Submission Worksheet

Submission Data

Course: IT114-450-M2025
Assignment: IT114 Milestone 1
Student: Lamia M. (lm87)
Status: Submitted | **Worksheet Progress:** 100%
Potential Grade: 10.00/10.00 (100.00%)
Received Grade: 0.00/10.00 (0.00%)
Started: 8/9/2025 4:43:16 AM
Updated: 8/9/2025 10:36:38 AM
Grading Link: https://learn.ethereallab.app/assignment/v3/IT114-450-M2025/it114-milestone-1/grading/lm87
View Link: https://learn.ethereallab.app/assignment/v3/IT114-450-M2025/it114-milestone-1/view/lm87

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

```
// UCID: LM87 | Date: 2025-08-09
// Summary: Binds to the provided port and blocks on accept() in a loop to wait for incoming clients.

try (ServerSocket serverSocket = new ServerSocket(port)) {
    createRoom(Room.LOBBY);// create the first room (lobby)
    while (isRunning) {
        info(message:"Waiting for next client");
        Socket incomingClient = serverSocket.accept(); // blocking action, waits for a client connection
        info(message:"Client connected");
        // wrap socket in a ServerThread, pass a callback to notify the Server when
        // they're initialized
        ServerThread serverThread = new ServerThread(incomingClient, this::onServerThreadInitialized);
        // start the thread (typically an external entity manages the lifecycle and we
        // don't have the thread start itself)
        serverThread.start();
        // Note: We don't yet add the ServerThread reference to our connectedClients map
    }
} catch (DuplicateRoomException e) {
```

Started server from the terminal; now listening on port 3000 and waiting for clients.

```
// UCID: LM87 | Date: 2025-08-09
// Summary: Binds to the provided port and blocks on accept() in a loop to wait for incoming clients.

try (ServerSocket serverSocket = new ServerSocket(port)) {
    createRoom(Room.LOBBY);// create the first room (lobby)
    while (isRunning) {
        info(message:"Waiting for next client");
        Socket incomingClient = serverSocket.accept(); // blocking action, waits for a client connection
        info(message:"Client connected");
        // wrap socket in a ServerThread, pass a callback to notify the Server when
        // they're initialized
        ServerThread serverThread = new ServerThread(incomingClient, this::onServerThreadInitialized);
        // start the thread (typically an external entity manages the lifecycle and we
        // don't have the thread start itself)
        serverThread.start();
        // Note: We don't yet add the ServerThread reference to our connectedClients map
    }
} catch (DuplicateRoomException e) {
```

Server.java binds to the given port and uses a blocking serverSocket.accept() loop to wait for incoming connections.

Saved: 8/9/2025 5:32:05 AM

Part 2:

Progress: 100%

Details:

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

Your Response:

`main` parses the port and calls `start(port)`. Inside `start`, the server creates a `ServerSocket` bound to the port and logs "Listening on port…". It then creates the **Lobby** room and enters a loop where it prints "Waiting for next client" and **blocks** on `serverSocket.accept()`. When a client connects, the socket is wrapped in a `ServerThread` (constructed with a callback), and that thread is started to handle the client I/O. In the callback `onServerThreadInitialized`, the server assigns a unique client ID and immediately adds the client to the Lobby via `joinRoom`. Rooms are tracked in a thread-safe `ConcurrentHashMap`, and a JVM shutdown hook calls `shutdown()` to disconnect everyone cleanly.

```
// ... inside start(int port)
try (ServerSocket serverSocket = new ServerSocket(port)) {
    createRoom(Room.LOBBY);
    while (isRunning) {
        info("Waiting for next client");
        Socket incomingClient = serverSocket.accept(); // blocking wait
        info("Client connected");
        ServerThread serverThread = new ServerThread(incomingClient, this::onSer
        serverThread.start();
    }
}
```

bind → wait (`accept()`) → spawn `ServerThread` → init → join Lobby.

Saved: 8/9/2025 5:32:05 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

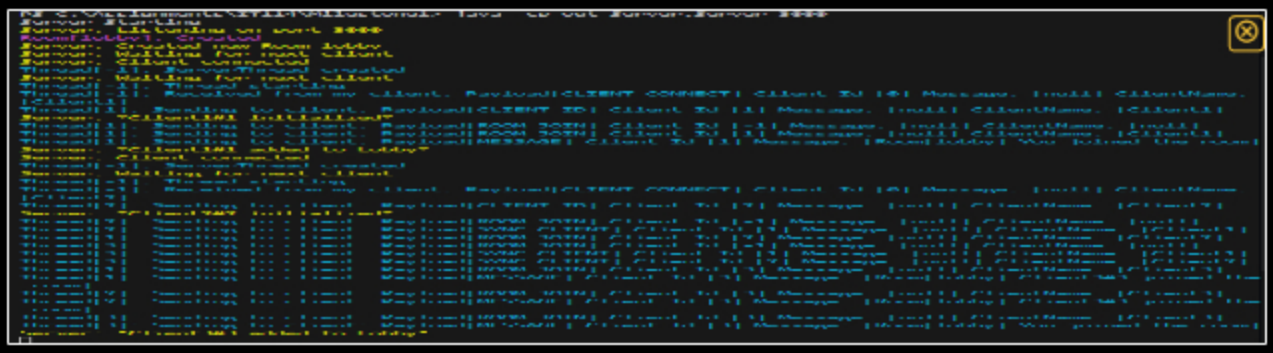
Progress: 100%

Part 1:

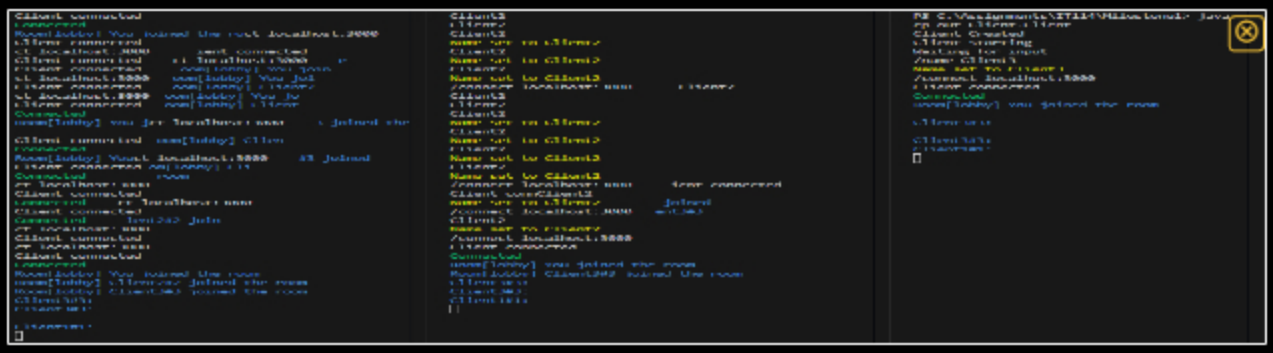
Progress: 100%

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



Server received 3 client connections; each handled by its own ServerThread.



Three clients (Client1, Client2, Client3) connected concurrently.

```java
// UCID: LM87 | Date: 2025 08 09
// Summary: Room maintains clients and relays messages to all members.
    protected void joinRoom(String name, ServerThread client) throws RoomNotFoundException {
        final String nameCheck = name.toLowerCase();
        if (!rooms.containsKey(nameCheck)) {
            throw new RoomNotFoundException(String.format(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);
    }
```

Per-client thread: processes payloads and delegates to room handlers

Saved: 8/9/2025 5:43:08 AM

Part 2:

Progress: 100%

Details:

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

Your Response:

The server binds a `ServerSocket` and loops on `accept()`. For each new socket it creates and `start()` a `ServerThread`, so every client runs on its own thread. When a thread initializes, `onServerThreadInitialized` assigns a unique ID and joins the client to the Lobby. Rooms are tracked in a `ConcurrentHashMap`, and each `Room` keeps a list of its clients and relays messages to members only. Because work happens per thread, multiple clients can connect, join rooms, and chat concurrently without blocking each other.

Saved: 8/9/2025 5:43:08 AM

Section #3: (2 pts.) Feature: Server Will Implement The Concept Of Rooms (With The Default Being "l obby")

Progress: 100%

Task #1 (2 pts.) - Evidence

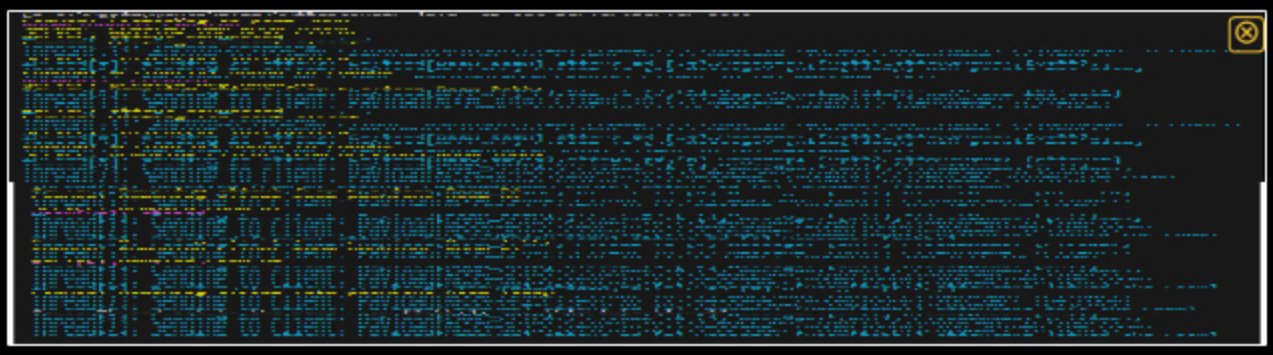
Progress: 100%

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)



Terminal output of rooms being created, joined and removed on server side.

```
    // UCID: LM87 | 2025-08-09
// Summary: Adds a new Room to the rooms map.
    protected void createRoom(String name) throws DuplicateRoomException {…

    /**…


// UCID: LM87 | Date: 2025 08 00
// Summary: Room maintains clients and relays messages to all members.
    protected void joinRoom(String name, ServerThread client) throws RoomNotFoundException {…


    // UCID: LM87 | 2025-08-09
// Summary: Deletes a room from the rooms map.
    protected void removeRoom(Room room) {…
```

Code snippet which does the creation, joining and removal from/in the rooms.

Saved: 8/9/2025 6:14:24 AM

Part 2:

Progress: 100%

Details:

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

Your Response:

When I make a room (e.g., /createroom R1), the server just checks if "R1" already exists in its rooms map. If not, it spins up a new Room object, stores it, and logs that it was created. Joining is basically "move me from my current room to that one": the server removes me from wherever I am (usually Lobby), adds me to R1, and sends those "left/joined the room" messages so everyone sees what happened. Leaving is the same idea—either /leaveroom or "join Lobby"—the server moves me back to Lobby and announces it. If a room ends up empty (and it's not Lobby), the server cleans it up by removing it from the rooms list. Lobby is created once at startup so there's always a default place to land.

Saved: 8/9/2025 6:14:24 AM

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

Progress: 100%

Task #1 (1 pt.) - Evidence

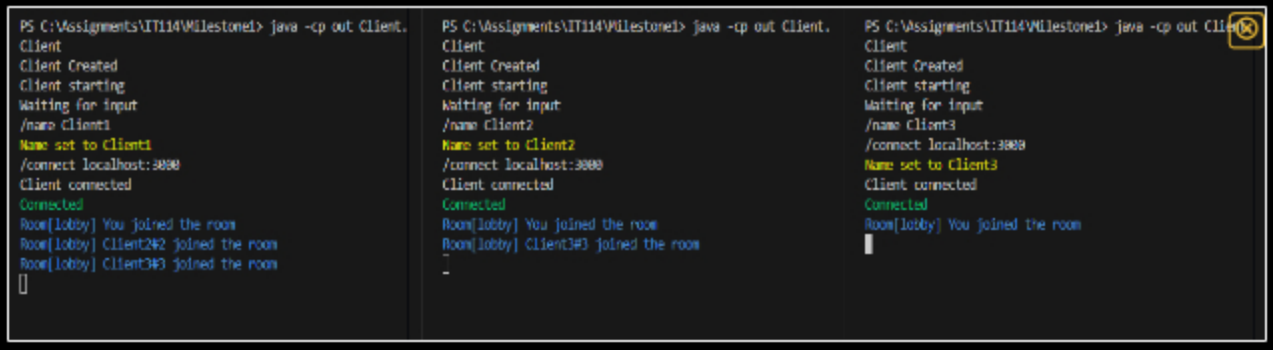
Progress: 100%

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



All 3 clients connected succesfully.

```
// UCID: LM87 | 2025-08-09
// Summary: Handles /name <value> and stores it before connecting.
else if (text.startsWith(Command.NAME.command)) {
    text = text.replace(Command.NAME.command, replacement:"").trim();
    if (text == null || text.length() == 0) {
        System.out.println(TextFX.colorize(text:"This command requires a name as an argument", Color.RED));
        return true;
    }
    myUser.setClientName(text);// temporary until we get a response from the server
    System.out.println(TextFX.colorize(String.format(format:"Name set to %s", myUser.getClientName()),
            Color.YELLOW));
    wasCommand = true;
} else if (text.equalsIgnoreCase(Command.LIST_USERS.command)) {
    System.out.println(TextFX.colorize(text:"Known clients:", Color.CYAN));
    knownClients.forEach((key, value) -> {
        System.out.println(TextFX.colorize(String.format(format:"%s%s", value.getDisplayName(),
                key == myUser.getClientId() ? " (you)" : ""), Color.CYAN));
```

Client: /name command sets desired name.

```
// UCID: LM87 | 2025-08-09
// Summary: Parses /connect host:port and opens the socket; then sends name.
if (isConnection("/" + text)) {
    if (myUser.getClientName() == null || myUser.getClientName().isEmpty()) {
        System.out.println(
                TextFX.colorize(text:"Please set your name via /name <name> before connecting", Color.RED));
        return true;
    }
    // replaces multiple spaces with a single space
    // splits on the space after connect (gives us host and port)
    // splits on : to get host as index 0 and port as index 1
    String[] parts = text.trim().replaceAll(regex:" +", replacement:" ").split(regex:" ")[1].split(regex:":");
    connect(parts[0].trim(), Integer.parseInt(parts[1].trim()));
    sendClientName(myUser.getClientName());// sync follow-up data (handshake)
    wasCommand = true;
}
```

Client: /connect parsed; connect() then sendClientName().

```
// UCID: LM87 | 2025-08-09
// Summary: Allows /connect localhost:port or IP:port.
final Pattern ipAddressPattern = Pattern
        .compile(regex:"/connect\\s+(\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}:\\d{3,5})");
final Pattern localhostPattern = Pattern.compile(regex:"/connect\\s+(localhost:\\d{3,5})");
```

Client: /connect regex for localhost or IP.

Saved: 8/9/2025 6:13:42 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:

When I type /name Alice, the client's input loop notices the slash and routes it into processClientCommand(). That block strips the /, matches the name command, and just saves the value into myUser.setClientName("Alice"). No network happens yet—it's basically me telling the app "this is what I want to be called," and it prints "Name set to Alice."

Then I do /connect localhost:3000. The client checks that format against two regexes (localhost

or an IP with a port). If I forgot to set a name, it warns me. Otherwise it parses the host/port and calls connect(host, port). That opens the socket, wires up the ObjectOutputStream/ObjectInputStream, prints “Client connected,” and spins up listenToServer() on a background thread. Right after connecting, the client sends my chosen name to the server via sendClientName() as a CLIENT_CONNECT payload. The server replies with a CLIENT_ID, which my client handles in processClientData()—that sets my real ID and confirms the name. At that point the terminal prints “Connected,” and I usually see “Room[lobby] You joined the room,” meaning the handshake is complete and I’m fully in.

Saved: 8/9/2025 6:13:42 AM

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

Progress: 100%

Task #1 (2 pts.) - Evidence

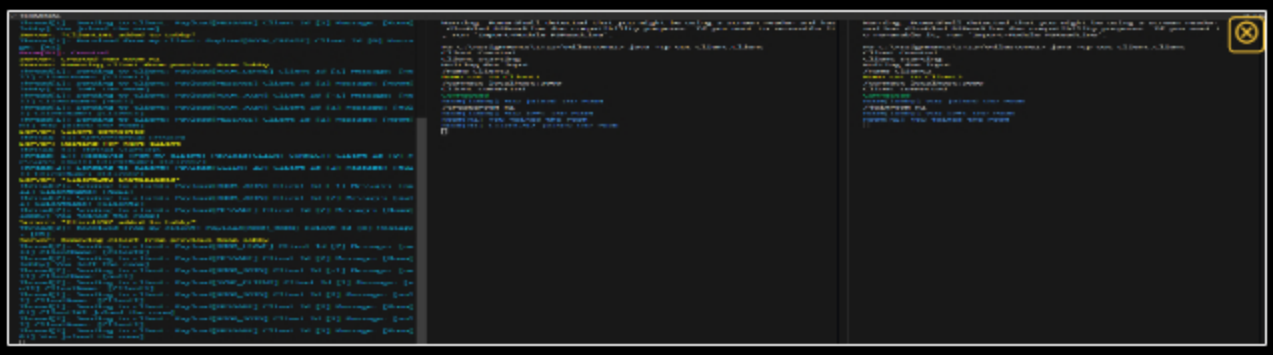
Progress: 100%

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



terminal output of /createroom and /joinroom

```
// UCID: LH07 | 2025-08-09
// Summary: Parses /createroom and /joinroom (two-word) ROOM_CREATE/ROOM_JOIN
} else if (text.startsWith(Command.CREATE_ROOM.command)) {
    text = text.replace(Command.CREATE_ROOM.command, replacement:"").trim();
    if (text == null || text.length() == 0) {
        System.out.println(TextFX.colorize(text:"This command requires a room name as an argument", Color.RED));
        return true;
    }
    sendRoomAction(text, RoomAction.CREATE);
    wasCommand = true;
} else if (text.startsWith(Command.JOIN_ROOM.command)) {
    text = text.replace(Command.JOIN_ROOM.command, replacement:"").trim();
    if (text == null || text.length() == 0) {
        System.out.println(TextFX.colorize(text:"This command requires a room name as an argument", Color.RED));
        return true;
    }
    sendRoomAction(text, RoomAction.JOIN);
    wasCommand = true;
} else if (text.startsWith(Command.LEAVE_ROOM.command) || text.startsWith(prefix:"/leave")) {
    // Note: Accounts for /leave and /leaveroom variants (or anything beginning with
    // /leave)
    sendRoomAction(text, RoomAction.LEAVE);
    wasCommand = true;
```

Client: parse room cmds and dispatch sendRoomAction().

```
    // UCID: LM87 | 2025-08-09
// Summary: Builds Payload with room name and ROOM_CREATE/ROOM_JOIN type.
    private void sendRoomAction(String roomName, RoomAction roomAction) throws IOException {
        Payload payload = new Payload();
        payload.setMessage(roomName);
        switch (roomAction) {
            case CREATE:
                payload.setPayloadType(PayloadType.ROOM_CREATE);
                break;
            case JOIN:
                payload.setPayloadType(PayloadType.ROOM_JOIN);
                break;
            case LEAVE:
                payload.setPayloadType(PayloadType.ROOM_LEAVE);
                break;
            default:
                System.out.println(TextFX.colorize(text:"Invalid room action", Color.RED));
                break;
        }
        sendToServer(payload);
    }
```

Client: send ROOM_CREATE/ROOM_JOIN to server.

```
    // UCID: LM97 | 2025-08-09
    // Summary: Prints join message and syncs known clients on ROOM_JOIN/SYNC_CLIENT.
    private void processRoomAction(Payload payload) {
        if (!(payload instanceof ConnectionPayload)) {
            error(message:"Invalid payload subtype for processRoomAction");
            return;
        }
        ConnectionPayload connectionPayload = (ConnectionPayload) payload;
        // use DEFAULT_CLIENT_ID to clear knownClients (mostly for disconnect and room
        // transition)
        if (connectionPayload.getClientId() == Constants.DEFAULT_CLIENT_ID) {
            knownClients.clear();
            return;
        }
        switch (connectionPayload.getPayloadType()) {

            case ROOM_LEAVE:
                // remove from map
                if (knownClients.containsKey(connectionPayload.getClientId())) {
                    knownClients.remove(connectionPayload.getClientId());
                }
                if (connectionPayload.getMessage() != null) {
                    System.out.println(TextFX.colorize(connectionPayload.getMessage(), Color.YELLOW));
                }

                break;
            case ROOM_JOIN:
                if (connectionPayload.getMessage() != null) {
                    System.out.println(TextFX.colorize(connectionPayload.getMessage(), Color.GREEN));
```

Client: shows "You joined the room" and updates roster.

Saved: 8/9/2025 6:28:54 AM

Part 2:

Progress: 100%

Details:

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

Your Response:

I type /createroom R1.

- The input loop sends this to processClientCommand(), which strips the "/" and matches Command.CREATE_ROOM ("createroom").
- It pulls out R1 and calls sendRoomAction("R1", RoomAction.CREATE).
- sendRoomAction builds a Payload with message="R1" and payloadType=ROOM_CREATE, then sendToServer(...).
- Server side: it receives ROOM_CREATE, runs createRoom("R1"), and (in this build) immediately moves me from my current room (Lobby) into R1.
- Server broadcasts join/leave updates; client's listenToServer() → processPayload() → processRoomAction() prints "Room[R1] You joined the room" and updates the roster.

I type /joinroom R1.

- processClientCommand() matches Command.JOIN_ROOM ("joinroom"), extracts the name, and calls sendRoomAction("R1", RoomAction.JOIN).
- sendRoomAction sets payloadType=ROOM_JOIN and sends it.
- Server side: joinRoom("R1", client) removes me from my current room (if any), adds me to R1, and sends:
  - a ROOM_LEAVE notice to the old room,
  - a ROOM_JOIN (and SYNC_CLIENT) to R1.
- Back on the client, processRoomAction() shows "You joined the room" in green and syncs the user list so I can see who's there.

Saved: 8/9/2025 6:28:54 AM

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

Progress: 100%

--- Section Collapsed ---

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

Progress: 100%

Task #1 (1 pt.) - Evidence

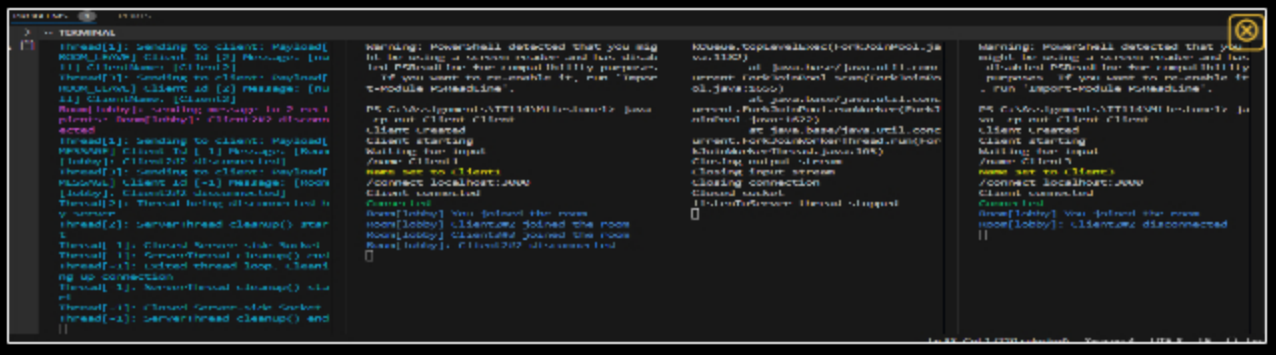
Progress: 100%

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



Client2 issues /disconnect; remaining clients receive leave/disconnect notices; server continues running.

Server terminated; clients remain running and report dropped connection without crashing.

After server restart, Client1/2/3 reconnect via /connect and receive normal join messages.

code snippet for disconnect

code snippet for closing server connection.

code snippet for shutting down server

Saved: 8/9/2025 9:53:51 AM

Part 2:

Progress: 100%

Details:

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

Your Response:

On the client side, if you type /disconnect, the client politely tells the server, "I'm leaving." The server then confirms the disconnect, and your client clears its user list, resets your name/id, and shows a "You disconnected" message. If the server vanishes unexpectedly (like someone pulled the plug), the client notices the connection drop, prints "Connection dropped," and shuts down its connection cleanly — but it keeps running so you can reconnect without restarting the program.

On the server side, if it gets a disconnect request from a client, it removes that person from the room, tells everyone else "so-and-so left/disconnected," and closes that client's socket. If the whole server shuts down, it runs a shutdown routine that goes through every room, tells all the clients the server is going away, and closes all their connections.

Saved: 8/9/2025 9:53:51 AM

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%

Terminal output showing package structure for Client, Common, Server, and Exceptions with corresponding Java files.

Saved: 8/9/2025 9:57:19 AM

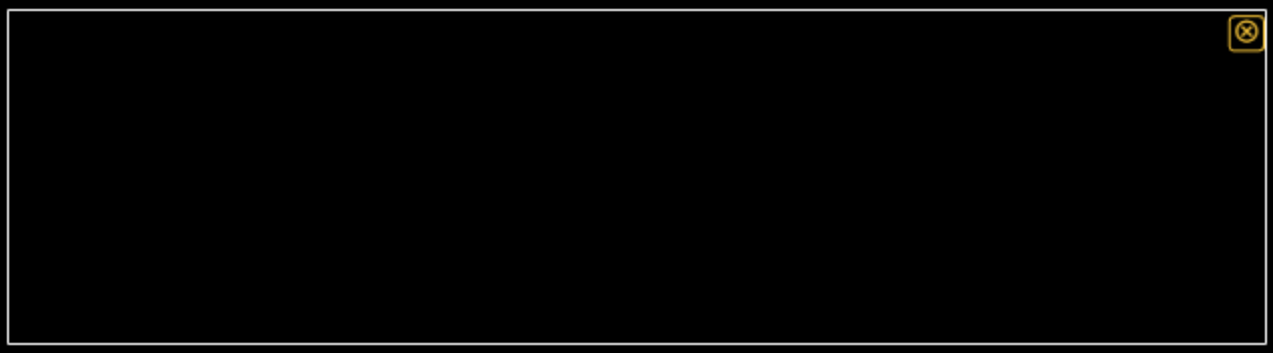
Task #2 (0.25 pts.) - Github Details

Progress: 100%

Part 1:

Progress: 100%

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



screenshot of commit changes

Saved: 8/9/2025 10:33:31 AM

Part 2:

Progress: 100%

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

URL #1
https://github.com/Lmostefaoui8/lm87-it114/pull/1/

URL
https://github.com/Lmostefaoui8/

Saved: 8/9/2025 10:33:31 AM

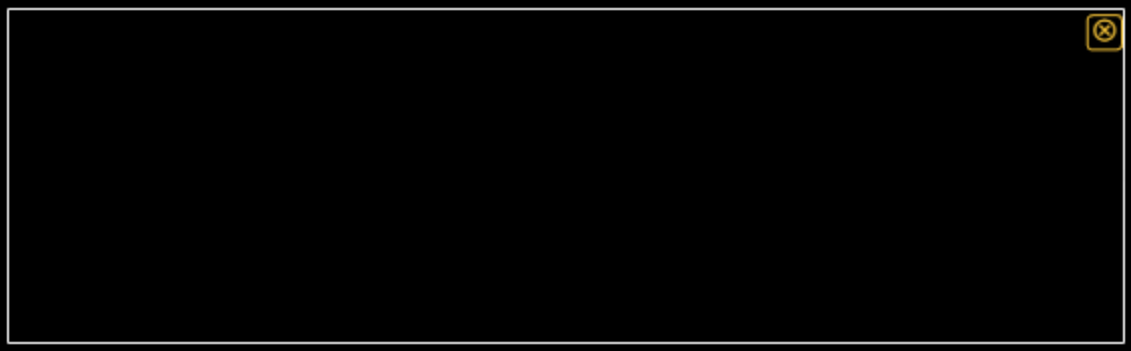
Task #3 (0.25 pts.) - WakaTime - Activity

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



i didn't set up this repo on wakatime.com

Saved: 8/9/2025 10:35:11 AM

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 to manage a client-server application where multiple players can interact in separate rooms. Specifically, I saw how to handle connections with ServerSocket.accept() and spin up threads so each client can send and receive data without blocking others. I also learned how to pass structured messages using Payload objects, which keeps the logic clean and consistent. The way the server routes messages only to clients in the same room really helped me understand scoped communication in multiplayer games.

Saved: 8/9/2025 10:36:03 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:

The easiest part was starting the server and connecting multiple clients. Once the commands were in place, running /name and /connect felt very straightforward. Creating and joining rooms was also easy because the framework already handled most of the logic—I just had to follow the steps and confirm the messages appeared as expected. It was satisfying to see the join/leave messages pop up exactly as described in the instructions.

Saved: 8/9/2025 10:36:13 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 making sure the workspace structure and Git workflow matched the requirements. Moving files into Client, Common, Server, and Exceptions packages without breaking imports took extra attention. I also found it tricky to remember all the submission steps—especially creating the Pull Request, adding the worksheet PDF to GitHub, and ensuring the history was correct. Once I got through that process, though, it made sense how each step fit into the bigger picture.

Saved: 8/9/2025 10:36:38 AM