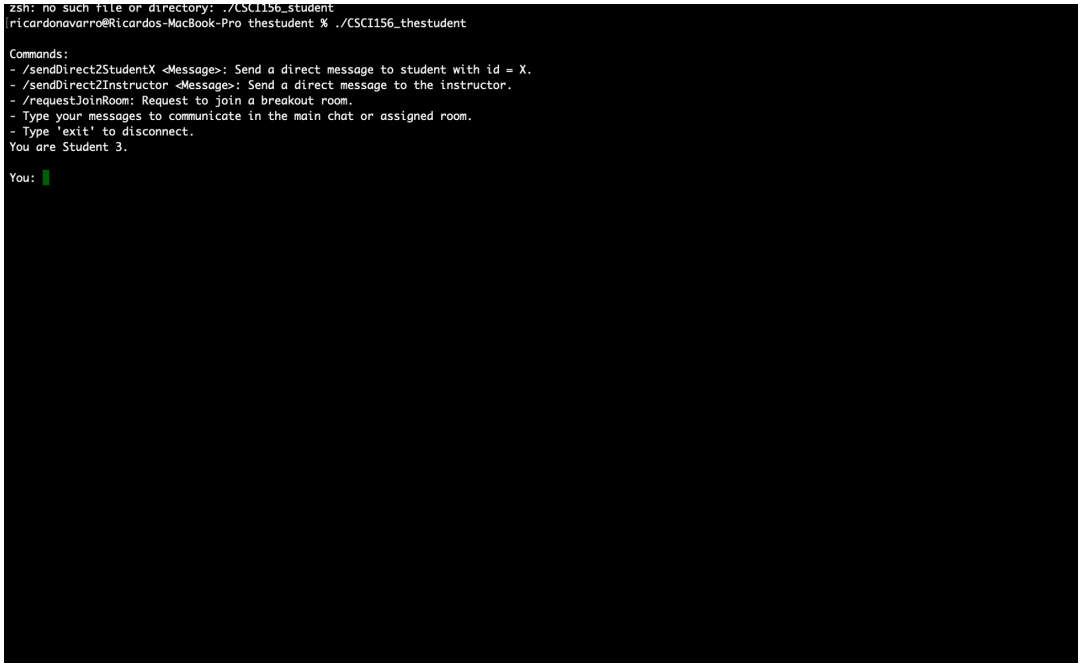
Instructor screen



STEP 3: Three students connect to the server using the CSCI156_thestudent.cpp program.

- Each student receives a message:
  "You are Student X."

Student 3 screen

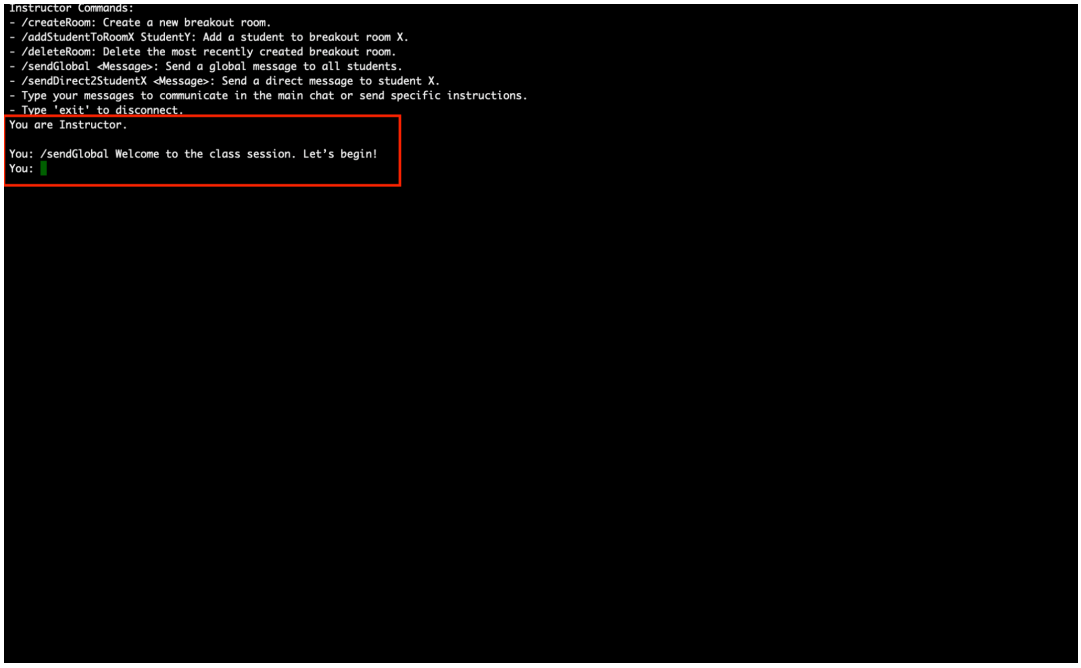


STEP 4: THE INSTRUCTOR SENDS A GLOBAL MESSAGE TO START THE SESSION:

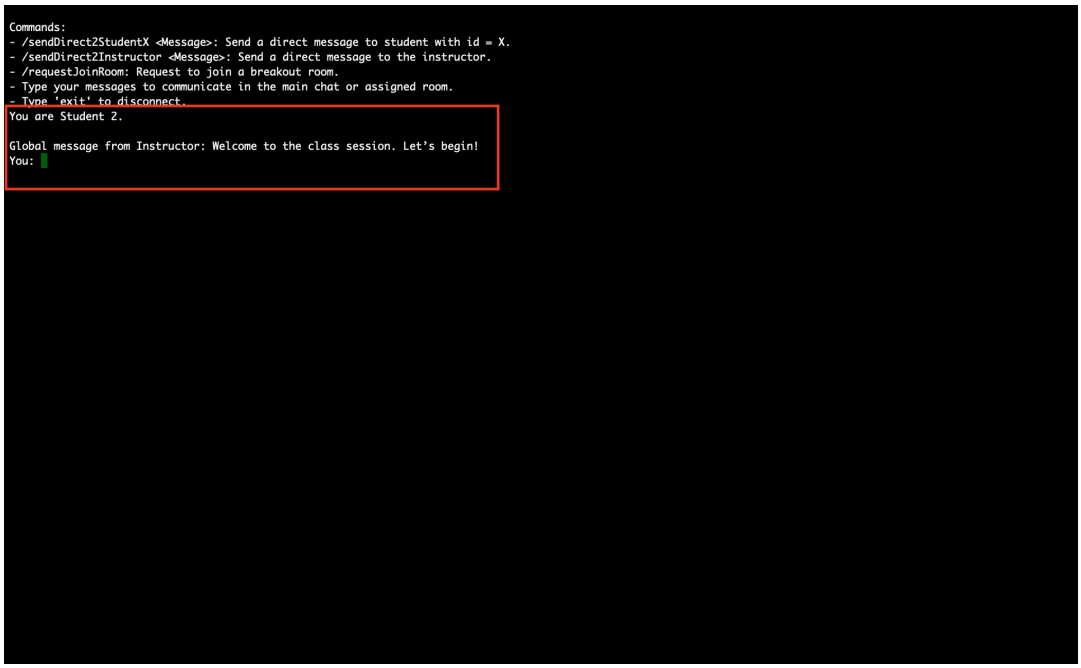
The instructor sends a global message to start the session:

- Command: /sendGlobal Welcome to the class session. Let’s begin!
- All students receive the message:
  "Global message from Instructor: Welcome to the class session. Let’s begin!"

Instructor screen



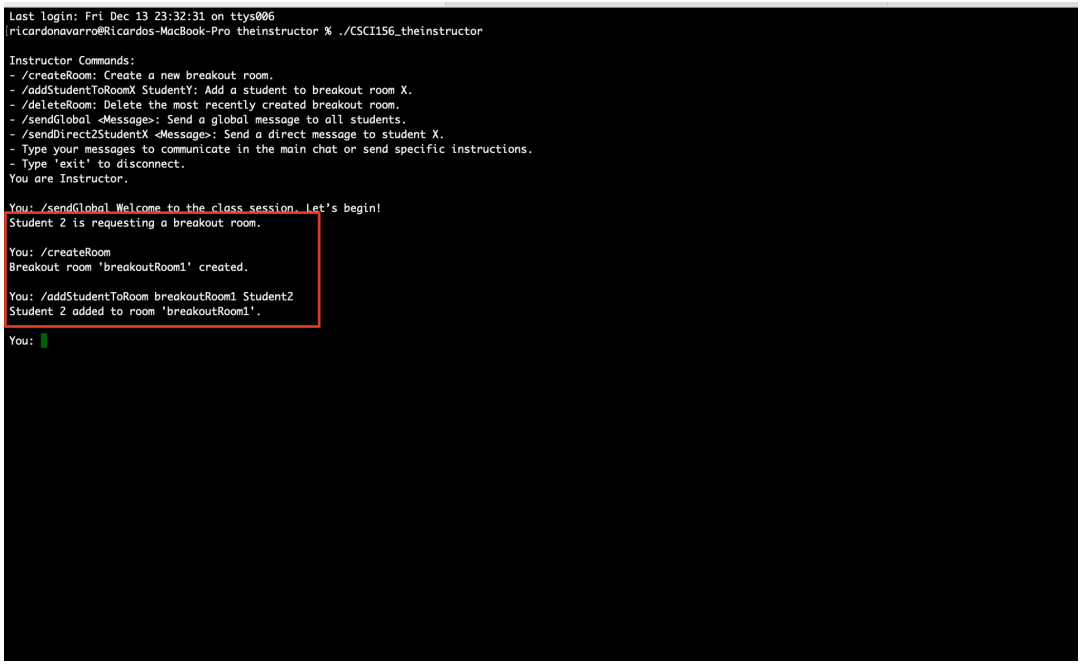
Student 2 screen



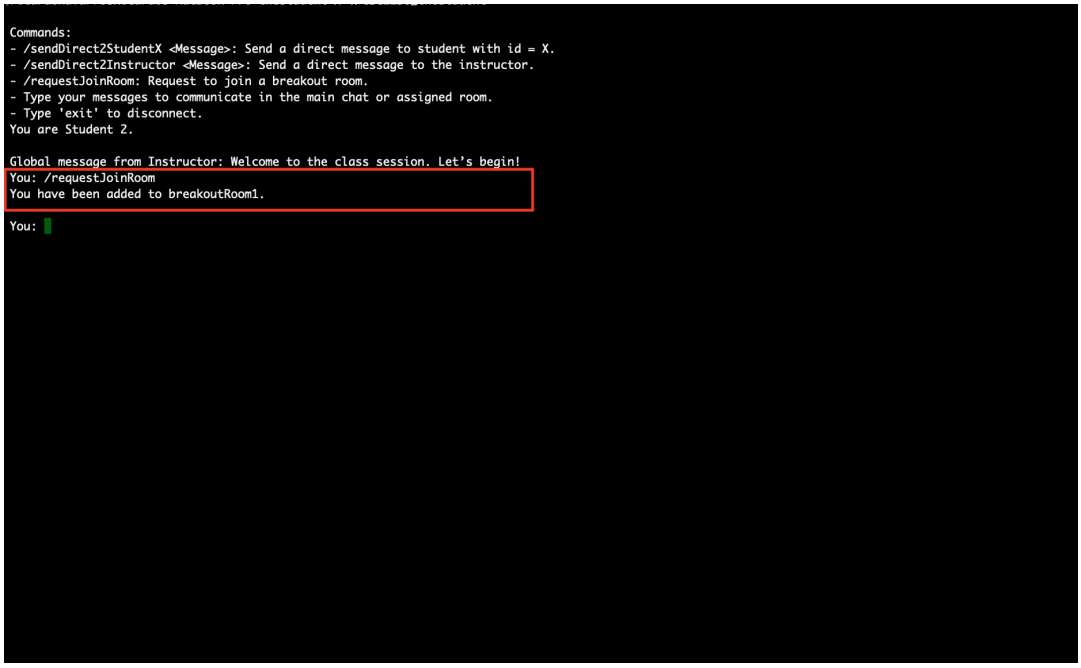
STEP 5: Student 2 requests a breakout room for a private discussion:

- Command: /requestJoinRoom
- The instructor sees the request and creates a breakout room:
- Command: /createRoom
- The server confirms: "Breakout room 'breakoutRoom1' created."
- The instructor adds Student 1 to the room:
- Command: /addStudentToRoom1 Student1
- Student 2 receives: "You have been added to breakoutRoom1."

Instructor screen



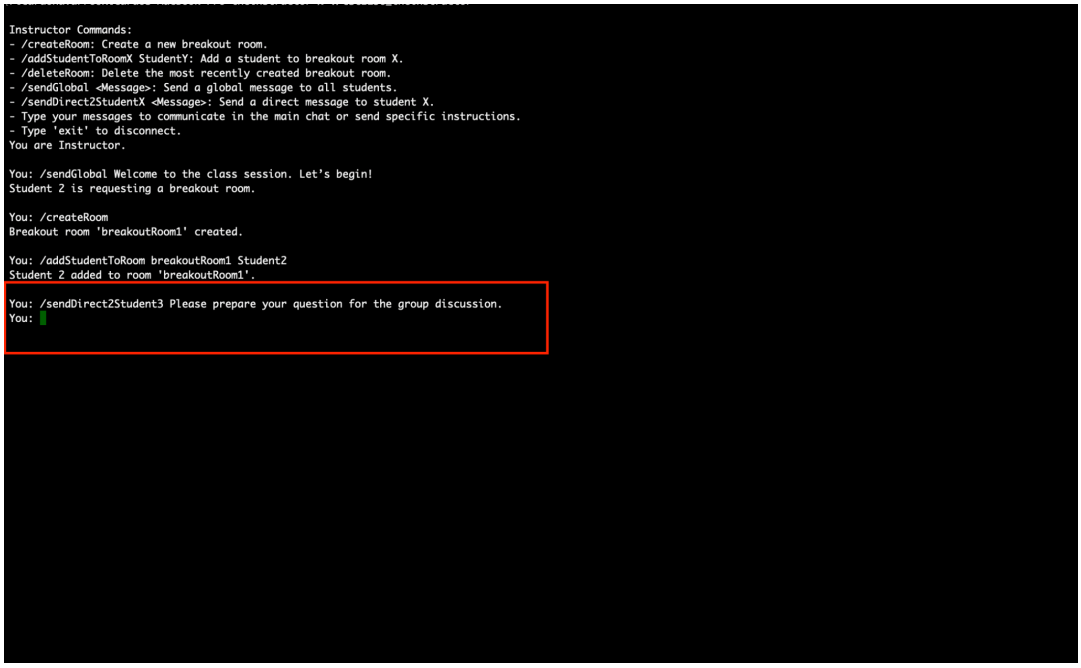
Student 2 screen



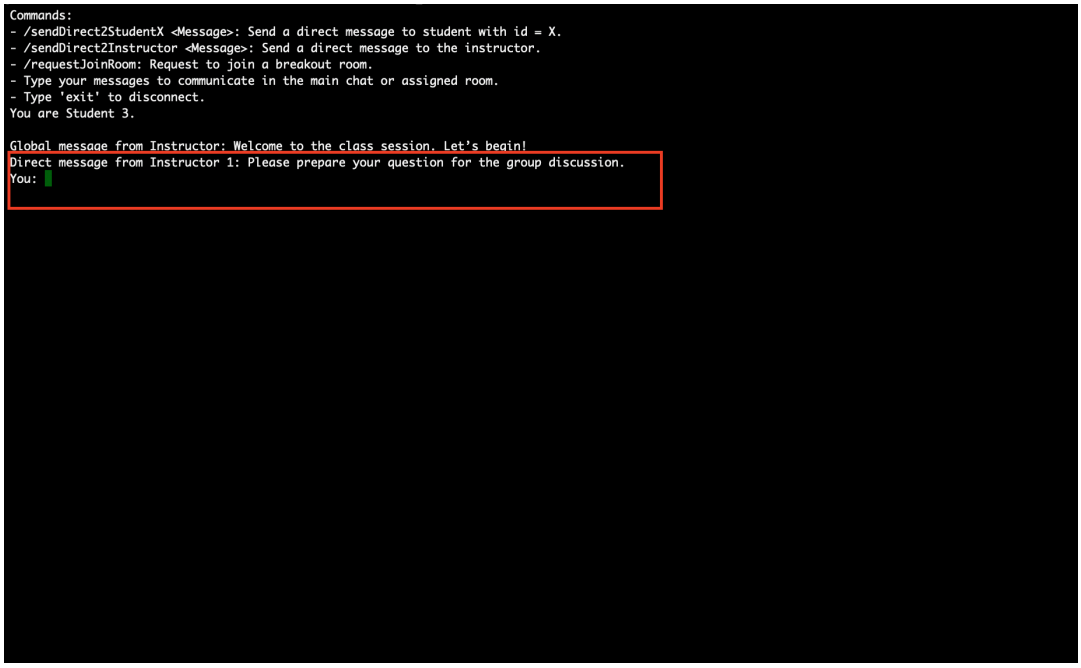
STEP 6: The instructor sends a direct message to Student 3:

- Command: /sendDirect2Student3 Please prepare your question for the group discussion.
- Student 3 receives the private message.

Instructor screen



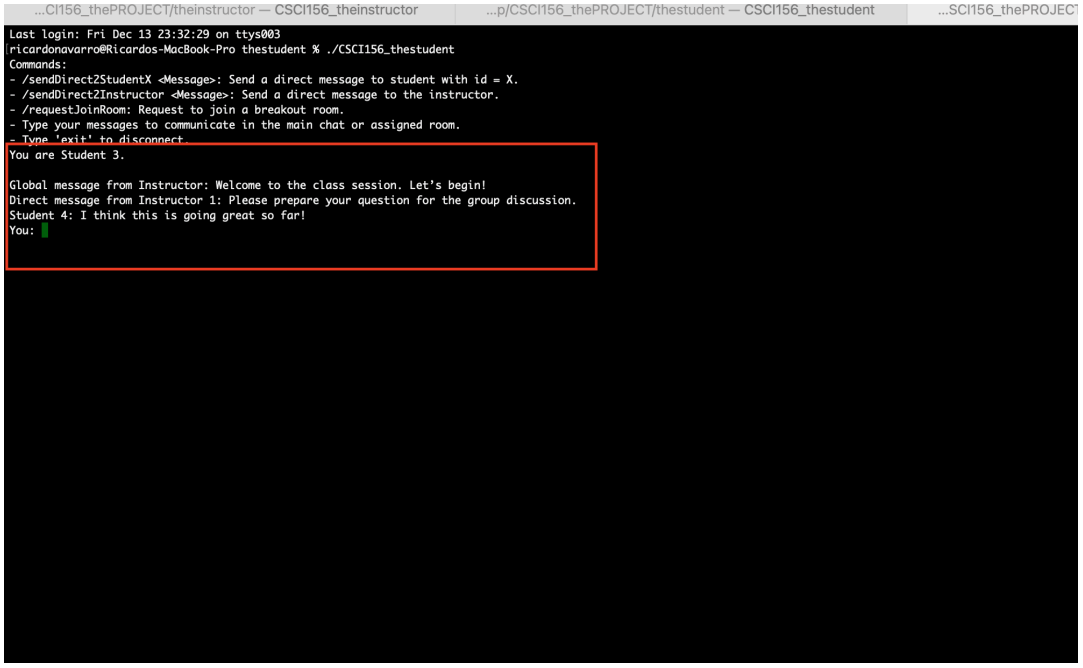
Student 3 screen



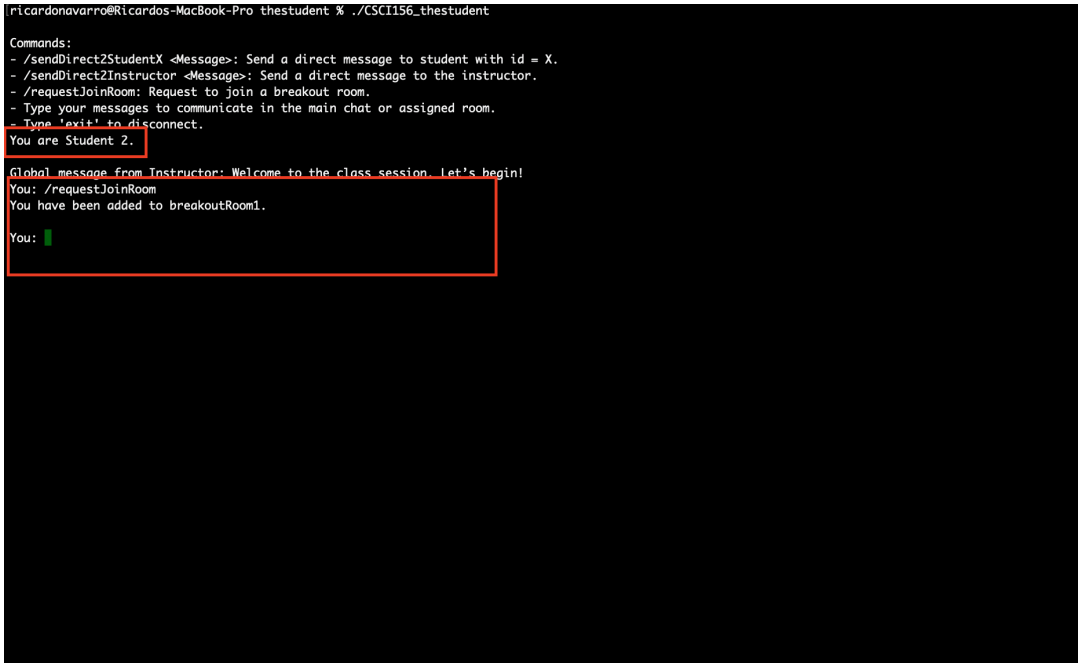
STEP 7:
Student 4 sends a message to the main chat:

- "I think this is going great so far!"
- All participants in the main chat, including the instructor, receive the message.(not student 2 who is in a breakout room)

Student 3 screen



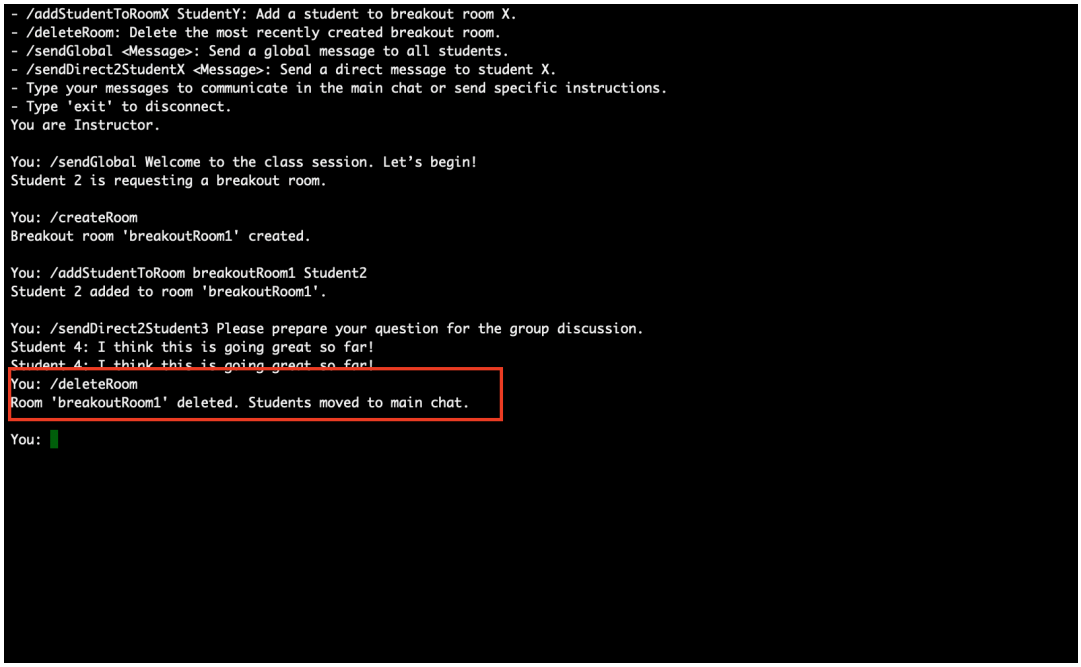
Student 2 screen



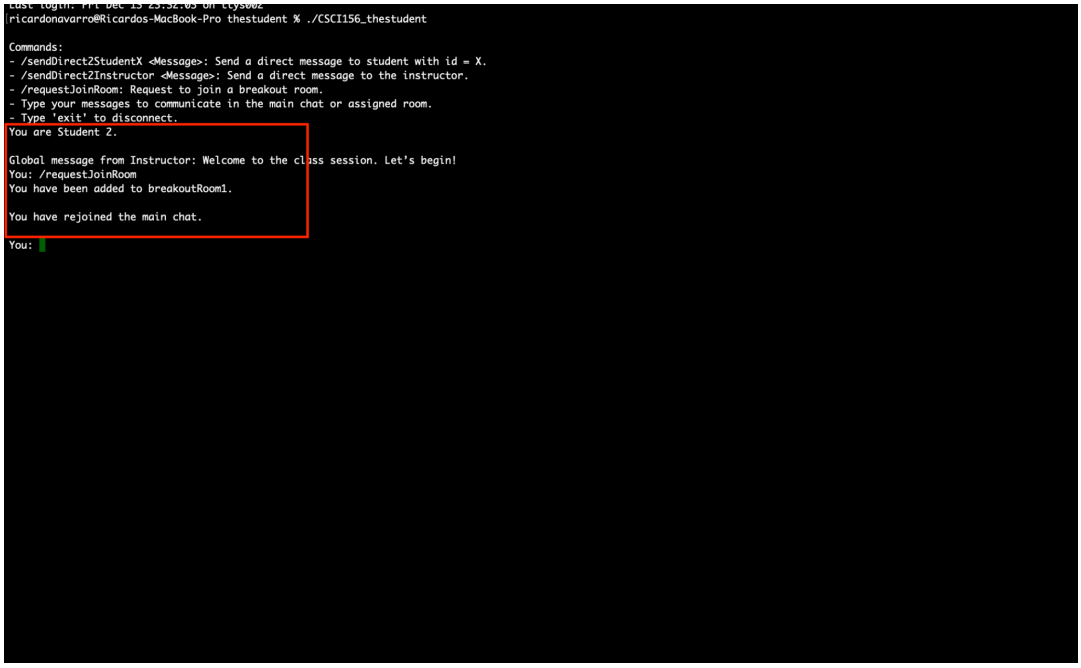
STEP 8: The instructor deletes the breakout room:

- Command: /deleteRoom
- Student 2 is moved back to the main chat and receives:
  "You have rejoined the main chat."

Instructor screen



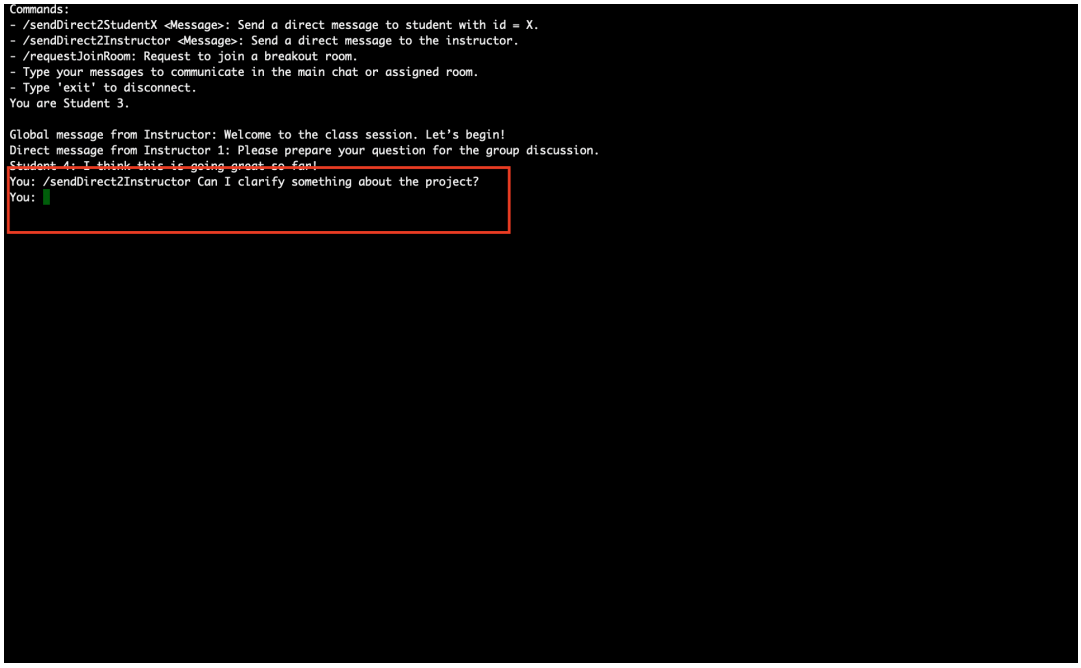
Student 2 screen



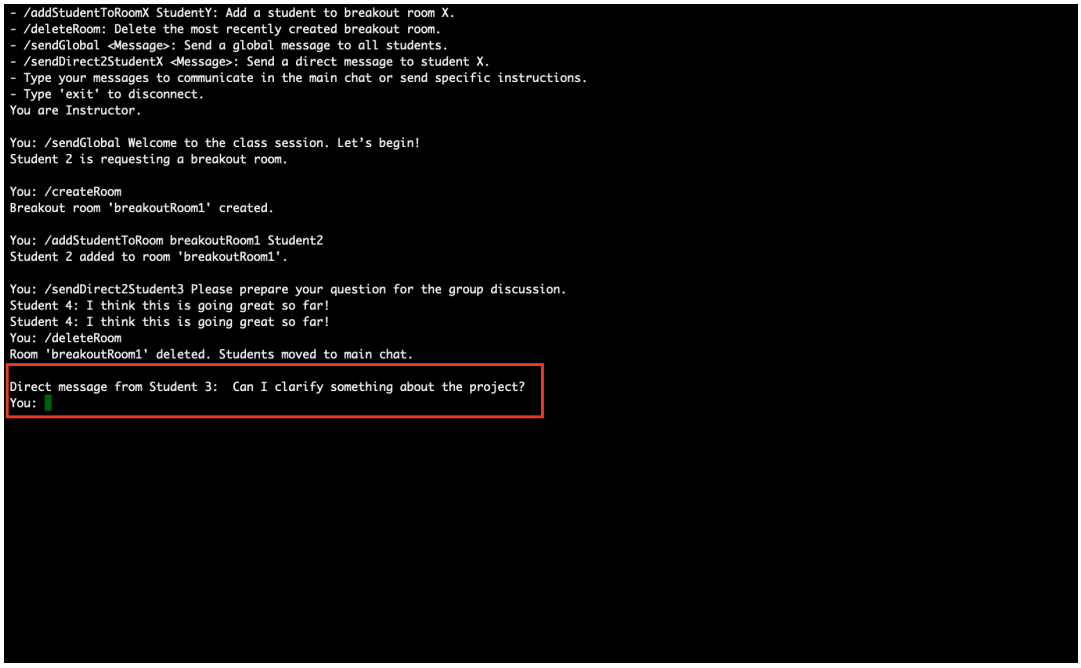
STEP 9: Student 3 sends a direct message to the instructor:

- Command: /sendDirect2Instructor Can I clarify something about the project?
- The instructor receives the private message.

Student 3 screen

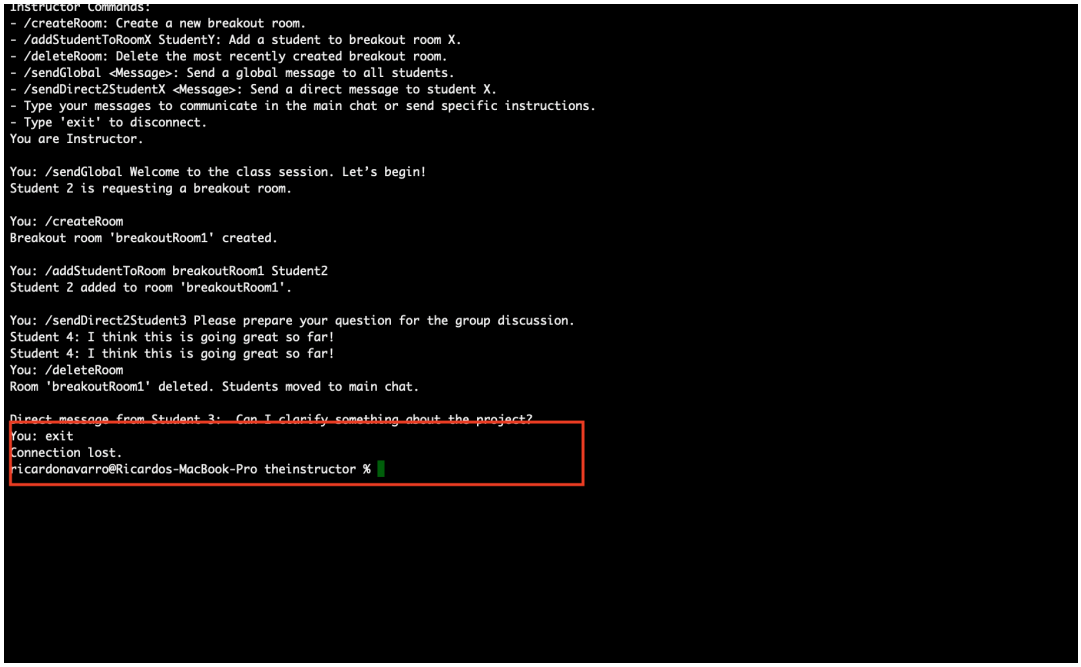


Instructor screen

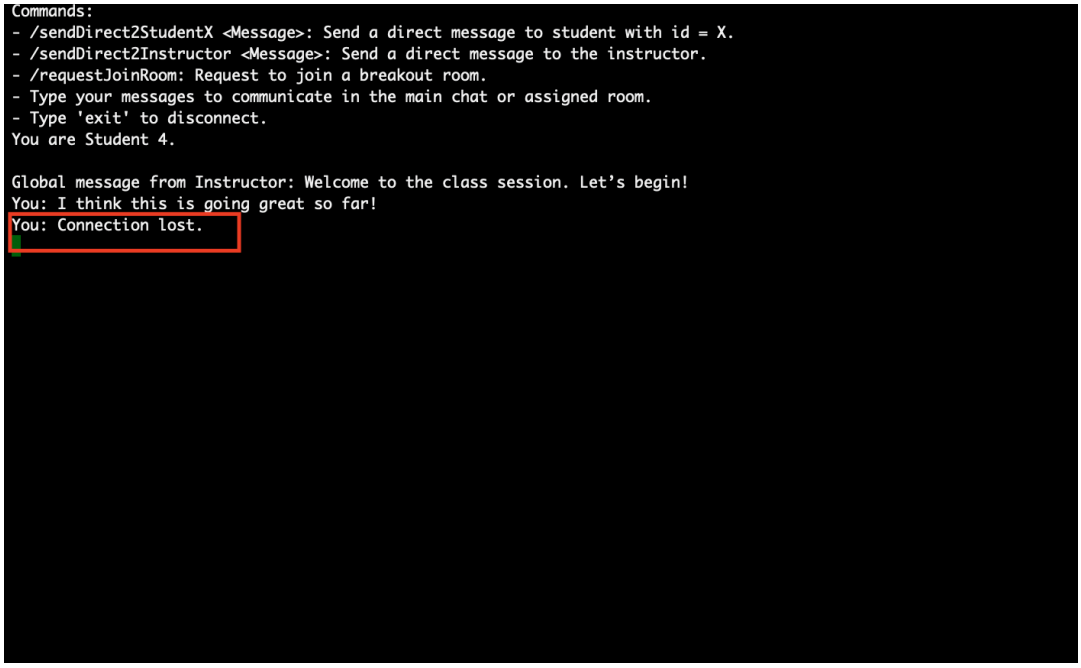


STEP 10: The instructor disconnects. The server continues until it is shutdown by whoever is controlling the server program. “Connection lost.” is displayed on every student’s screen.

Instructor screen



Student 4 screen



4.2 Output Analysis

PROGRAM STRENGTHS

The server handled connections from the instructor and all three students without crashes or errors.

- Each client successfully authenticated using the predefined password.
- All commands worked as expected:
    - /createRoom correctly created breakout rooms.
    - /addStudentToRoomX moved students to breakout rooms successfully.
    - /deleteRoom returned students to the main chat when rooms were deleted.
    - /sendGlobal and /sendDirect2StudentX sent messages as expected.
- Students could send direct messages to the instructor or the main chat using /sendDirect2Instructor or plain messages.
- The server handled concurrent communication between multiple clients using threads, ensuring that all clients could interact in real-time.

LIMITATIONS AND BUGS

- Duplicate Messages:
    - Messages sent by students were received twice by the instructor due to a bug in the server logic. This issue impacts readability but does not crash the program.
- No Join/Leave Notifications:
    - The program does not notify clients when someone joins or leaves the chat.

4.3 Client Profile

Server (CSCI156_server.cpp)

The server is the main part of the program. It manages all connections and commands. It:

- Manages Connections:
    - Lets one instructor and multiple students connect.
    - Uses a password to make sure only valid clients join.
- Handles Commands:
    - Runs instructor commands to manage breakout rooms and send messages.
    - Processes student requests for private messages and breakout rooms.
- Runs Threads:
    - Creates a new thread for every client so they can all work at the same time.
- Broadcasts Messages:
    - Sends global messages to everyone and private messages to specific people.
- Breakout Rooms:
    - Tracks breakout rooms and manages who is in them

Instructor Client (CSCI156_instructor.cpp)

The instructor client lets the instructor manage the class and interact with students. It can:

- Commands:
    - /createRoom: Make a breakout room for private chats.
    - /addStudentToRoomX StudentY: Move a student to a breakout room.
    - /deleteRoom: Close a breakout room and move the students back to the main chat.

- /sendGlobal <Message>: Send a message to all students.
- /sendDirect2StudentX <Message>: Send a private message to one student.
- exit: Leave the session and disconnect from the server.
- The instructor connects to the server using a built-in password in the code.

Student Client (CSCI156_student.cpp)

The student client allows students to join the session and interact. They can:

- Commands:
  - /sendDirect2StudentX <Message>: Send a private message to another student.
  - /sendDirect2Instructor <Message>: Send a private message to the instructor.
  - /requestJoinRoom: Ask to join a breakout room.
  - exit: Leave the session and disconnect from the server.

5. References

- **https://www.geeksforgeeks.org/socket-programming-cc/**
- **https://www.linuxhowtos.org/C_C++/socket.htm**
- **https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/sockets/socket-services**
- **https://www.youtube.com/watch?v=gntyAFoZp-E**

6. Appendix

- **CSCI156_server.cpp**

```
#include "pch.h"
#include <iostream>
#include <string>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <thread>
#include <vector>
```

```
#include <mutex>
#include <map>
#include <algorithm>
#include <sstream>
#pragma comment(lib, "ws2_32.lib")
#define PORT 8080   // port 8080 is commonly used for custom applications and avoids conflicts with reserved system ports
#define PASSWORD "@mypassword@"
using namespace std;



// CLASS TO STORE CLIENT INFORMATION
class Client {
public:
    SOCKET client_socket;
    int client_id;
    string type; // "student" or "instructor"


    Client(SOCKET socket, int id, const string& client_type)
        : client_socket(socket), client_id(id), type(client_type) {}
};


vector<Client> all_clients;             // all connected clients
vector<Client> main_chat;               // clients in the main chat
map<string, vector<Client>> breakout_rooms; // breakout room mapping (RoomName -> List of clients)
Client* instructor_client = nullptr;    // pointer to the instructor client
mutex clients_mutex;


int client_counter = 0;   // tracks all clients
int room_counter = 0;     // unique ID generator for breakout rooms


// HELPER FUNCTION TO FIND A CLIENT BY ID
Client* find_client_by_id(int client_id) {
    for (auto& client : all_clients) {
        if (client.client_id == client_id) {
            return &client;     // return pointer to client if id matches
        }
    }
    return nullptr;     // return null if no match found
```

```
}


// BROADCAST TO ALL CLIENTS IN THE MAIN CHAT(EXCLUDING THE SENDER)
void broadcast_to_main_chat(const string& message, SOCKET sender_socket) {
    clients_mutex.lock();   // lock to ensure safe access to client lists
    for (auto& client : main_chat) {
        if (client.client_socket != sender_socket) {    // dont send to message to sender
            send(client.client_socket, message.c_str(), (int)message.size(), 0);
        }
    }
    if (instructor_client && instructor_client->client_socket != sender_socket) {
        send(instructor_client->client_socket, message.c_str(), (int)message.size(), 0);    // send to instructor
    }
    clients_mutex.unlock(); // unlock after sending messages
}


// BROADCAST TO A SPECIFIC BREAKOUT ROOM
void broadcast_to_room(const string& room_name, const string& message, SOCKET sender_socket) {
    clients_mutex.lock();   // lock for thread safety
    if (breakout_rooms.count(room_name)) {  // check if room exists
        for (auto& client : breakout_rooms[room_name]) {
            if (client.client_socket != sender_socket) {    // don't send to sender
                send(client.client_socket, message.c_str(), (int)message.size(), 0);
            }
        }
        // notify the instructor with room details
        if (instructor_client) {
            string instructor_message = "Message from " + room_name + ": " + message;
            send(instructor_client->client_socket, instructor_message.c_str(), (int)instructor_message.size(),
0);
        }
    }
    clients_mutex.unlock(); // unlock after broadcasting
}


// BROADCAST TO ALL CLIENTS (GLOBAL BROADCAST)
void broadcast_to_all_clients(const string& message) {
    clients_mutex.lock();   // lock for thread safety
    for (auto& client : all_clients) {
        send(client.client_socket, message.c_str(), (int)message.size(), 0);    // send message to all clients
    }
```

```
    clients_mutex.unlock(); // unlock after sending
}


// NOTIFY A STUDENT ABOUT BEING ADDED OR REMOVED FROM A BREAKOUT ROOM
void notify_student(Client& student, const string& message) {
    send(student.client_socket, message.c_str(), (int)message.size(), 0);   // send notification message
}



// HANDLE DIRECT MESSAGES
void handle_direct_message(const string& command, SOCKET sender_socket, int sender_id, const
string& sender_type) {
    if (command.rfind("/sendDirect2Student", 0) == 0) {     // handle direct message to student
        size_t id_start = command.find("Student") + 7;      // locate student id in command
        size_t space_pos = command.find(' ', id_start);     // locate space after id
        if (space_pos == string::npos) {
            string error_message = "Invalid command format. Use /sendDirect2StudentX <Message>.\n";
            send(sender_socket, error_message.c_str(), (int)error_message.size(), 0);   // send error message
            return;
        }
        int student_id = stoi(command.substr(id_start, space_pos - id_start));  // extract student id
        string message = command.substr(space_pos + 1); // extract message

        clients_mutex.lock();   // lock for thread safety
        Client* recipient = find_client_by_id(student_id);  // find recipient student
        if (recipient != nullptr && recipient->type == "student") {
            string direct_message = "Direct message from " + sender_type + " " + to_string(sender_id) + ": " +
message;
            send(recipient->client_socket, direct_message.c_str(), (int)direct_message.size(), 0);  // send
message
        }
        else {
            string error_message = "Student " + to_string(student_id) + " not found.\n";
            send(sender_socket, error_message.c_str(), (int)error_message.size(), 0);   // send error message
        }
        clients_mutex.unlock(); // unlock after sending
    }
    else if (command.rfind("/sendDirect2Instructor", 0) == 0) { // handle direct message to instructor
        string message = command.substr(22);    // extract message

        clients_mutex.lock();   // lock for thread safety
        if (instructor_client != nullptr) {
```

```
            string direct_message = "Direct message from Student " + to_string(sender_id) + ": " + message;
            send(instructor_client->client_socket, direct_message.c_str(), (int)direct_message.size(), 0);  //
send message
        }
        else {
            string error_message = "Instructor not found.\n";
            send(sender_socket, error_message.c_str(), (int)error_message.size(), 0);   // send error message
        }
        clients_mutex.unlock(); // unlock after sending
    }
    else {
        string error_message = "Invalid direct message format.\n";
        send(sender_socket, error_message.c_str(), (int)error_message.size(), 0);   // send error message
    }
}



// HANDLE INSTRUCTOR COMMANDS
void handle_instructor_command(const string& command, SOCKET sender_socket) {
    stringstream ss(command);
    string cmd;
    ss >> cmd;


    if (cmd == "/createRoom") {     // create a new breakout room
        clients_mutex.lock();       // lock for thread safety
        room_counter++;             // increment room counter
        string room_name = "breakoutRoom" + to_string(room_counter);    // generate room name
        breakout_rooms[room_name] = vector<Client>();                   // create empty room
        clients_mutex.unlock();                                         // unlock after creating room


        string success_message = "Breakout room '" + room_name + "' created.\n";
        send(sender_socket, success_message.c_str(), (int)success_message.size(), 0);   // notify instructor
    }
    else if (cmd == "/deleteRoom") {    // delete the most recently created room***
        clients_mutex.lock();           // lock for thread safety
        if (!breakout_rooms.empty()) {
            auto it = --breakout_rooms.end();   // get the last room
            string room_name = it->first;


            for (auto& student : it->second) {
```

```
            main_chat.push_back(student);   // move students back to main chat
            notify_student(student, "You have rejoined the main chat.\n");  // notify students
        }
        breakout_rooms.erase(room_name);    // delete room
        room_counter--; // Adjust room counter


        string success_message = "Room '" + room_name + "' deleted. Students moved to main chat.\n";
        send(sender_socket, success_message.c_str(), (int)success_message.size(), 0);   // notify instructor
    }
    else {
        string error_message = "No breakout rooms to delete.\n";
        send(sender_socket, error_message.c_str(), (int)error_message.size(), 0);   // send error message
    }
    clients_mutex.unlock();     // unlock after deleting
  }
  else if (cmd.rfind("/addStudentToRoom", 0) == 0) {  // add a student to a room***
    string room_name;
    string student_str;
    ss >> room_name >> student_str; // extract the room name and student identifier


    if (room_name.rfind("breakoutRoom", 0) == 0 && student_str.rfind("Student", 0) == 0) {  // check
for valid room and student identifiers
        int student_id = stoi(student_str.substr(7));       // extract student ID from the identifier
        clients_mutex.lock();                               // lock for thread safety
        Client* student = find_client_by_id(student_id);    // find the student client by ID


        if (student && breakout_rooms.count(room_name)) {   // ensure the student exists and the room is
valid
            breakout_rooms[room_name].push_back(*student);  // add the student to the breakout room
            main_chat.erase(remove_if(main_chat.begin(), main_chat.end(),
                [student](const Client& c) { return c.client_id == student->client_id; }), main_chat.end());
// remove student from main chat
            notify_student(*student, "You have been added to " + room_name + ".\n");    // notify the
student


            string success_message = "Student " + to_string(student_id) + " added to room '" + room_name
+ "'.\n";
            send(sender_socket, success_message.c_str(), (int)success_message.size(), 0);   // notify the
instructor of success
        }
```

```
            else {
                string error_message = (student ? "Room '" + room_name + "' does not exist.\n" :
                    "Student " + to_string(student_id) + " not found.\n");
                send(sender_socket, error_message.c_str(), (int)error_message.size(), 0);   // notify the
instructor of the error
            }
            clients_mutex.unlock();     // unlock after processing
        }
        else {
            string error_message = "Invalid command format. Use /addStudentToRoomX StudentY.\n";
            send(sender_socket, error_message.c_str(), (int)error_message.size(), 0);   // notify the instructor
of the invalid format
        }
    }
    else if (cmd == "/sendGlobal") {    // handle global messages from instructor***
        string global_message;
        getline(ss, global_message);                                // read the global message from the command
input
        if (!global_message.empty() && global_message[0] == ' ') {
            global_message = global_message.substr(1);                // remove leading space if present
        }
        global_message = "Global message from Instructor: " + global_message;   // format the global
message
        broadcast_to_all_clients(global_message);       // send the message to all clients
    }
    else if (cmd.rfind("/sendDirect2Student", 0) == 0) {    // handle direct messages to a specific student***
        try {
            handle_direct_message(command, sender_socket, instructor_client->client_id, "Instructor");  //
process the direct message
        }
        catch (const exception& e) {
            string error_message = "Error processing direct message: " + string(e.what()) + "\n";
            send(sender_socket, error_message.c_str(), (int)error_message.size(), 0);   // notify instructor of
error
        }
    }
    else {  // handle invalid commands
        string error_message = "Invalid command.\n";
        send(sender_socket, error_message.c_str(), (int)error_message.size(), 0);   // notify instructor of
invalid command
    }
}
```

```
// HANDLE CLIENT MESSAGES AND COMMANDS
void handle_client(SOCKET client_socket, int client_id) {   // process client connections and
communication
   char buffer[1024] = { 0 };


   // Require password
   int valread = recv(client_socket, buffer, sizeof(buffer) - 1, 0);   // read password from client
   buffer[valread] = '\0';     // null-terminate the password string
   if (strcmp(buffer, PASSWORD) != 0) {
      string error_message = "Incorrect password. Disconnecting.\n";  // check if password matches
      send(client_socket, error_message.c_str(), (int)error_message.size(), 0);   // send error message
      closesocket(client_socket);     // close the connection
      return;
   }


   // Receive client type
   memset(buffer, 0, sizeof(buffer));
   valread = recv(client_socket, buffer, sizeof(buffer) - 1, 0);   // read client type (instructor or student)
   string client_type(buffer);


   // Add client to appropriate list
   clients_mutex.lock();              // lock client list for thread safety
   if (client_type == "instructor") {
      if (instructor_client != nullptr) {     // ensure only one instructor is connected
         string error_message = "An instructor is already connected.\n";
         send(client_socket, error_message.c_str(), (int)error_message.size(), 0);   // notify client
         closesocket(client_socket); // close the connection
         clients_mutex.unlock();
         return;
      }
      instructor_client = new Client(client_socket, client_id, "instructor"); // add instructor to the client list
      main_chat.push_back(*instructor_client); // add instructor to the main chat
      send(client_socket, "You are Instructor.\n", 21, 0);    // send confirmation to the instructor
   }
   else if (client_type == "student") {                        // handle student connection
      Client new_student(client_socket, client_id, "student");    // create a new student client
      all_clients.push_back(new_student);                      // add student to all_clients list
      main_chat.push_back(new_student);                        // add student to the main chat
      string welcome_message = "You are Student " + to_string(client_id) + ".\n";
```

```
        send(client_socket, welcome_message.c_str(), (int)welcome_message.size(), 0);   // send
confirmation to student
    }
    clients_mutex.unlock(); // unlock the client list


    // COMMUNICATION LOOP***
    while (true) {
        memset(buffer, 0, sizeof(buffer));      // clear the buffer for new input
        valread = recv(client_socket, buffer, sizeof(buffer) - 1, 0);   // read messages from client


        if (valread <= 0 || strcmp(buffer, "exit") == 0) {  // handle client disconnection
            clients_mutex.lock();   // lock for safe removal
            all_clients.erase(remove_if(all_clients.begin(), all_clients.end(),
                [client_socket](const Client& c) { return c.client_socket == client_socket; }), all_clients.end());
// remove client from all_clients
            main_chat.erase(remove_if(main_chat.begin(), main_chat.end(),
                [client_socket](const Client& c) { return c.client_socket == client_socket; }), main_chat.end());
// remove client from main chat
            if (instructor_client && instructor_client->client_socket == client_socket) {   // handle instructor
disconnection
                delete instructor_client;
                instructor_client = nullptr;
            }
            clients_mutex.unlock();     // unlock after removal
            closesocket(client_socket); // close the socket
            break;                      // exit the communication loop
        }


        string message(buffer);             // convert message to string
        if (client_type == "instructor") {
            if (message.rfind("/", 0) == 0) {   // check if message is a command
                handle_instructor_command(message, client_socket);  // process instructor commands
            }
            else {
                broadcast_to_main_chat("Instructor: " + message, client_socket);    // broadcast instructor
message
            }
        }
        else if (client_type == "student") {
            if (message == "/requestJoinRoom") {    // handle room join request
```

```
            string request_message = "Student " + to_string(client_id) + " is requesting a breakout
room.\n";
            if (instructor_client) {
                send(instructor_client->client_socket, request_message.c_str(), (int)request_message.size(),
0);    // notify instructor
            }
        }
        else if (message.rfind("/sendDirect2", 0) == 0) {   // handle direct messages
            handle_direct_message(message, client_socket, client_id, "Student");
        }
        else {
            string room_name;
            clients_mutex.lock();               // lock breakout rooms for access
            for (auto& pair : breakout_rooms) { // iterate through breakout rooms
                const string& key = pair.first; // room name
                const vector<Client>& value = pair.second;  // clients in the room
                if (find_if(value.begin(), value.end(), [client_id](const Client& c) { return c.client_id ==
client_id; }) != value.end()) {
                    room_name = key;    // find the room the student is in
                    break;
                }
            }
            clients_mutex.unlock();     // unlock breakout rooms
            if (!room_name.empty()) {
                broadcast_to_room(room_name, "Student " + to_string(client_id) + ": " + message,
client_socket);    // broadcast within room
            }
            else {
                broadcast_to_main_chat("Student " + to_string(client_id) + ": " + message, client_socket);
// broadcast to main chat
            }
        }
    }
  }
}


// MAIN FUNCTION TO START THE SERVER
int main() {
  WSADATA wsaData;
  WSAStartup(MAKEWORD(2, 2), &wsaData);   // initialize Winstock


  SOCKET server_socket = socket(AF_INET, SOCK_STREAM, 0); // create server socket
```

```
    sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;          // IPv4
    server_addr.sin_addr.s_addr = INADDR_ANY;   // accecpt connections from any address
    server_addr.sin_port = htons(PORT);         // set the port


    bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr));   // bind the socket
    listen(server_socket, SOMAXCONN);   // start listeneing for connections


    cout << "Server is running on port " << PORT << endl;       // notify that the server is running


    while (true) {      // continuously accept clients
        SOCKET client_socket = accept(server_socket, NULL, NULL);       // accept a client connection
        client_counter++;                                       // increment the client counter
        thread(handle_client, client_socket, client_counter).detach();  // handle the client in a separate thread
    }
    WSACleanup();   // clean up Winstock resources
    return 0;
}
```

- **CSCI156_instructor.cpp**

```
#include <iostream>
#include <string>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <thread>
#pragma comment(lib, "ws2_32.lib")
#define PORT 8080               // port number used to connect to the server
#define PASSWORD "@mypassword@" // password used to authenticate the instructor client
using namespace std;

SOCKET client_fd;       // socket descriptor for the client
bool running = true;    // flag to keep the main loop running


// RECEIVE MESSAGES FROM THE SERVER
void receive_messages() {
    char buffer[1024] = { 0 };  // buffer to store received messages
    while (running) {
        int valread = recv(client_fd, buffer, sizeof(buffer) - 1, 0);   // read messages from the server
```

```
        if (valread > 0) {
            buffer[valread] = '\0';         // null-terminate the recieved string
            cout << "\r" << buffer << endl; // display the message
            cout << "You: ";
            fflush(stdout); // ensure the prompt is shown after the message
        }
        else {
            cerr << "Connection lost. Error code: " << WSAGetLastError() << endl;   // handle connection
loss
            running = false;    // stop the loop
        }
    }
}

int main() {
    WSADATA wsaData;

    // INTITIALIZE WINSTOCK
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        cerr << "WSAStartup failed\n";
        return -1;
    }

    // CREATE SOCKET
    if ((client_fd = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET) {
        cerr << "Socket creation error\n";
        WSACleanup();
        return -1;
    }

    // SETUP SERVER ADDRESS
    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;   // IPv4
    server_addr.sin_port = htons(PORT); // convert port number to network byte order
    const char* server_ip_address = "195.155.45.26"; // server IP

    if (inet_pton(AF_INET, server_ip_address, &server_addr.sin_addr) <= 0) {
        cerr << "Invalid address / Address not supported\n";
        closesocket(client_fd);
        WSACleanup();
        return -1;
    }

    // CONNECT TO SERVER
```

```
    if (connect(client_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        cerr << "Connection failed\n";
        closesocket(client_fd);
        WSACleanup();
        return -1;
    }

    // AUTHENTICATE WITH SERVER
    send(client_fd, PASSWORD, strlen(PASSWORD), 0); // send password
    send(client_fd, "instructor", 10, 0);           // identify as an instructor

   // DISPLAY INSTRUCTOR COMMANDS
    cout << "Instructor Commands:\n";
    cout << "- /createRoom: Create a new breakout room.\n";
    cout << "- /addStudentToRoomX StudentY: Add a student to breakout room X.\n";
    cout << "- /deleteRoom: Delete the most recently created breakout room.\n";
    cout << "- /sendGlobal <Message>: Send a global message to all students.\n";
    cout << "- /sendDirect2StudentX <Message>: Send a direct message to student X.\n";
    cout << "- Type your messages to communicate in the main chat or send specific instructions.\n";
    cout << "- Type 'exit' to disconnect.\n";


    thread receiver(receive_messages);  // start thread handle incoming messages

    string message;
    while (running) {
        cout << "You: ";
        getline(cin, message);  // read input from the instructor

        if (message == "exit") {    // check if the instructor wants to exit
            running = false;
            break;
        }

        send(client_fd, message.c_str(), message.size(), 0);    // send the message to the server
    }

    receiver.join();        // wait for the message reciever thread to finish
    closesocket(client_fd); // close the socket
    WSACleanup();           // clean up the Winsock resources
    return 0;
}
```

- **CSCI156_student.cpp**

```
#include <iostream>
#include <string>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <thread>
#pragma comment(lib, "ws2_32.lib")
#define PORT 8080                  // port number used to connnect to the server
#define PASSWORD "@mypassword@"     // password used to authenticate the student client
using namespace std;

SOCKET client_fd;       // socket descriptor for the client
bool running = true;    // flag to keep the main loop running

// RECEIVE MESSAGES FROM THE SERVER
void receive_messages() {
   char buffer[1024] = { 0 };  // buffer to store received messages
   while (running) {
      int valread = recv(client_fd, buffer, sizeof(buffer) - 1, 0);   // read messages from the server
      if (valread > 0) {
         buffer[valread] = '\0'; // null-terminate the recieved string
         cout << "\r" << buffer << endl; // display the message
         cout << "You: ";
         fflush(stdout); // ensure prompt is shown after the message
      }
      else {
         cerr << "Connection lost. Error code: " << WSAGetLastError() << endl;   // handle connection
loss
         running = false;    // stop the loop
      }
   }
}

int main() {
   WSADATA wsaData;

   // INITIALIZE WINSOCK
   if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
      cerr << "WSAStartup failed\n";
      return -1;
   }

   // CREATE SOCKET
   if ((client_fd = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET) {
```

```
    cerr << "Socket creation error\n";
    WSACleanup();
    return -1;
}

// SETUP SERVER ADDRESS
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;   // IPv4
server_addr.sin_port = htons(PORT); // convert port number to network byte order
const char* server_ip_address = "195.155.45.26"; // Replace with actual server IP

if (inet_pton(AF_INET, server_ip_address, &server_addr.sin_addr) <= 0) {
    cerr << "Invalid address / Address not supported\n";
    closesocket(client_fd);
    WSACleanup();
    return -1;
}

// CONNECT TO SERVER
if (connect(client_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
    cerr << "Connection failed\n";
    closesocket(client_fd);
    WSACleanup();
    return -1;
}

// AUTHENTICATE WITH SERVER
send(client_fd, PASSWORD, strlen(PASSWORD), 0); // send password
send(client_fd, "student", 7, 0);               // identify as a student

// DISPLAY STUDENT COMMANDS
cout << "Commands:\n";
cout << "- /sendDirect2StudentX <Message>: Send a direct message to student with id = X.\n";
cout << "- /sendDirect2Instructor <Message>: Send a direct message to the instructor.\n";
cout << "- /requestJoinRoom: Request to join a breakout room.\n";
cout << "- Type your messages to communicate in the main chat or assigned room.\n";
cout << "- Type 'exit' to disconnect.\n";

thread receiver(receive_messages);  // start thread to handle incoming messages

string message;
while (running) {
    cout << "You: ";
    getline(cin, message);  // read input from the student
```

```
        if (message == "exit") {    // check if the student wants to exit
            running = false;
            break;
        }

        send(client_fd, message.c_str(), message.size(), 0);    // send the message
    }

    receiver.join();        // wait for the message receiver thread to finish
    closesocket(client_fd); // close the socket
    WSACleanup();           // clean up Winsock resources
    return 0;
}
```

7. AI HELP

We used ChatGPT to help with parsing the commands in the server code. Specifically, we referred to it for help in implementing the logic for the first instructor command, /createRoom. Once we understood how to handle and parse commands, we were able to do the same approach to implement the other instructor commands (e.g., /addStudentToRoomX, /deleteRoom, etc.) without additional help.

The exact code where ChatGPT helped is in the handle_instructor_command function. Below is the portion of the code where the /createRoom command is implemented:

```
if (cmd == "/createRoom") {     // create a new breakout room
    clients_mutex.lock();       // lock for thread safety
    room_counter++;             // increment room counter
    string room_name = "breakoutRoom" + to_string(room_counter);    // generate room name
    breakout_rooms[room_name] = vector<Client>();                   // create empty room
    clients_mutex.unlock();                                         // unlock after creating room

    string success_message = "Breakout room '" + room_name + "' created.\n";
    send(sender_socket, success_message.c_str(), (int)success_message.size(), 0);   // notify instructor
}
```

It should be on line 168 in server.cpp. 168-178.

This first implementation provided the structure for parsing and handling other commands, which we were able to modify and adapt ourselves.

8. Conclusions

The breakout room program uses TCP to ensure reliable and ordered delivery of messages, which is important for communication between the instructor and students. The server runs on Windows using Winsock, while the client can run on either Windows or macOS, making it versatile. The server uses threads to handle multiple clients at the same time, and a mutex keeps shared resources, like the client list and breakout rooms, safe.

This project successfully created a breakout room system with real-time communication. It allows the instructor to manage rooms and send messages, while students can interact through the main chat or private messages.