Submission Worksheet

Submission Data

Course: IT114-003-F2025
Assignment: IT114 Milestone 1
Student: Alanor S. (ass89)
Status: Submitted | **Worksheet Progress:** 100%
Potential Grade: 10.00/10.00 (100.00%)
Received Grade: 0.00/10.00 (0.00%)
Started: 11/6/2025 1:10:40 PM
Updated: 11/8/2025 6:26:09 PM
Grading Link: https://learn.ethereallab.app/assignment/v3/IT114-003-F2025/it114-milestone-1/grading/ass89
View Link: https://learn.ethereallab.app/assignment/v3/IT114-003-F2025/it114-milestone-1/view/ass89

Instructions

- Overview Link: https://youtu.be/9dZPFwi76ak

1. Refer to Milestone1 of any of these docs:
    2. Rock Paper Scissors
    3. Basic Battleship
    4. Hangman / Word guess
    5. Trivia
    6. Go Fish
    7. Pictionary / Drawing
2. Ensure you read all instructions and objectives before starting.
3. Ensure you've gone through each lesson related to this Milestone
4. Switch to the Milestone1 branch
    1. `git checkout Milestone1` (ensure proper starting branch)
    2. `git pull origin Milestone1` (ensure history is up to date)
5. Copy `Part5` and rename the copy as `Project` (this new folder should be in the root of your repo)
6. Organize the files into their respective packages `Client`, `Common`, `Server`, `Exceptions`
    1. Hint: If it's open, you can refer to the Milestone 2 Prep lesson
7. Fill out the below worksheet
    1. Ensure there's a comment with your UCID, date, and brief summary of the snippet in each screenshot
    2. Since this Milestone was majorly done via lessons, the required comments should be placed in areas of analysis of the requirements in this worksheet. There shouldn't need to be any actual code changes beyond the restructure.
8. Once finished, click "Submit and Export"
9. Locally add the generated PDF to a folder of your choosing inside your repository folder and move it to Github
    1. `git add .`
    2. `git commit -m "adding PDF"`
    3. `git push origin Milestone1`
    4. On Github merge the pull request from `Milestone1` to `main`
10. Upload the same PDF to Canvas
11. Sync Local

```
1. git checkout main
2. git pull origin main
```

Section #1: (1 pt.) Feature: Server Can Be Started Via Command Line And Listen To Connections

Progress: 100%

Task #1 (1 pt.) - Evidence

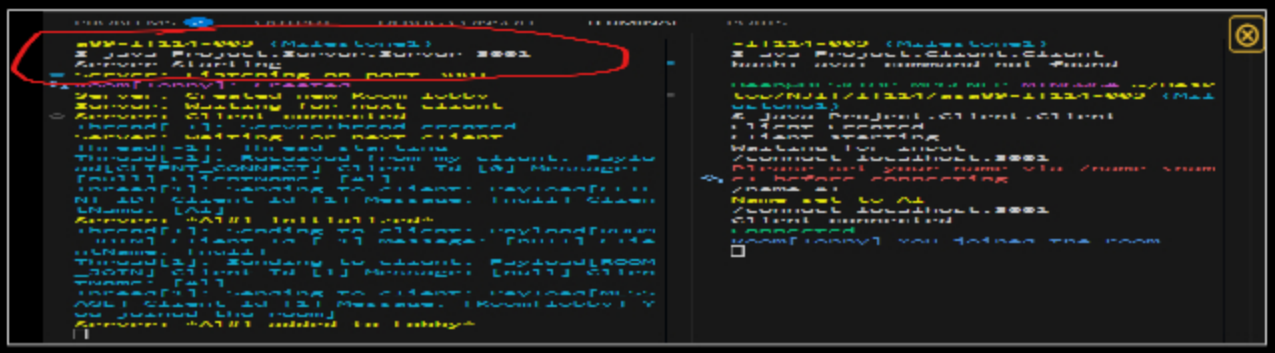
Progress: 100%

Part 1:

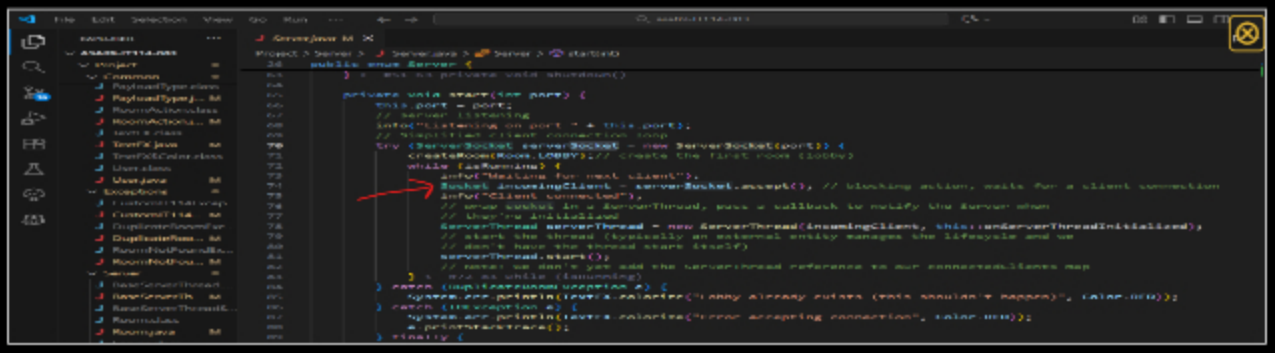
Progress: 100%

Details:

- Show the terminal output of the server started and listening
- Show the relevant snippet of the code that waits for incoming connections



Show the terminal output of the server started and listening



Show the relevant snippet of the code that waits for incoming connections

Saved: 11/8/2025 7:21:13 AM

Part 2:

Progress: 100%

Details:

- Briefly explain how the server-side waits for and accepts/handles connections

Your Response:

On the server side, the program waits for and handles client connections using a ServerSocket. When the server starts, it creates a ServerSocket bound to a specified port, the default in this case, and enters a continuous loop while it's running. Inside this loop, it calls serverSocket.accept(), a blocking operation; meaning the program pauses and waits until a client attempts to connect. Once a client connects, the server accepts the connection and creates a new ServerThread instance to handle communication with that client independently. The server then starts this thread, allowing multiple clients to connect and communicate concurrently without blocking new connections. When a ServerThread is initialized, the server assignns it a unique ID and adds it to the default room, in our case the Lobby, so it can interact with other clients. On high level explanation; the server continously listens for new clients, manages active connections, and handles them with multithreading.

Saved: 11/8/2025 7:21:13 AM

Section #2: (1 pt.) Feature: Server Should Be Able To Allow More Than One Client To Be Connected At Once

Progress: 100%

Task #1 (1 pt.) - Evidence

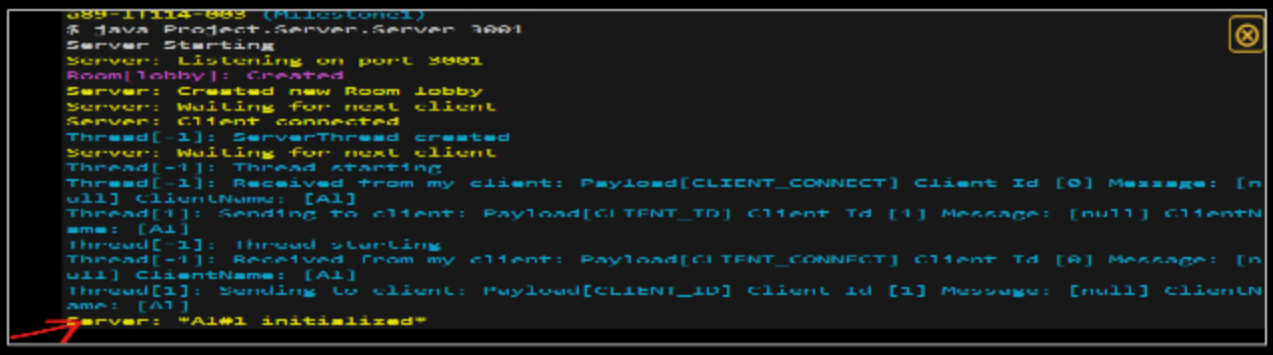
Progress: 100%

Part 1:

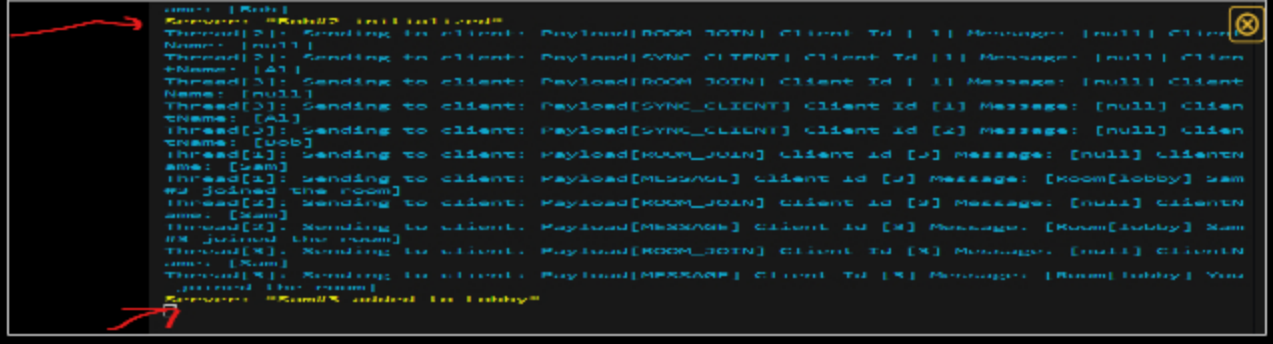
Progress: 100%

Details:

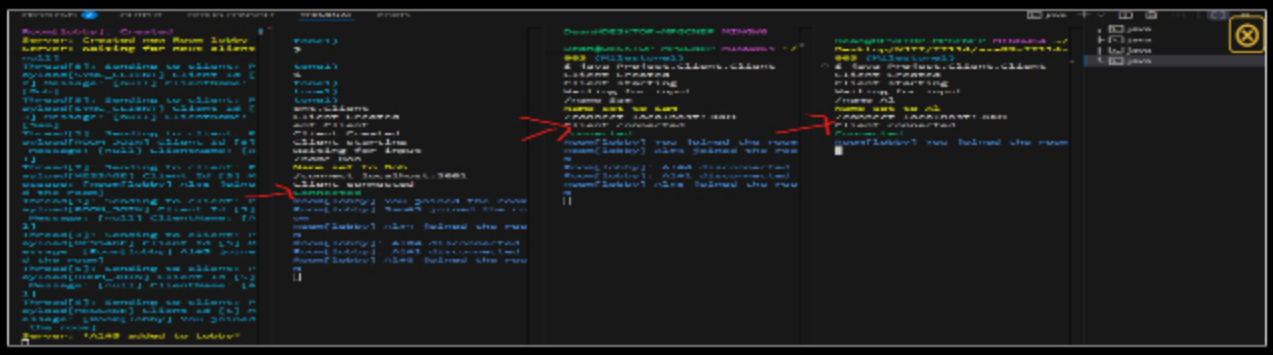
- Show the terminal output of the server receiving multiple connections
- Show at least 3 Clients connected (best to use the split terminal feature)
- Show the relevant snippets of code that handle logic for multiple connections



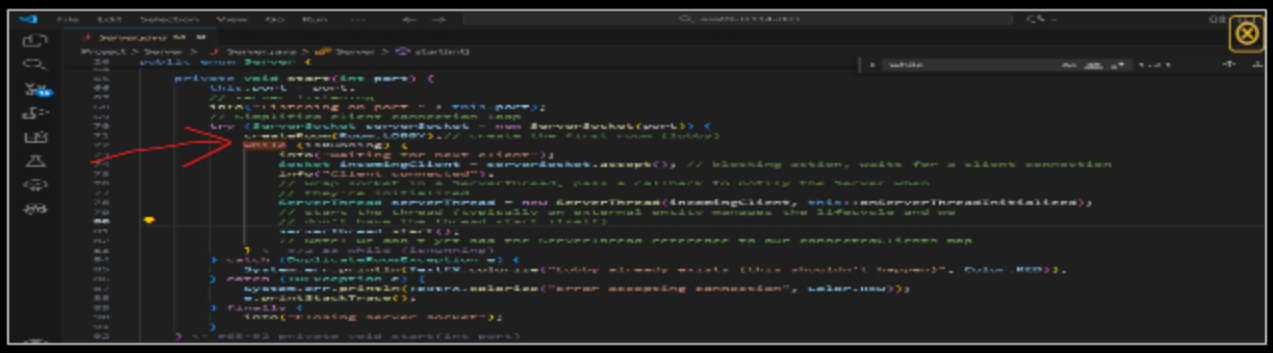
Show the terminal output of the server receiving multiple connections



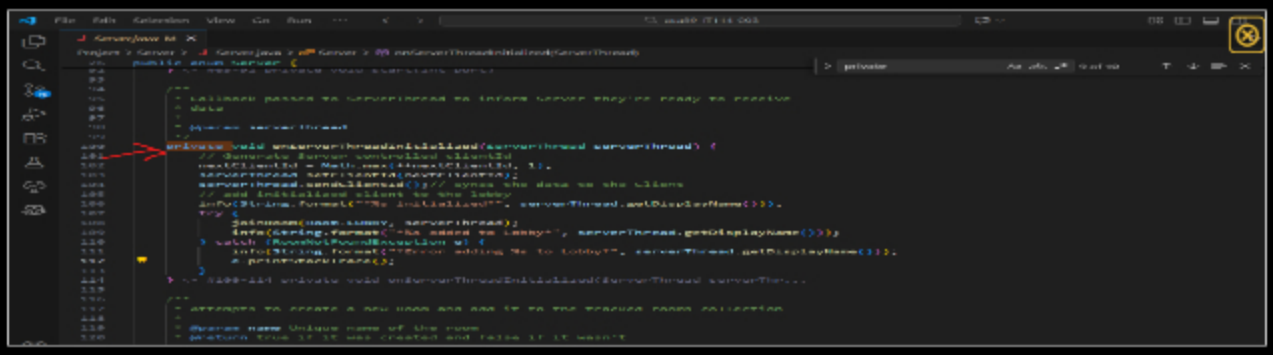
Show the terminal output of the server receiving multiple connections



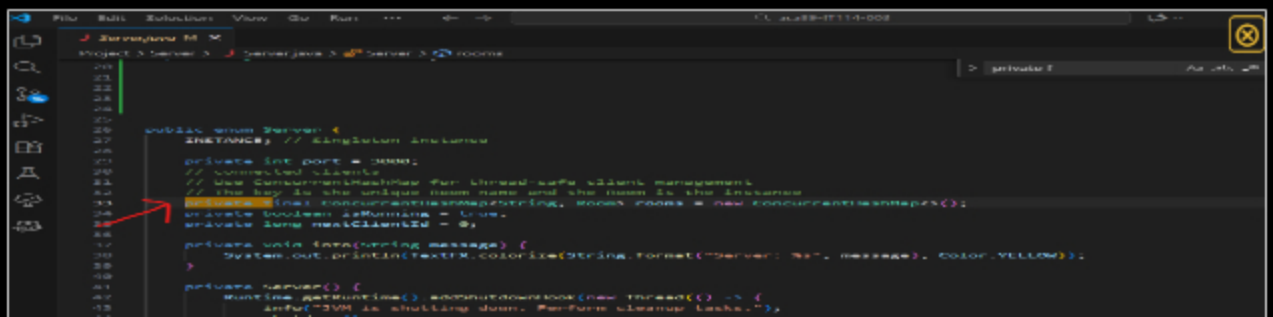
Show at least 3 Clients connected (best to use the split terminal feature) Bob, Sam, Al



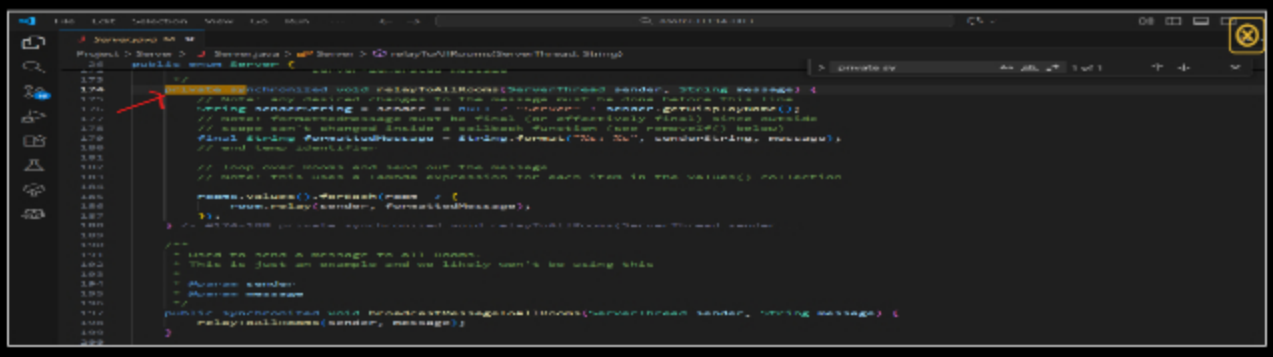
Show the relevant snippets of code that handle logic for multiple connections server side



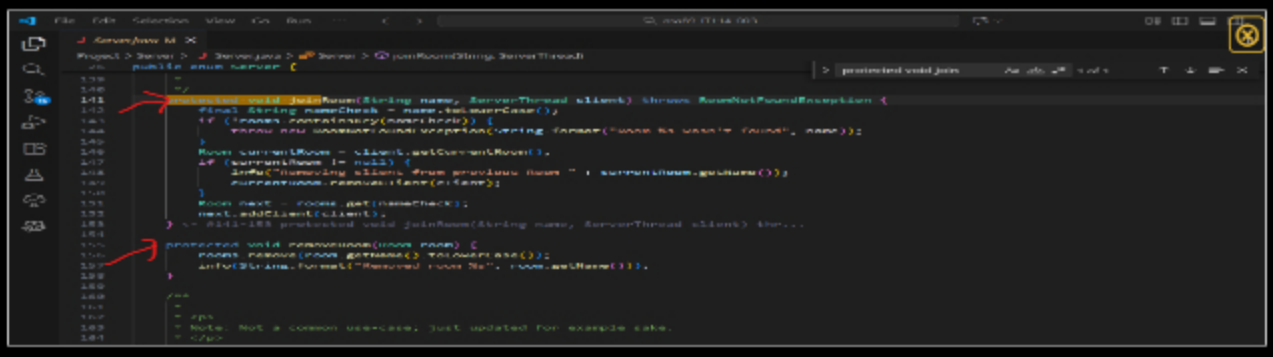
Show the relevant snippets of code that handle logic for multiple connections server side



Show the relevant snippets of code that handle logic for multiple connections server side



Show the relevant snippets of code that handle logic for multiple connections server side



Show the relevant snippets of code that handle logic for multiple connections server side

Saved: 11/8/2025 7:25:29 AM

Part 2:

Progress: 100%

Details:

- Briefly explain how the server-side handles multiple connected clients

Your Response:

The server hnadles multiple clients using multithreading. Each time a new client connects, the server creates a new ServerThread object dedicated to that client, including all sending, recieving, and processing messages, all independently of other clients. Because each client runs its own thread, the server can interact with many clients at the same time without blocking or waiting for one client's actions to finish before serving another, allowing for concurrent handling of messages.

Saved: 11/8/2025 7:25:29 AM

Section #3: (2 pts.) Feature: Server Will Implement The Concept Of Rooms (With The

Default Being "I obby")

Progress: 100%

Task #1 (2 pts.) - Evidence

Progress: 100%

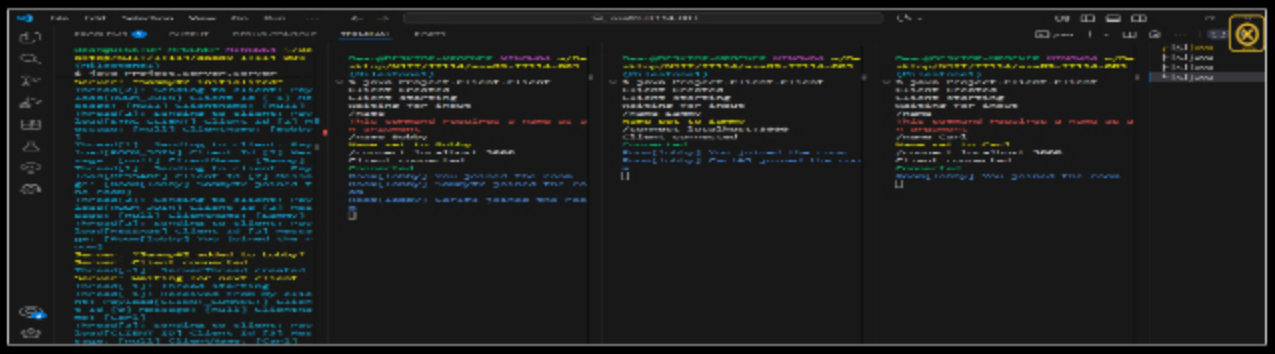
Part 1:

Progress: 100%

Details:

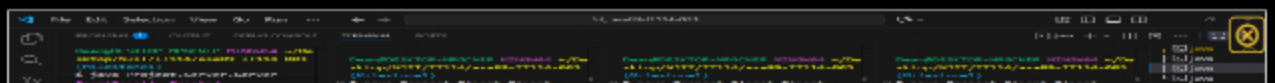
- Show the terminal output of rooms being created, joined, and removed (server-side)
- Show the relevant snippets of code that handle room management (create, join, leave, remove) (server-side)

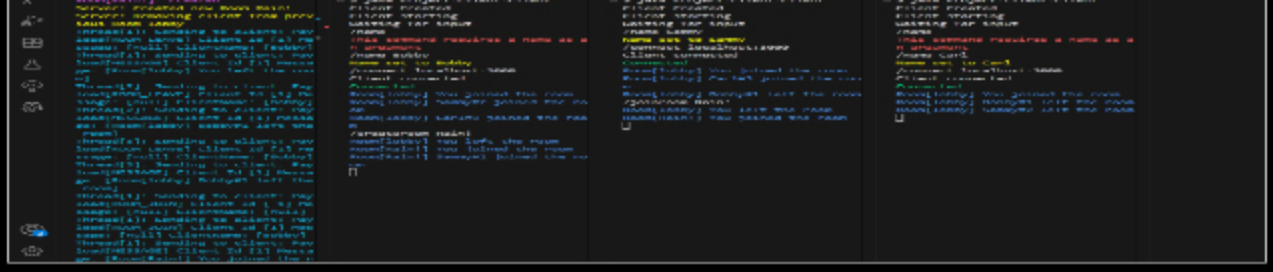
Show the terminal output of rooms being created, joined, and removed (server-side)



Show the terminal output of rooms being created, joined, and removed (server-side)

Show the terminal output of rooms being created, joined, and removed (server-side)

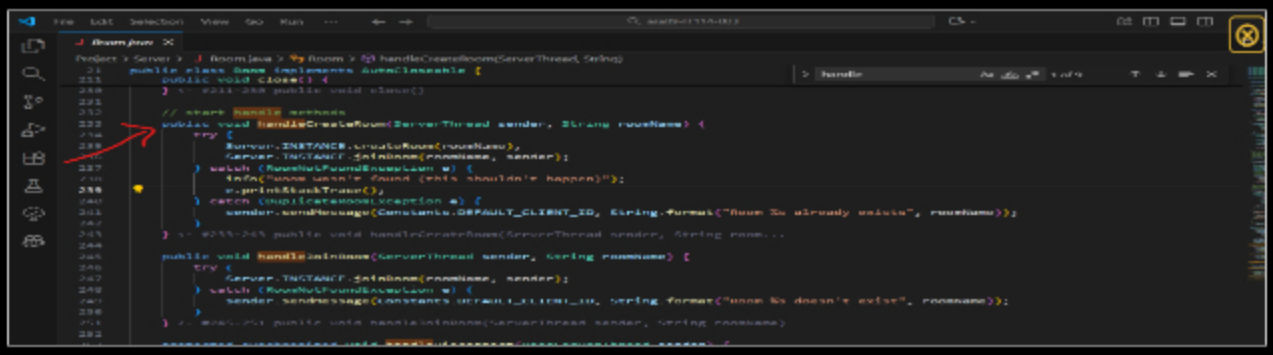




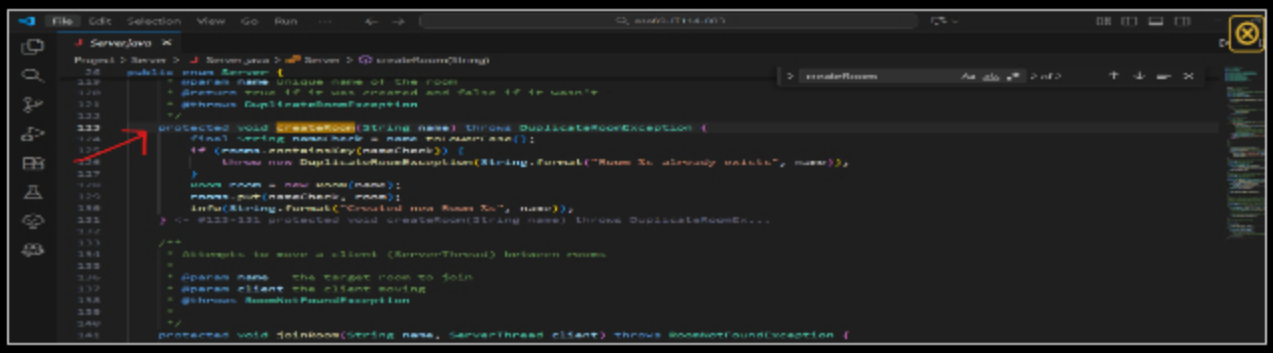
Creating and joining a room

Creating and joining a room

User joining and leaving a room



code for creating a room in Room.java



code for creating a room in Server.java

```
public class Room implements AutoCloseable {
    public void handleCreateRoom(ServerThread sender, String roomName) {
        sender.sendMessage(Constants.DEFAULT_CLIENT_ID, String.format("Room %s already exists", roomName));
    }

    public void handleJoinRoom(ServerThread sender, String roomName) {
        try {
            Server.INSTANCE.joinRoom(roomName, sender);
        } catch (RoomNotFoundException e) {
            sender.sendMessage(Constants.DEFAULT_CLIENT_ID, String.format("Room %s doesn't exist", roomName));
        }
    }

    protected synchronized void handleDisconnect(BaseServerThread sender) {
        handleDisconnect((ServerThread) sender);
    }

    /**
     * Expose access to the disconnect action
     *
     * @param serverThread
```

code for joining a room in Room.java

```
public enum Server {
    /**
    protected void joinRoom(String name, ServerThread client) throws RoomNotFoundException {
        final String nameCheck = name.toLowerCase();
        if (rooms.containsKey(nameCheck)) {
            throw new RoomNotFoundException(String.format("Room %s wasn't found", name));
        }
        Room currentRoom = client.getCurrentRoom();
        if (currentRoom != null) {
            info("Removing client from previous room " + currentRoom.getName());
            currentRoom.removeClient(client);
        }
        Room next = rooms.get(nameCheck);
        next.addClient(client);
    }

    protected void removeRoom(Room room) {
        rooms.remove(room.getName().toLowerCase());
        info(String.format("Removed room %s", room.getName()));
    }

    /**
     *
     * rps
```

code for joining a room in Server.java

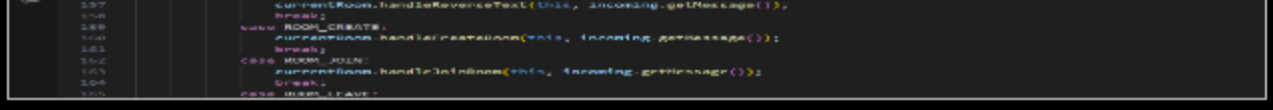
```
Room.java
Project > Server > Room.java > Room > handleDisconnect(BaseServerThread)
public class Room implements AutoCloseable {
    */
    protected synchronized void handleDisconnect(ServerThread sender) {
        disconnect(sender);
    }

    protected synchronized void handleReverseText(ServerThread sender, String text) {
        StringBuilder sb = new StringBuilder(text);
        sb.reverse();
        String rev = sb.toString();
        relay(sender, rev);
    }

    protected synchronized void handleMessage(ServerThread sender, String text) {
        relay(sender, text);
    }
    // end handle methods
} <- #21-277 public class Room implements AutoCloseable
```

leaving/disconnecting in room.java part 1

```
Room.java
Project > Server > Room.java > Room > disconnect(ServerThread)
public class Room implements AutoCloseable {
    protected synchronized void disconnect(ServerThread client) {
        if (isRunning) { // stop action if room isn't running
            return;
        }
        ServerThread disconnectingServerThread = clientsInRoom.remove(client.getClientId());
        if (disconnectingServerThread != null) {
            clientsInRoom.values().removeIf(serverThread -> {
                if (serverThread.getClientId() == disconnectingServerThread.getClientId()) {
                    return true;
                }
                boolean failedToSend = serverThread.sendClientInfo(disconnectingServerThread.getClientId(),
                        disconnectingServerThread.getClientName(), DisconnectRoom);
                if (failedToSend) {
                    System.out.printing(
                            String.format("Removing disconnected %s from list", serverThread.getDisplayName()));
                    disconnect(serverThread);
                } <- #178-188 if (failedToSend)
                return failedToSend;
            });
            relay(null, disconnectingServerThread.getDisplayName() + " disconnected");
            disconnectingServerThread.disconnect();
        } <- #169-189 if (disconnectingServerThread != null)
        autoCleanup();
```

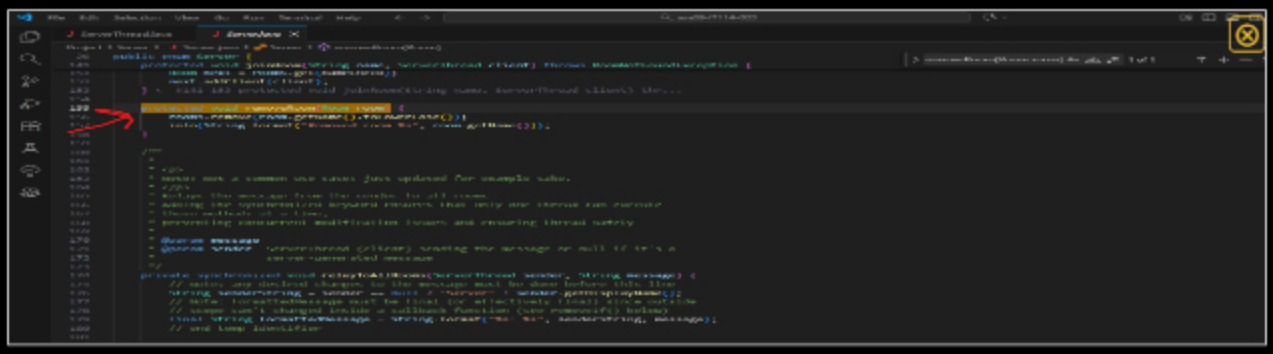
leaving/disconnecting in room.java part 2

```
ServerThread.java
Project > Server > ServerThread.java > ServerThread > processPayload(Payload)
public class ServerThread extends BaseServerThread {
    @Override
    protected void processPayload(Payload incoming) {

        switch (incoming.getPayloadType()) {
            case CLIENT_CONNECT:
                setClientName(((ConnectionPayload) incoming).getClientName().trim());

                break;
            case DISCONNECT:
                currentRoom.handleDisconnect(this);
                break;
            case MESSAGE:
                currentRoom.handleMessage(this, incoming.getMessage());
                break;
            case REVERSE:
```

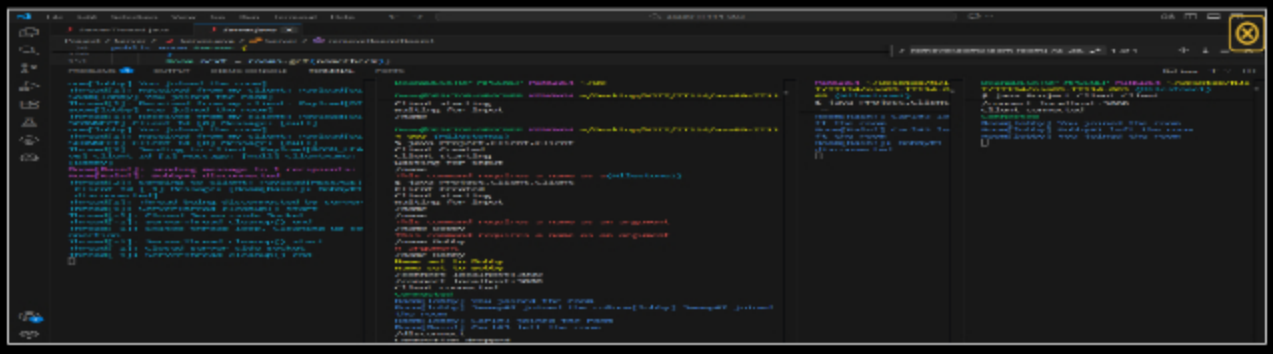


leaving/disconnecting in ServerThread.java. ServerThread.java calls this on PayloadType.DISCONNECT

removing/cleaning up a room in Room.java



removing/cleaning up a room in Server.java



/disconnect command

Saved: 11/8/2025 9:07:43 AM

Part 2:

Progress: 100%

Details:

- Briefly explain how the server-side handles room creation, joining/leaving, and removal

Your Response:

The server manages rooms using a ConcurrentHashMap, where each room name maps to a Room object. When a new room is created, the createRoom() method checks if a room with the

Room object. When a new room is created, the createRoom() method checks if a room with the same name already exists; if not, it adds the new Room instance to the map. To join a room, the joinRoom() method first removes the client from their current room, if any, and then adds them to the new one using that room's addClient() method. If a client leaves and the room becomes empty, the room can later be removed from the map using removeRoom(), which deletes it by name.

Saved: 11/8/2025 9:07:43 AM

Section #4: (1 pt.) Feature: Client Can Be Started Via The Command Line

Progress: 100%

Task #1 (1 pt.) - Evidence

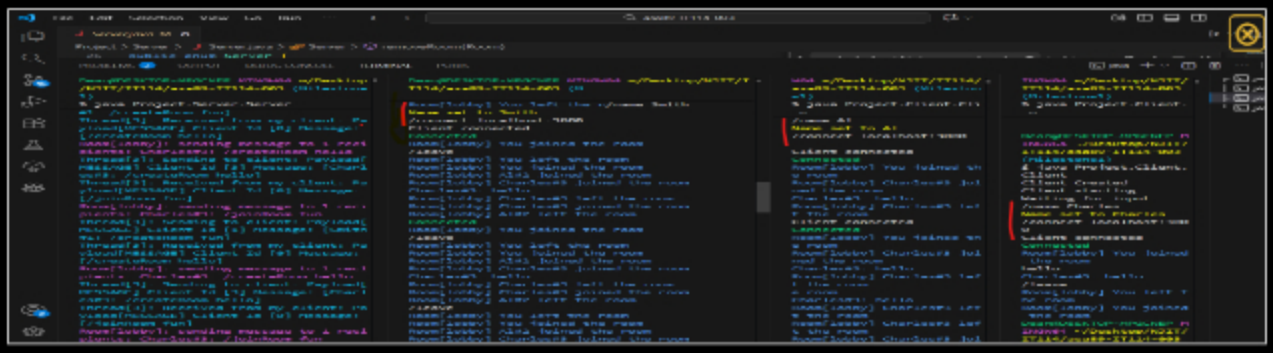
Progress: 100%

Part 1:

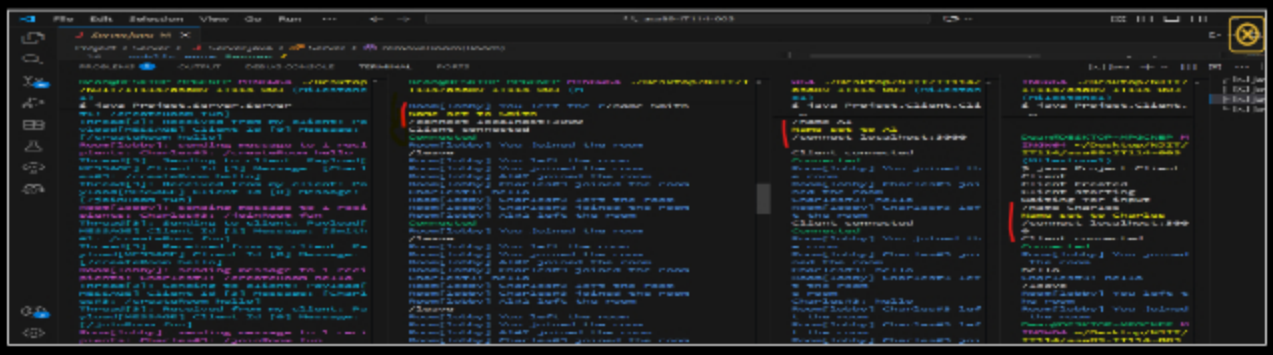
Progress: 100%

Details:

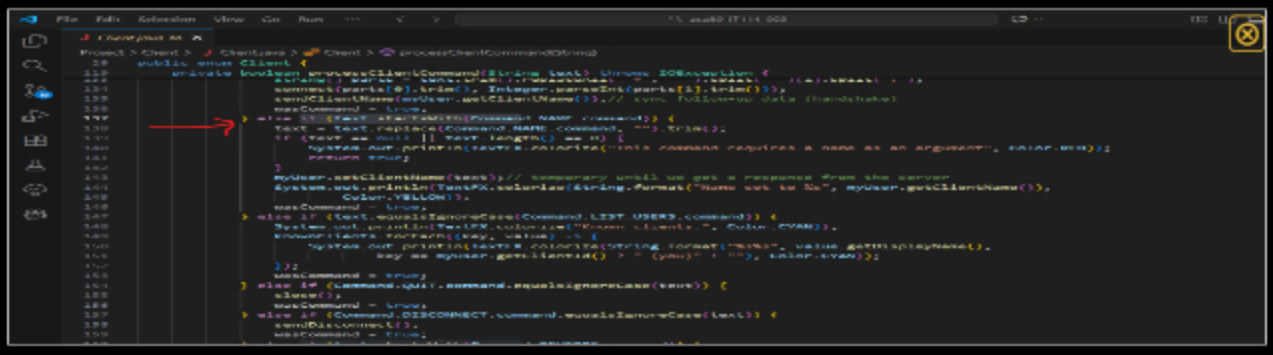
- Show the terminal output of the /name and /connect commands for each of 3 clients (best to use the split terminal feature)
- Output should show evidence of a successful connection
- Show the relevant snippets of code that handle the processes for /name, /connect, and the confirmation of being fully setup/connected



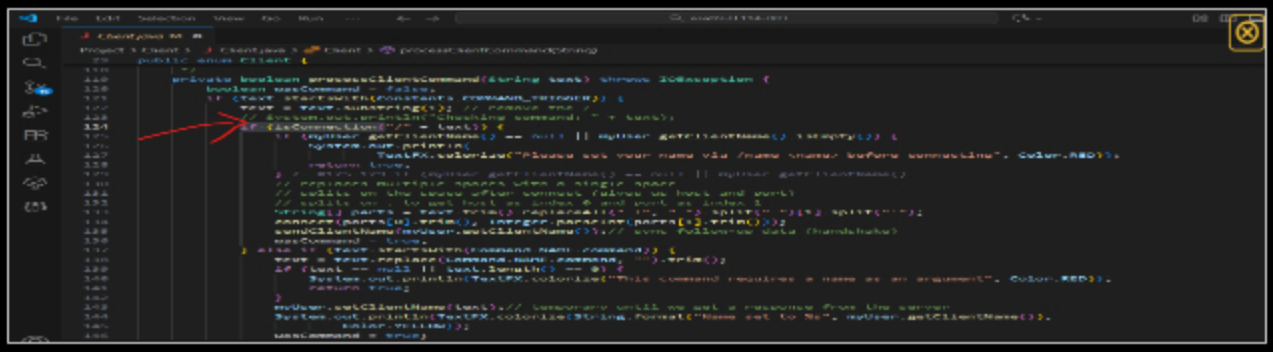
Show the terminal output of the /name and /connect commands for each of 3 clients (best to use the split terminal feature)



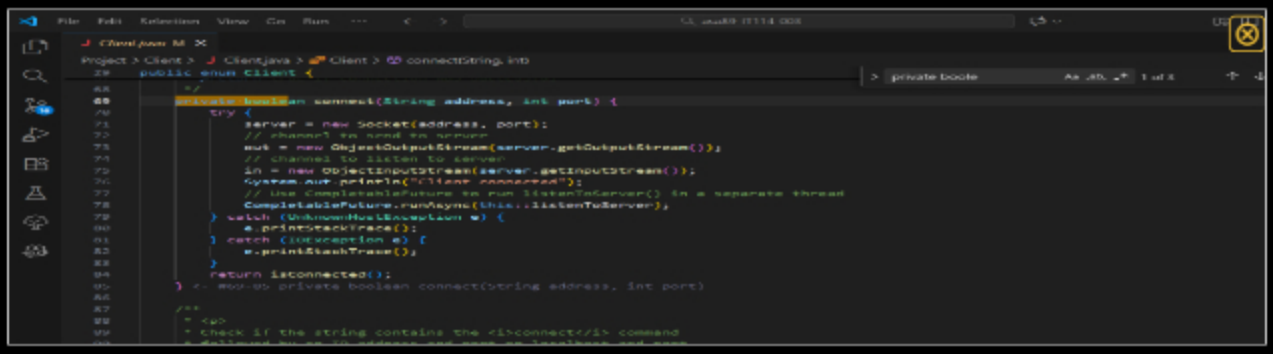
Output should show evidence of a successful connection



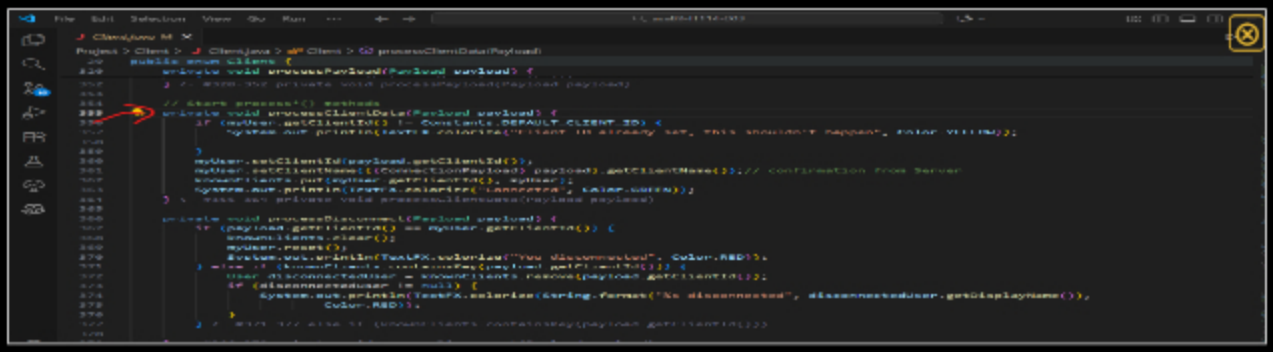
Show the relevant snippets of code that handle the processes for /name



Show the relevant snippets of code that handle the processes for /connect pt 1



Show the relevant snippets of code that handle the processes for /connect pt 2



Show the relevant snippets of code that handle the processes for confirmation of being fully setup/conn

Saved: 11/8/2025 7:41:21 AM

Part 2:

Progress: 100%

Details:

- Briefly explain how the /name and /connect commands work and the code flow that leads to a successful connection for the client

Your Response:

The /name command allows the client to set their display name before connecting to the server. When the user types in /name Al for example, the input is processed by the processClientCommand() method, which detects the /name keyword and stores "Al" locally in the myUser object without sending anything over the network. The /connect command, on the other hand, establishes the actual connection to the server. When the user types /connect localhost:3000, the client validates the input, ensures a name has been set, and then calls the connect() method to create a socket connection to the specified host and port. Once connected, it sends a CLIENT_CONNECT payload containing the client's name to the server. On the server side, the ServerThread recieves the payload, assigns the name to the client, and responds with a CLIENT_ID payload confirming the connection. The client's listenToServer() method processes this response, updates the client's ID and name, and prints "Connected" to confirm success. On a high level; /name prepares the client's identity, and /connect initiates and completes the handshake that establishes the client's connection to the server.

Saved: 11/8/2025 7:41:21 AM

Section #5: (2 pts.) Feature: Client Can Create/j oin Rooms

Progress: 100%

Task #1 (2 pts.) - Evidence

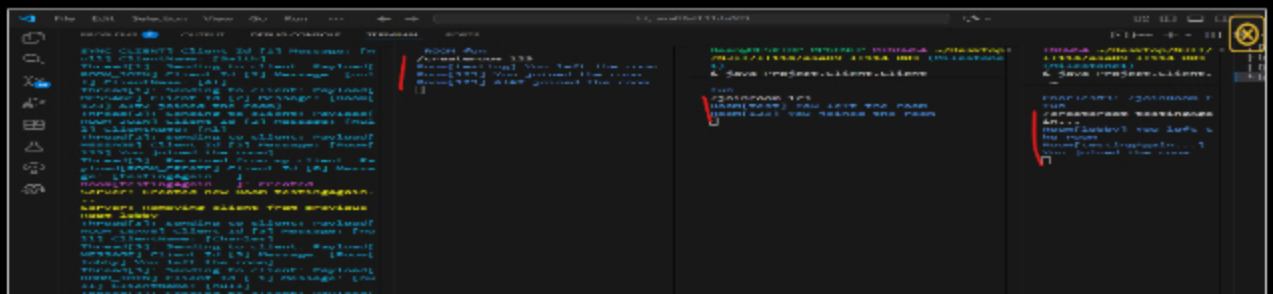
Progress: 100%

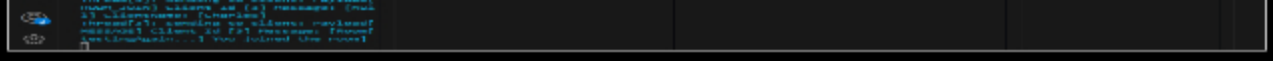
Part 1:

Progress: 100%

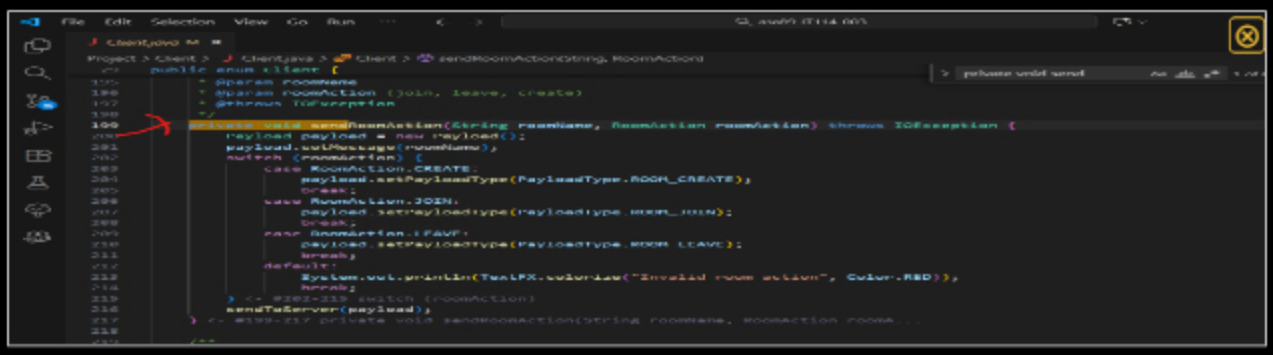
Details:

- Show the terminal output of the /createroom and /joinroom
- Output should show evidence of a successful creation/join in both scenarios
- Show the relevant snippets of code that handle the client-side processes for room creation and joining

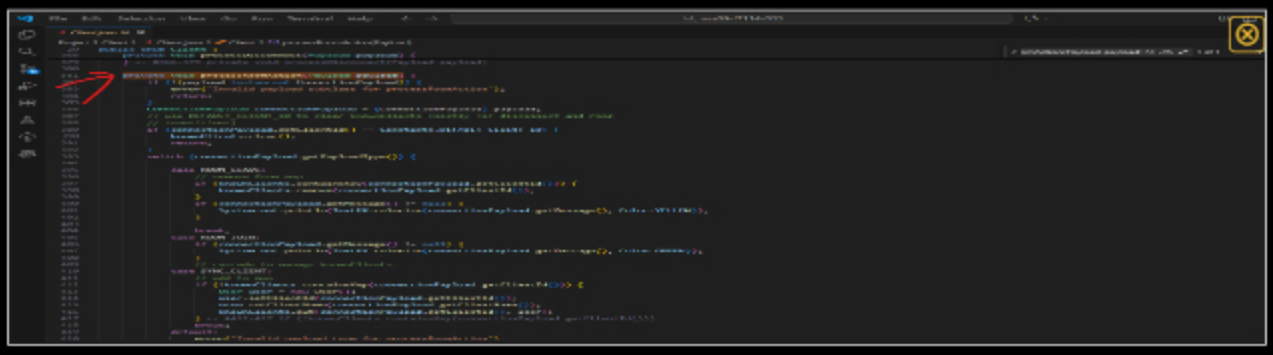




Show the terminal output of the /createroom and /joinroom Output should show evidence of a successful creation/join in both

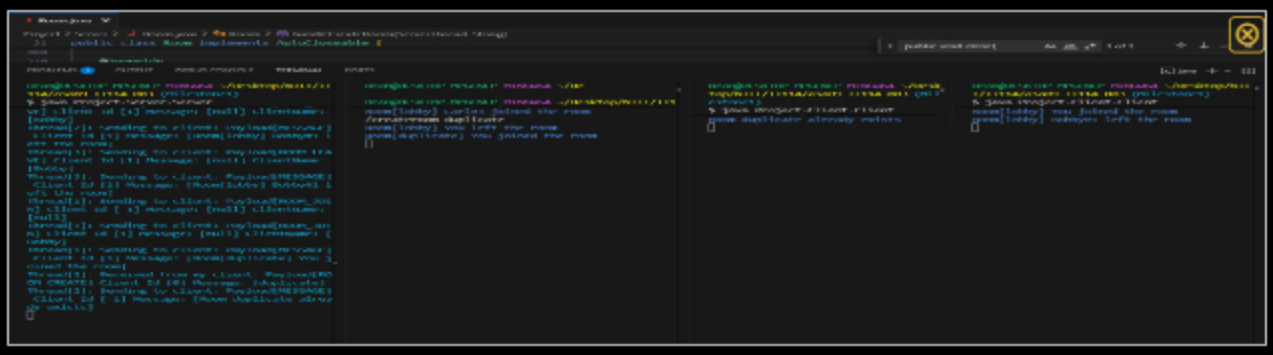


Show the relevant snippets of code that handle the client-side processes for room creation and joining



Show the relevant snippets of code that handle the client-side processes for room creation and joining

Show the relevant snippets of code that handle the client-side processes for room creation and joining



duplicate room handled gracefully

Saved: 11/8/2025 6:26:09 PM

Part 2:

Progress: 100%

Details:

- Briefly explain how the /createroom and /join room commands work and the related code flow for each

Your Response:

When a client issues the /createroom or /joinroom command, the client-side Client class first parses the input and calls the sendRoomAction() method. This method creates a Payload object, sets it PayloadType to either ROOM_CREATE or ROOM_JOIN, and includes the target room name in the message field. The payload is then sent to the server over the established socket connection. On the server side, the ServerThread's processPayload() method recieves the payload and routes it to the appropriate handler in the client's current room: handleCreateRoom() for creation or handleJoinRoom() for joining. These handlers interact with the Server to either create a new Room object, if the room doesn't already exist, or move the client from their current room to the target room, updating internal room membership structures. The server then sends a ConnectionPayload back to the client, confirming the room creation or successful join, which the client processes in processRoomAction() to update its local state and display feedback, completing the command's flow.

Saved: 11/8/2025 6:26:09 PM

Section #6: (1 pt.) Feature: Client Can Send Messages

Progress: 100%

Task #1 (1 pt.) - Evidence

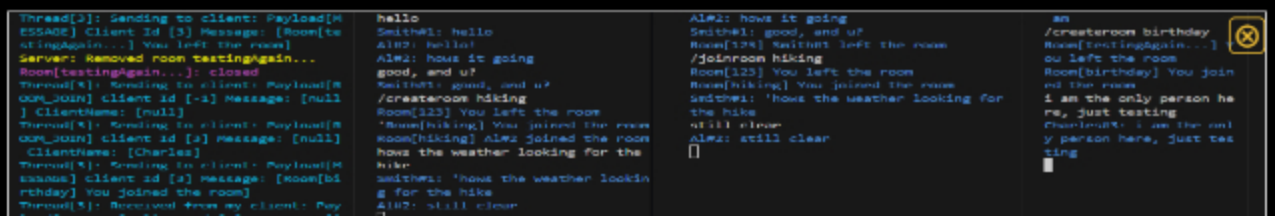
Progress: 100%

Part 1:

Progress: 100%

Details:

- Show the terminal output of a few messages from each of 3 clients
- Include examples of clients grouped into other rooms
- Show the relevant snippets of code that handle the message process from client to server-side and back



```
Load[MESSAGE] Client Id [0] Message: [I
 am the only person here, just testing]
Room[birthday]: sending message to 1 re
cipients: Charles#3: I am the only pers
on here, just testing
Thread[5]: Sending to client: Payload[M
ESSAGE] Client Id [3] Message: [Charles
#3: I am the only person here, just tes
ting]
```

Include examples of clients grouped into other rooms AI and Smith in same room (hiking), Charles in room by himself (birthday)

```
Thread[5]: Sending to client: Payload[M
ESSAGE] Client Id [4] Message: [Room[te
stingAgain...] You left the room]
Server: Removed room testingAgain...
Room[testingAgain...]: closed
Thread[2]: Sending to client: Payload[R
OOM_JOIN] Client Id [-1] Message: [null
] ClientName: [null]
Thread[2]: Sending to client: Payload[R
OOM_JOIN] Client Id [3] Message: [null]
 ClientName: [Charles]
Thread[2]: Sending to client: Payload[M
ESSAGE] Client Id [3] Message: [Room[bi
rthday] You joined the room]
Thread[2]: Received from my client: Pay
load[MESSAGE] Client Id [0] Message: [I
 am the only person here, just testing]
Room[birthday]: sending message to 1 re
cipients: Charles#3: I am the only pers
on here, just testing
Thread[5]: Sending to client: Payload[M
ESSAGE] Client Id [3] Message: [Charles
#3: I am the only person here, just tes
ting]
```

```
hello
Smith#1: hello
AI#2: hello!
AI#2: how is it going
good, and u?
Smith#1: good, and u?
/createroom hiking
Room[123] You left the room
'Room[hiking] You joined the room
Room[hiking] AI#2 joined the room
hows the weather looking for the
hike
Smith#1: 'hows the weather lookin
g for the hike
AI#2: still clear
```

```
AI#2: hows it going
Smith#1: good, and u?
Room[123] Smith#1 left the room
/joinroom hiking
Room[123] You left the room
Room[hiking] You joined the room
Smith#1: 'hows the weather looking for
the hike
still clear
AI#2: still clear
```

```
 am
/createroom birthday
Room[testingAgain...]
ou left the room
Room[birthday] You join
ed the room
i am the only person he
re, just testing
Charles#3: I am the onl
y person here, just tes
ting
```

Show the terminal output of a few messages from each of 3 clients

```
File Edit Selection View Go Run ...
Client.java M ×
Project > Client > Client.java > Client
public enum Client {
    * @param message
    * @throws IOException
    */
    private void sendMessage(String message) throws IOException {
        Payload payload = new Payload();
        payload.setMessage(message);
        payload.setPayloadType(PayloadType.MESSAGE);
        sendToServer(payload);
    } <- #250-255 private void sendMessage(String message) throws IOException

    /**
    * Sends the client's name to the server (what the user desires to be called)
    *
    * @param name
    * @throws IOException
    */
    private void sendClientName(String name) throws IOException {
        ConnectionPayload payload = new ConnectionPayload();
        payload.setClientName(name);
        payload.setPayloadType(PayloadType.CLIENT_CONNECT);
        sendToServer(payload);
    } <- #262-268 private void sendClientName(String name) throws IOException
```

Client-side: Sending a message to the server

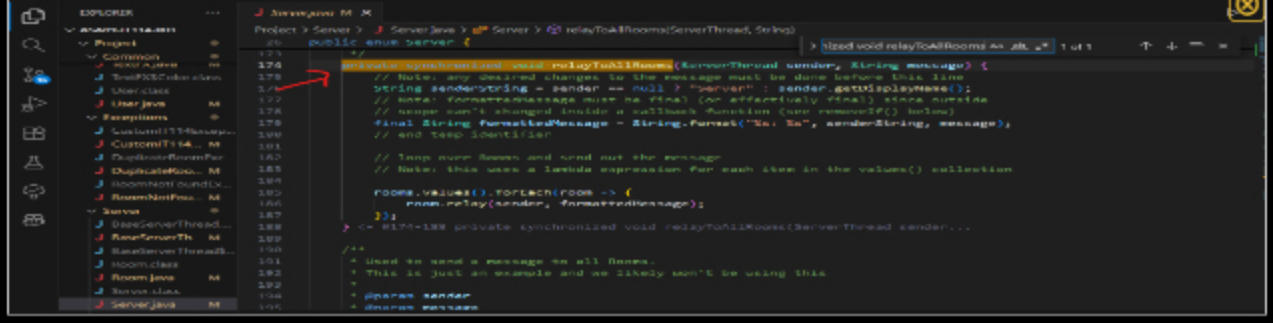
```
File Edit Selection View Go Run ...
Client.java M ×
Project > Client > Client.java > Client > sendToServer(Payload)
public enum Client {
    */
    private void sendClientName(String name) throws IOException {
        ConnectionPayload payload = new ConnectionPayload();
        payload.setClientName(name);
        payload.setPayloadType(PayloadType.CLIENT_CONNECT);
        sendToServer(payload);
    } <- #265-268 private void sendClientName(String name) throws IOException

    private void sendToServer(Payload payload) throws IOException {
        if (isConnected()) {
            out.writeObject(payload);
            out.flush(); // good practice to ensure data is written out immediately
        } else {
            System.out.println(
                "Not connected to server (hint: type `/connect host:port` without the quotes and replace host/port with the necessary
        }
    } <- #270-278 private void sendToServer(Payload payload) throws IOException
    // End Send*() methods

    public void start() throws IOException {
```

Client-side: Sending a message to the server

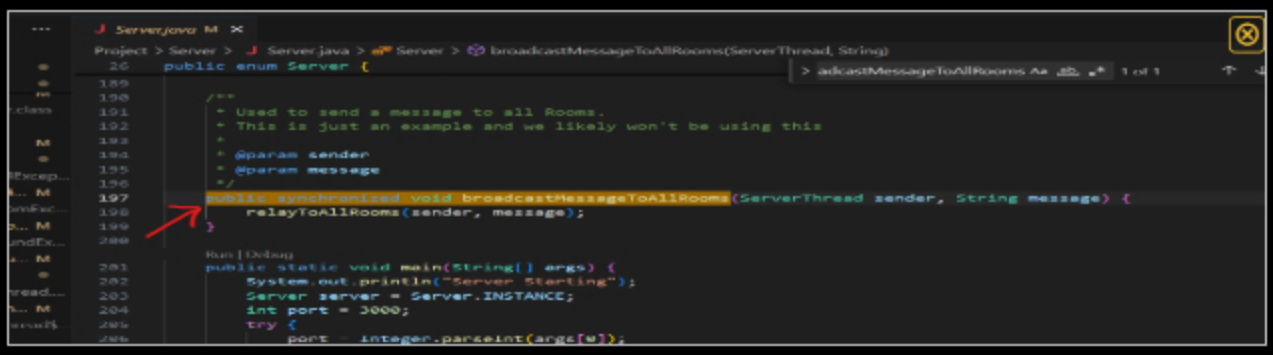
```
File Edit Selection View Go Run ...
Client.java M ×
Project > Client > Client.java > Client > listenToInput()
public enum Client {
    /**
    * Listens for keyboard input from the user
    */
    private void listenToInput() {
        try (Scanner si = new Scanner(System.in)) {
            System.out.println("Waiting for input"); // moved here to avoid console spam
            while (isRunning) { // Run until isRunning is false
                String userInput = si.nextLine();
                if (processClientCommand(userInput)) {
                    sendMessage(userInput);
                }
            } <- #440-445 while (isRunning)
        } catch (IOException ioException) {
            System.out.println("Error in listenToInput()");
            ioException.printStackTrace();
        }
        System.out.println("listenToInput thread stopped");
    } <- #437-451 private void listenToInput()

    /**
```

Triggering message send: From user input

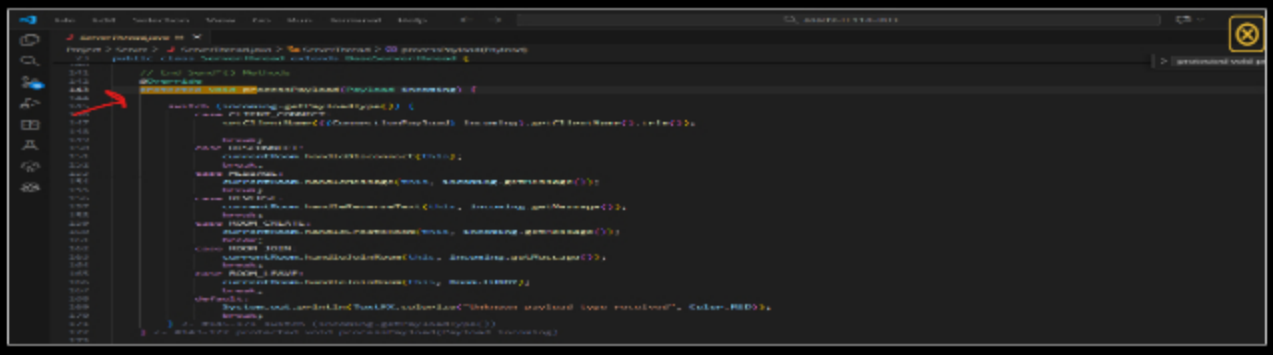
```
File Edit Selection View Go Run ...
```



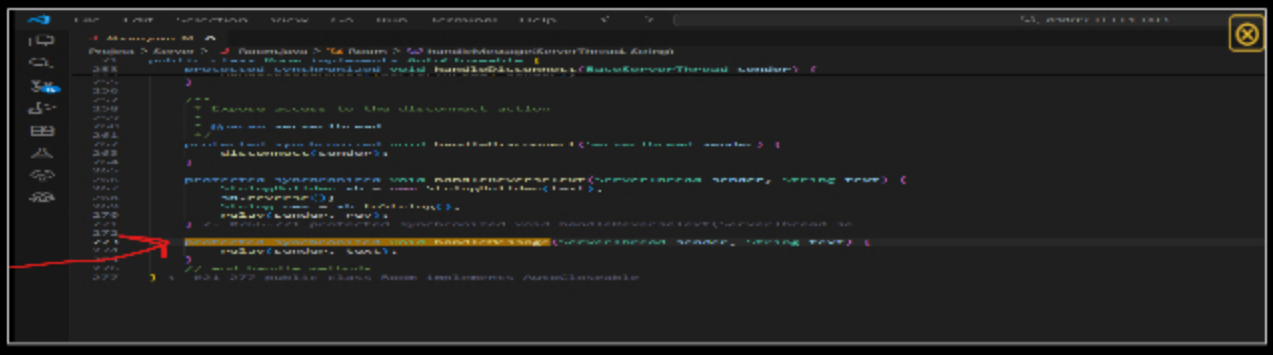
Server-side: Relaying messages to clients



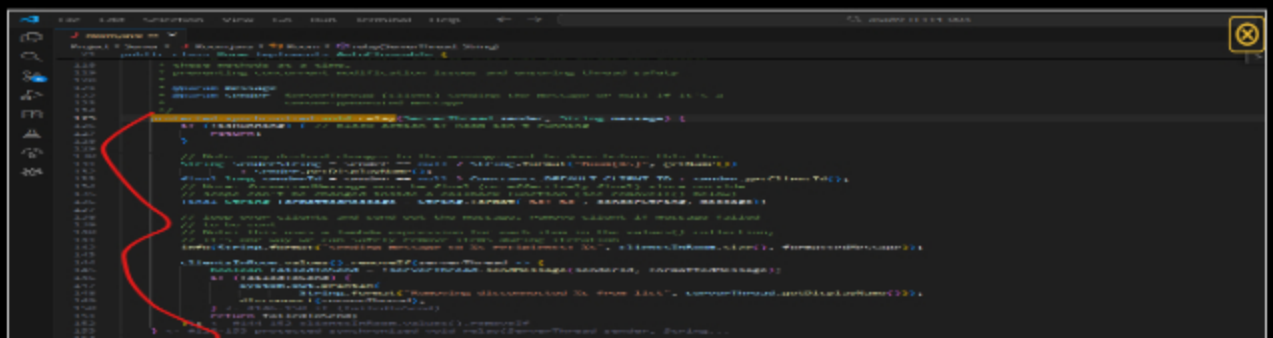
Server-side: Relaying messages to clients



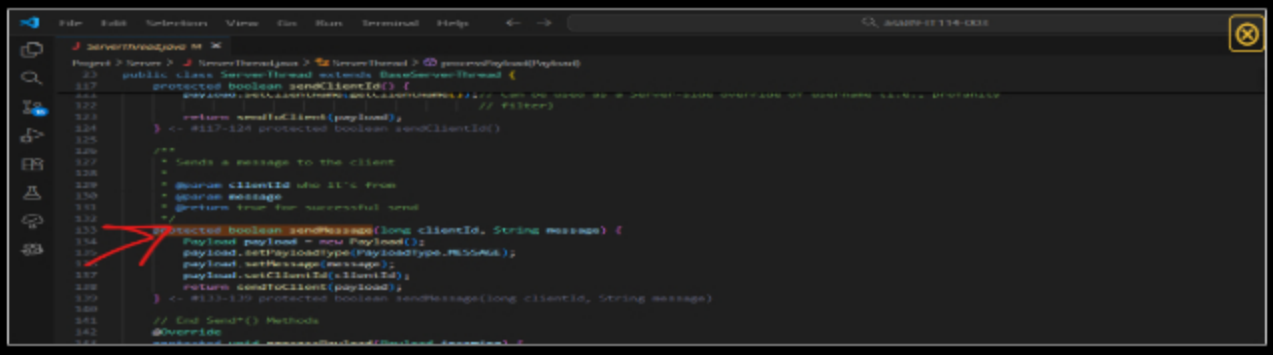
ServerThread receives the payload



Room relays the message



Room relays the message



ServerThread sends back to clients

Saved: 11/8/2025 8:11:15 AM

Part 2:

Progress: 100%

Details:

- Briefly explain how the message code flow works

Your Response:

When a client types a regular message (not a command), the client-side Client class detects that it's not a recognized command in processClientCommand(). It then creates a Payload object with PayloadType.MESSAGE and the message text, and sends it to the server via sendToServer(). On the server side, the ServerThread handling that client recieves the payload in processPayload(), recognizes it as a MESSAGE type, and forwards it to the current room's handleMessage() method. The Room object then iterates over all connected clients in that room and relays the message to each client's ServerThread using sendMessage(). Each recieving client processes the incoming payload in its processPayload() method, specifically in processMessage(), and prints the message to the console. Meaning, that a message typed by one client gets delivered to all other clients in the same room in real time.

Saved: 11/8/2025 8:11:15 AM

Section #7: (1 pt.) Feature: Disconnection

Progress: 100%

Task #1 (1 pt.) - Evidence

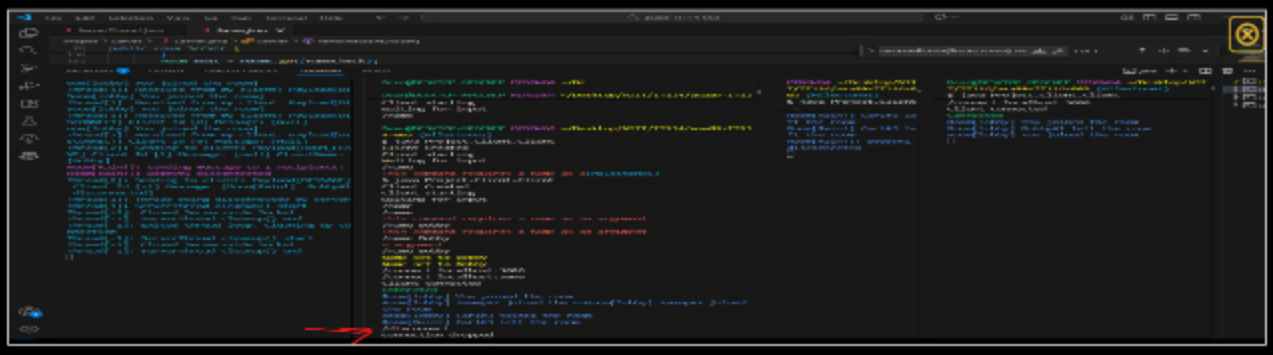
Progress: 100%

Part 1:

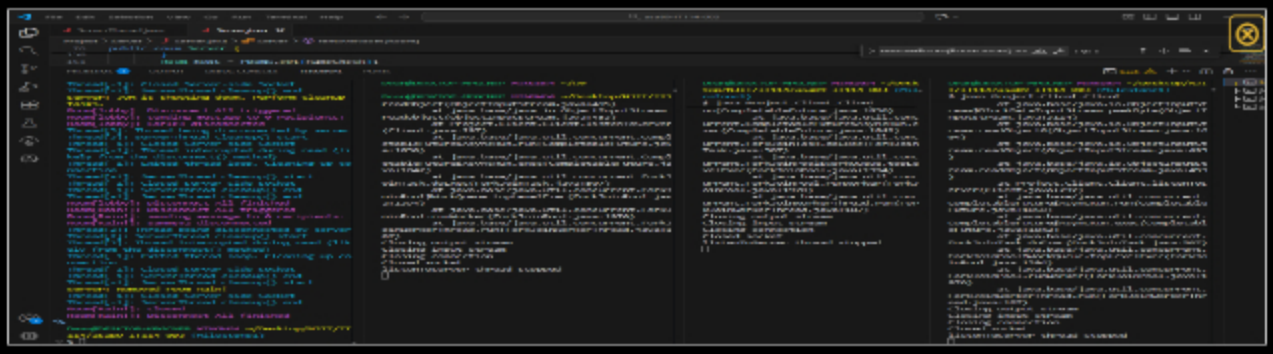
Progress: 100%

Details:

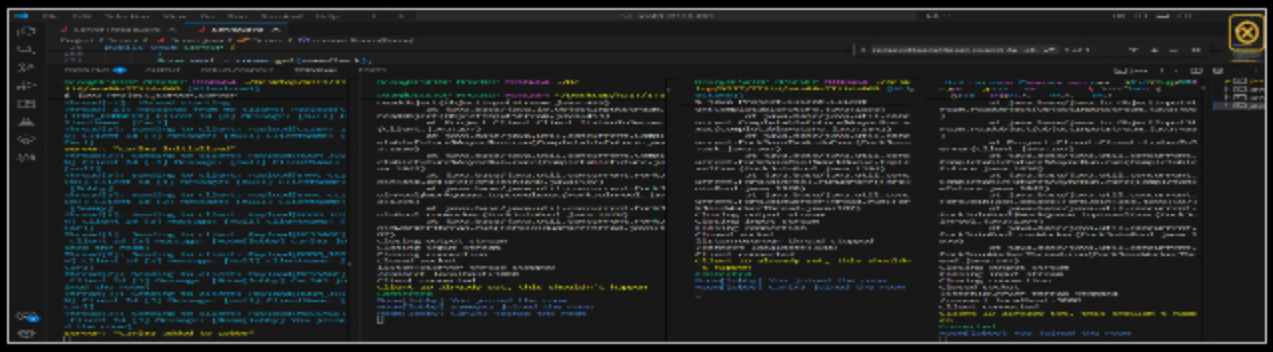
- Show examples of clients disconnecting (server should still be active)
- Show examples of server disconnecting (clients should be active but disconnected)
- Show examples of clients reconnecting when a server is brought back online
- Examples should include relevant messages of the actions occuring
- Show the relevant snippets of code that handle the client-side disconnection process
- Show the relevant snippets of code that handle the server-side termination process



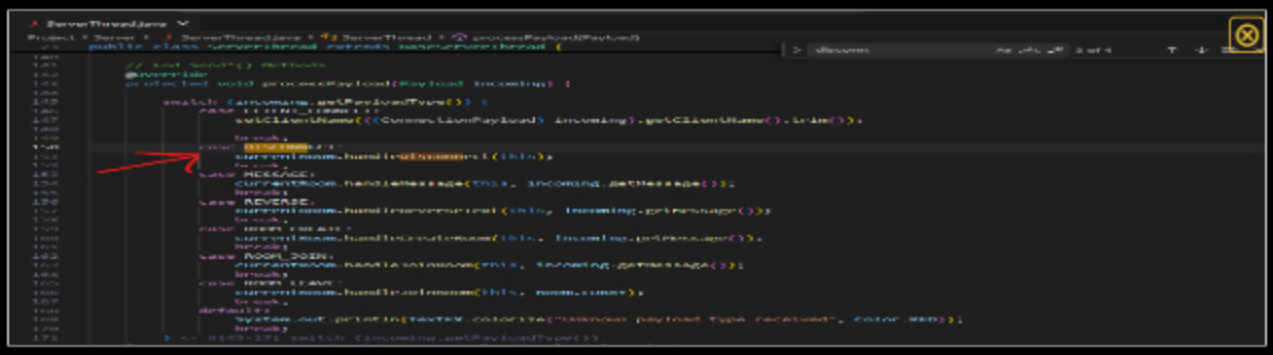
Show examples of clients disconnecting (server should still be active)



Show examples of server disconnecting (clients should be active but disconnected)



Show examples of clients reconnecting when a server is brought back online



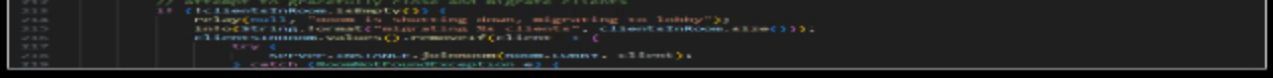
Show the relevant snippets of code that handle the client-side disconnection process

Show the relevant snippets of code that handle the client-side disconnection process

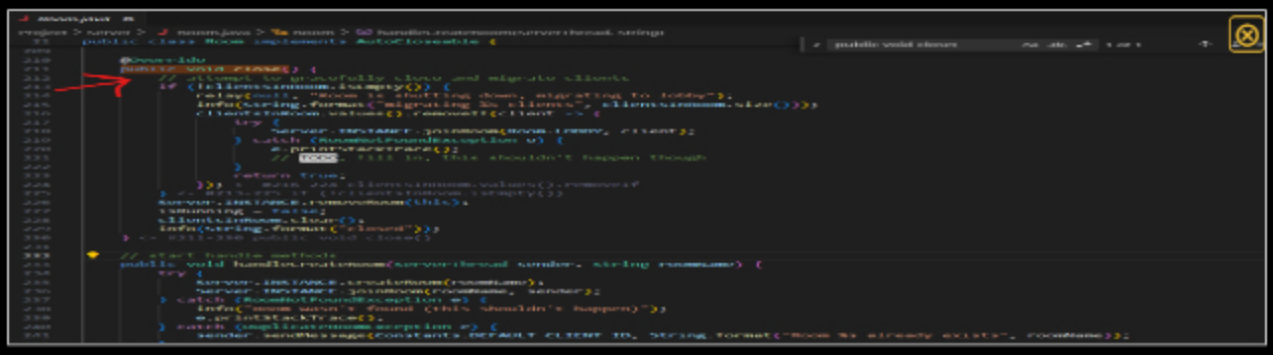
Show the relevant snippets of code that handle the client-side disconnection process

Show the relevant snippets of code that handle the client-side disconnection process

Show the relevant snippets of code that handle the server-side termination process



Show the relevant snippets of code that handle the server-side termination process



Show the relevant snippets of code that handle the server-side termination process

Saved: 11/8/2025 6:24:06 PM

Part 2:

Progress: 100%

Details:

- Briefly explain how both client and server gracefully handle their disconnect/termination logic

Your Response:

On the client side, when a user issues a disconnect command, like /disconnect or /quit, or the client application is closing, the Client class sets isRunning to false and calls close() or closeServerConnection(). This closes the socket and both the input and output streams to the server, meaning no further data is sent or recieved. Any listening threads, such as listenToInput() and listenToServer(), naturally exit their loops because isRunning is false or the streams are closed. On the server side, when a client disconnects, either intentionally or due to a network issue, the ServerThread recieves a DISCONNECT payload and calls currentRoom.handleDisconnect(), which removes the client from the room and updates the server's room membership structures. The server also takes the client out of its list of active users to free up memory and resources. If the server itself is shutting down, a special cleanup process automatically runs to safely disconnect all connected clients and clear out all rooms using disconnectAll() before the server stops.

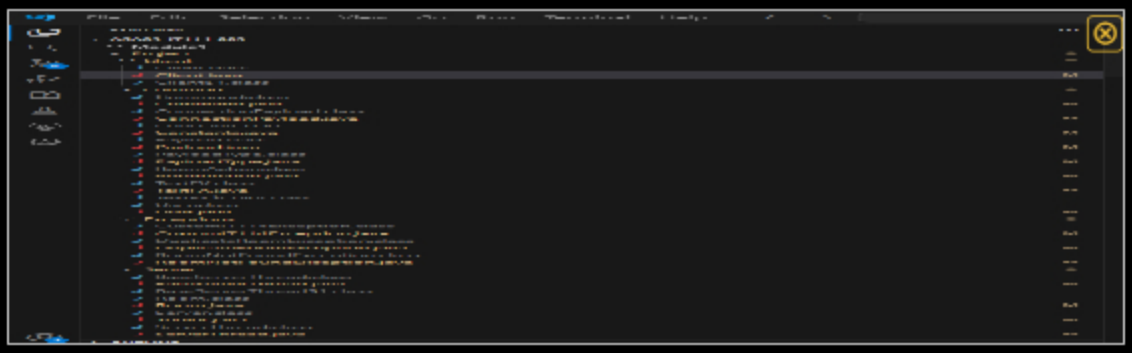
Saved: 11/8/2025 6:24:06 PM

Section #8: (1 pt.) Misc

Progress: 100%

Task #1 (0.25 pts.) - Show the proper workspace structure with the new Client, Common, Server, and Exceptions packages

Progress: 100%



Show the proper workspace structure with the new Client, Common, Server, and Exceptions packages

Saved: 11/6/2025 2:55:09 PM

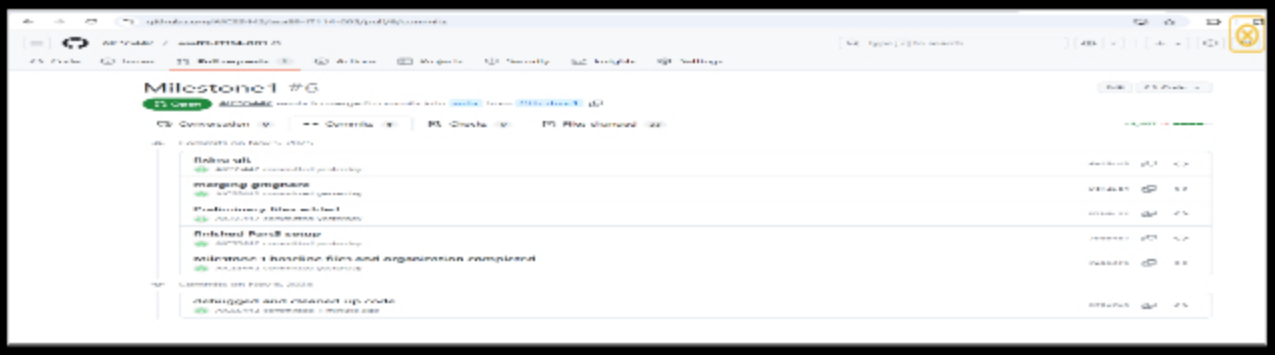
Task #2 (0.25 pts.) - Github Details

Progress: 100%

Part 1:

Progress: 100%

Details:
From the Commits tab of the Pull Request screenshot the commit history



From the Commits tab of the Pull Request screenshot the commit history

Saved: 11/6/2025 2:58:39 PM

Part 2:

Progress: 100%

Details:
Include the link to the Pull Request (should end in `/pull/#`)

URL #1
https://github.com/AICSS442/asa89-

URL
https://github.com/AICSS442/asa

Saved: 11/6/2025 2:58:39 PM

Task #3 (0.25 pts.) - WakaTime - Activity

Progress: 100%

Details:

- Visit the WakaTime.com Dashboard
- Click `Projects` and find your repository
- Capture the overall time at the top that includes the repository name
- Capture the individual time at the bottom that includes the file time
- Note: The duration isn't relevant for the grade and the visual graphs aren't necessary

wakatime

wakatime

Saved: 11/6/2025 3:00:29 PM

Task #4 (0.25 pts.) - Reflection

Progress: 100%

Task #1 (0.33 pts.) - What did you learn?

Progress: 100%

Details:
Briefly answer the question (at least a few decent sentences)

Your Response:

I learned how a server can handle multiple clients at the same time using multithreading. By creating a seperate thread for each connected client, the server can listen for new connections while also managing messages and commands from existing clients. This means that clients can interact with the server simultaneously without blocking each other.

Saved: 11/8/2025 8:22:50 AM

Task #2 (0.33 pts.) - What was the easiest part of the assignment?

Progress: 100%

Details:
Briefly answer the question (at least a few decent sentences)

Your Response:

I think the most straightforward part of this was understanding the logic behind sending and recieving simple messages between the client and server. Once the basic socket connection was established, creating a Payload for a message and printing it on the other side is straightforward compared to managing rooms, handling commands, or making sure there was proper multithreading and disconnect logic. Basically; the client sends, the server recieves, the server relays, the client displays.

Saved: 11/8/2025 8:26:06 AM

Task #3 (0.33 pts.) - What was the hardest part of the assignment?

Progress: 100%

Details:
Briefly answer the question (at least a few decent sentences)

Your Response:

The hardest part was managing rooms and moving clients between them while making sure everything runs smoothl with multiple threads. For example, when a client sends a createRoom or /joinRoom command, the Client class sends a

client sends a /createRoom or /joinRoom command, the Client class sends a Payload to the server, and the ServerThread calls methods like currentRoom.handleCreateRoom() or currentRoom.handleJoinRoom(). These methods update the Server's rooms map and add or remove clients from rooms. Since many clients can be connected at the same time, the server uses a ConcurrentHashMap to safely manage rooms without crashing when multiple threads try to change the data at once.

Saved: 11/8/2025 8:31:09 AM