Application documentation

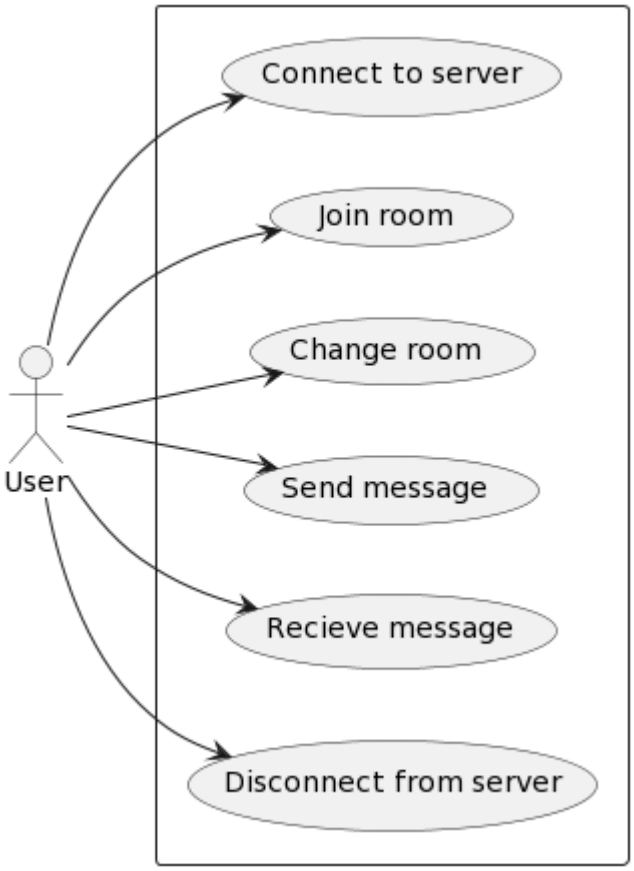
General system description:

Upon server start, application creates vector of structure objects Clients that contains such information: this client's socket and to which room this client is connected. After this server starts listening for incoming connections on port 8080. After accepting client it calls function handleClient() in separate thread. Inside of this function client proposed to choose the room and after this information about client's socket and it's room gets pushed to the vector. After this program enters an infinite loop where user can: disconnect, join another room or send a message. Each message gets added to the queue and then broadcasted with broadcastMessages(). To rejoin another room user should send REJOIN_roomId as his message where room id is number of desired room. On the client side, we have threads for receiving and sending messages.

Application protocol:

Client sends string with room id (1 byte) after which server assigns this client to the specific room.
When message is being sent it gets added to the queue and then broadcasted.
REJOIN: client sends message in format "REJOIN_[roomid]" (size: 6 bytes + id size) after which server accesses client's structure in vector and changes it's assigned room.

Possible use cases



Screenshots and code examples:

Connect to server:

```
Server is listening on port 8080...
Client 356 connected.
```

```
Connected to server.
Enter room ID (1, 2, or 3):
```

```
std::lock_guard<std::mutex> lock(consoleMutex);
std::cout << "Client " << clientSocket << " connected.\n";
std::thread clientThread(handleClient, clientSocket);
clientThread.detach();
```

Join room:

```
Server is listening on port 8080...
Client 356 connected.
Client 356 connected to room 1
```

```
Connected to server.
Enter room ID (1, 2, or 3): 1
```

```
if (bytesReceived > 0) {
    buffer[bytesReceived] = '\0';
    room = std::stoi(buffer);
    std::cout << "Client " << clientSocket << " connected to room " << room << std::endl;
}

addClient(clientSocket, room);
```

Change room:

```
Server is listening on port 8080...
Client 356 connected.
Client 356 connected to room 1
Client 356 rejoined room 2
```

```
Connected to server.
Enter room ID (1, 2, or 3): 1
REJOIN_2
```

```
if (message.substr(0, 7) == "REJOIN_") {
    int newRoom = std::stoi(message.substr(7));
    if (newRoom >= 1 && newRoom <= 3) {
        std::lock_guard<std::mutex> lock(consoleMutex);
        std::cout << "Client " << clientSocket << " rejoined room " << newRoom << std::endl;
        {
            std::lock_guard<std::mutex> lock(clientsMutex);
            for (size_t i = 0; i < clients.size(); ++i) {
                room = newRoom;
                if (clients[i].socket == clientSocket) clients[i].room = newRoom;
            }
        }
    }
}
```

Disconnect from server:

```
Client 356 disconnected from room 2
```

```
if (bytesReceived <= 0) {
    std::lock_guard<std::mutex> lock(consoleMutex);
    std::cout << "Client " << clientSocket << " disconnected from room " << room << std::endl;
    break;
}
```

Send/receive message:

```
Connected to server.
Enter room ID (1, 2, or 3): 2
Hello
```

```
Connected to server.
Enter room ID (1, 2, or 3): 1
REJOIN_2
Client: Hello
```

```
    buffer[bytesReceived] = '\0';
    std::string message(buffer);

    if (message.substr(0, 7) == "REJOIN_") {
        int newRoom = std::stoi(message.substr(7));
        if (newRoom >= 1 && newRoom <= 3) {
            std::lock_guard<std::mutex> lock(consoleMutex);
            std::cout << "Client " << clientSocket << " rejoined room " << newRoom << std::endl;
            {
                std::lock_guard<std::mutex> lock(clientsMutex);
                for (size_t i = 0; i < clients.size(); ++i) {
                    room = newRoom;
                    if (clients[i].socket == clientSocket) clients[i].room = newRoom;
                }
            }
        }
    }
    else {
        addMessageToQueue({ message, clientSocket, room });
    }
```

```
void addMessageToQueue(const Message& message) {
    std::lock_guard<std::mutex> lock(messageQueueMutex);
    messageQueue.push(message);
    messageAvailableCondition.notify_one();
}

void broadcastMessage(const std::string& message, SOCKET senderSocket, int room) {
    std::lock_guard<std::mutex> lock(consoleMutex);
    std::cout << "Client " << senderSocket << ": " << message << std::endl;
    {
        std::lock_guard<std::mutex> lock(clientsMutex);
        for (const Client& client : clients) {
            if (client.socket != senderSocket && client.room == room) {
                send(client.socket, message.c_str(), message.size() + 1, 0);
            }
        }
    }
}

void broadcastMessages() {
    while (true) {
        std::unique_lock<std::mutex> lock(messageQueueMutex);
        messageAvailableCondition.wait(lock, [] { return !messageQueue.empty(); });
        while (!messageQueue.empty()) {
            Message message = messageQueue.front();
            messageQueue.pop();
            broadcastMessage(message.message, message.senderSocket, message.room);
        }
    }
}
```