Submission Worksheet

Submission Data

Course: IT114-450-M2025
Assignment: IT114 Milestone 1
Student: Alvina A. (aa3375)
Status: Graded | **Worksheet Progress:** 81%
Potential Grade: 9.35/10.00 (93.50%)
Received Grade: 9.25/10.00 (92.50%)
Started: 7/26/2025 1:23:36 AM
Updated: 7/26/2025 1:23:36 AM
Grading Link: https://learn.ethereallab.app/assignment/v3/IT114-450-M2025/it114-milestone-1/grading/aa3375
View Link: https://learn.ethereallab.app/assignment/v3/IT114-450-M2025/it114-milestone-1/view/aa3375

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

```
PS C:\It1114\IT114-2025-Module3-Homework\Projects> javac
Server\Server.java
PS C:\It1114\IT114-2025-Module3-Homework\Projects> java S
erver.Server 3000
Server Starting
Server: Listening on port 3000
Room[lobby]: Created
Server: Created new Room lobby
Server: Waiting for next client
```

Terminal output confirming the server was successfully started and is now listening for incoming client connections on port 3000

```
private void start(int port) {
    this.port = port;
    // server listening
    info("Listening on port " + this.port);
    // Simplified client connection loop
    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) {
        System.err.println(TextFX.colorize(text:"Lobby already exists (this shouldn't happen)", Color.RED));
    } catch (IOException e) {
```

Java code snippet from Server.java that initializes the server socket, listens for incoming connections

Saved: 7/25/2025 10:34:51 PM

Part 2:

Progress: 100%

Details:

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

Your Response:

On the server side, a ServerSocket is created and bound to a port (e.g., 3000). The server enters an infinite loop, blocking on serverSocket.accept(), which waits for a client to connect. Once a client connects, a new Socket object is returned, and a message is printed confirming the connection. This setup allows the server to continuously listen and respond to multiple client connection requests.

Saved: 7/25/2025 10:34:51 PM

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

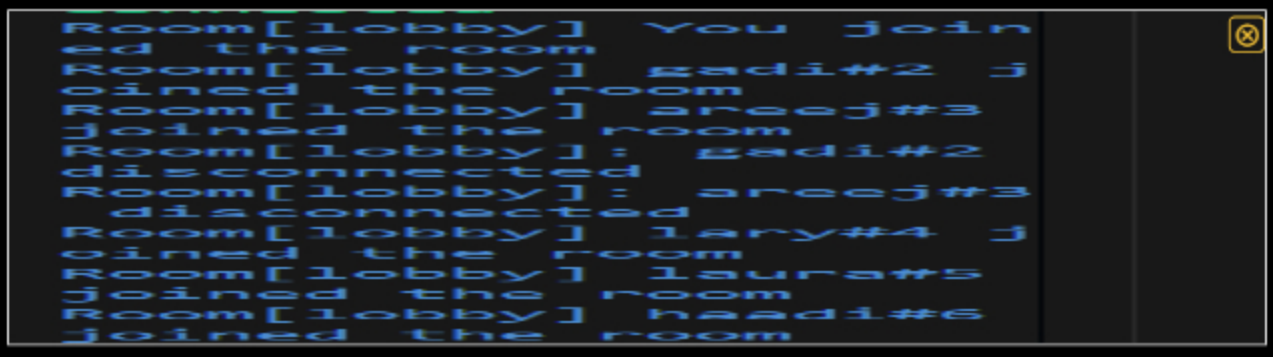
Progress: 100%

Part 1:

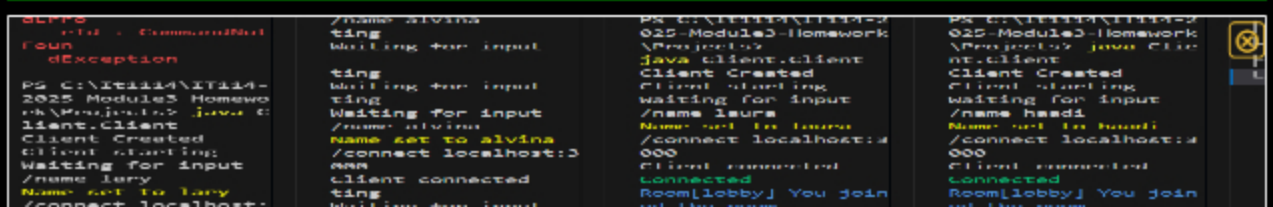
Progress: 100%

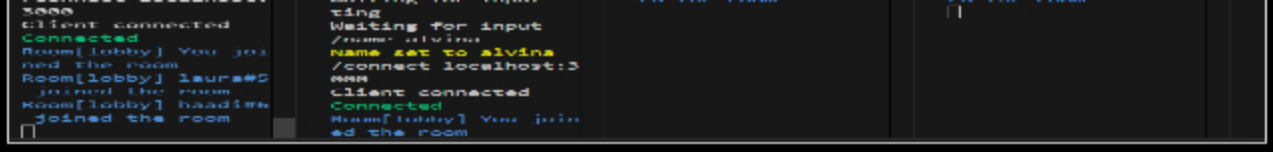
Details:

- 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



This is the clip of server receiving multiple connections.





4 clients are connect to the server.

```java
private void start(int port) {
    this.port = port;       You, yesterday • Project
    // server listening
    info("Listening on port " + this.port);
    // Simplified client connection loop
    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) {
        System.err.println(TextFX.colorize(text:"Lobby already exists (this shouldn't happen)", Color.RED));
```

code that handles multiple connections

Saved: 7/25/2025 10:36:24 PM

Part 2:

Progress: 100%

Details:

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

Your Response:

The server listens on the specified port by creating a ServerSocket. Inside a loop controlled by isRunning, it waits (blocks) for client connections using serverSocket.accept(). When a client connects, the server wraps the client socket in a ServerThread object, which handles communication with that client. Each ServerThread runs in its own thread (started with serverThread.start()), allowing the server to handle multiple clients concurrently without blocking. This threaded design enables the server to keep accepting new clients while managing all connected clients independently.

Saved: 7/25/2025 10:36:24 PM

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)

```
Client connected
Connected
Room[lobby] You joined the room
/createroom Lobby
Room Lobby already exists
/joinroom Lobby
Room[lobby] You left the room
Room[lobby] You joined the room
/leaveroom Lobby
Room[lobby] You left the room
```

Server terminal shows room being created, joining and leaving room.

```
protected void createRoom(String name) throws DuplicateRoomException {
    final String nameCheck = name.toLowerCase();
    if (rooms.containsKey(nameCheck)) {
        throw new DuplicateRoomException(String.format(format:"Room %s already exists", name));
    }
    Room room = new Room(name);
    rooms.put(nameCheck, room);
    info(String.format(format:"Created new Room %s", name));
}
```

This is the code of room being created on server side

```
 */
protected void joinRoom(String name, ServerThread client) throws RoomNotFoundException {        You, 2 days
    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);
}
```

This is the code of room being join on server side and leavinf

```
protected void removeRoom(Room room) {      You, 2 days ago • Project
    rooms.remove(room.getName().toLowerCase());
    info(String.format(format:"Removed room %s", room.getName()));
}
```

This is the code of room being removed server side.

Saved: 7/26/2025 12:05:03 AM

Part 2:

Progress: 100%

Details:

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

Your Response:

When a client sends a /createroom command, the server checks if the room name already exists in the global room map. If not, it creates a new Room object, adds it to the map, and moves the client into it. For /joinroom, the server checks if the room exists; if it does, the client is removed from their current room and added to the new one. Leaving a room typically happens automatically when joining another or disconnecting.

Saved: 7/26/2025 12:05:03 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

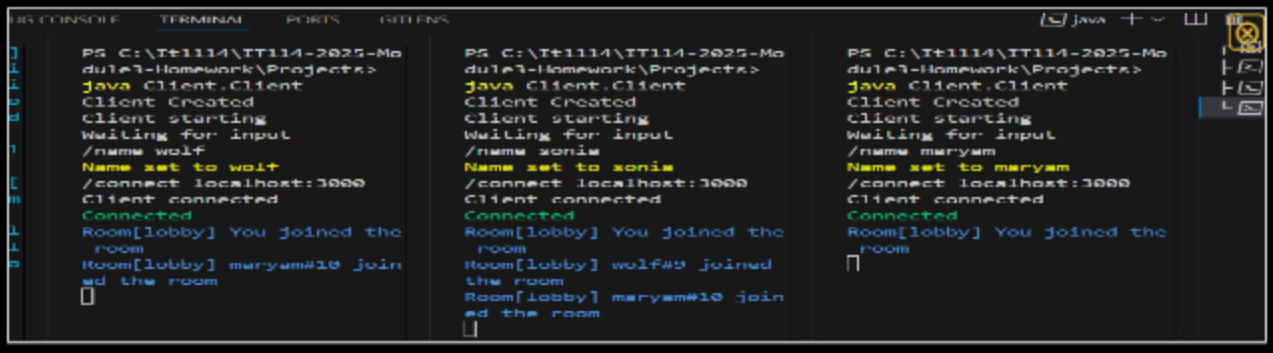
```
PS C:\It1114\IT114-2025-Module3-Homework\Projects> java Client.Client
Client Created
Client starting
Waiting for input
/name wolf
Name set to wolf
/connect localhost:3000
Client connected
```

```
PS C:\It1114\IT114-2025-Module3-Homework\Projects> java Client.Client
Client Created
Client starting
Waiting for input
/name sonia
Name set to sonia
/connect localhost:3000
Client connected
```

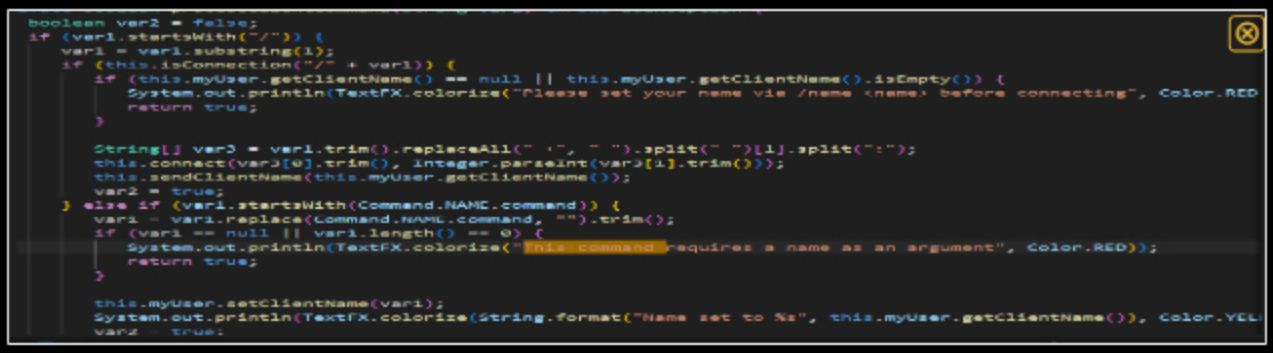
```
PS C:\It1114\IT114-2025-Module3-Homework\Projects> java Client.Client
Client Created
Client starting
Waiting for input
/name maryam
Name set to maryam
/connect localhost:3000
Client connected
```

Connected Connected Connected

This is the output of the commands /name, and /connect for each 3 clients.



This is the evidence for successful connection.



This is the process of the /name code

```
private boolean processClientCommand(String var1) throws IOException {
    boolean var2 = false;
    if (var1.startsWith("/")) {
        var1 = var1.substring(1);
        if (this.isConnection("/" + var1)) {
            if (this.myUser.getClientName() == null || this.myUser.getClientName().isEmpty()) {
                System.out.println(TextFX.colorize("Please set your name via /name <name> before connecting", Colo
                return true;
            }
```

This is the process of /connect code.

```
private boolean connect(String var1, int var2) {
    try {
        this.server = new Socket(var1, var2);
        this.out = new ObjectOutputStream(this.server.getOutputStream());
        this.in = new ObjectInputStream(this.server.getInputStream());
        System.out.println("Client connected");
        CompletableFuture.runAsync(this::listenToServer);
    } catch (UnknownHostException var4) {
        var4.printStackTrace();
    } catch (IOException var5) {
        var5.printStackTrace();
    }

    return this.isConnected();
}
```

Part 2 for the /connect

```
private void processClientData(Payload var1) {
    if (this.myUser.getClientId() != -1L) {
        System.out.println(TextFX.colorize("Client ID already set, this shouldn't happen", Color.YELLOW));
    }
```

```
    this.myUser.setClientId(var1.getClientId());
    this.myUser.setClientName(((ConnectionPayload)var1).getClientName());
    this.knownClients.put(this.myUser.getClientId(), this.myUser);
    System.out.println(TextFX.colorize("Connected", Color.GREEN));
}
```

This is code for confirmation of being fully connected/setup

Saved: 7/26/2025 12:06:47 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 the user types /name username, the client sets that name locally using myUser.setClientName(text) and prints confirmation. This name is stored but not yet sent to the server. Then, when the user types /connect host:port, the client first checks if a name has been set. If it has, the client opens a socket connection (connect()), sets up input/output streams, and starts listening for server messages via a new thread. After connection, the client immediately sends the username to the server using sendClientName() with a CLIENT_CONNECT payload. The server responds with a CLIENT_ID payload, which is handled by processClientData() to confirm the connection and store the assigned ID. At this point, the client is fully connected and ready to interact.

Saved: 7/26/2025 12:06:47 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

```
/createroom pizza
Room[lobby] You left the room
Room[pizza] You joined the room
/joinroom lobby
Room[pizza] You left the room
Room[lobby] You joined the room
```

Room creation is silent on the client by default. It shows "You joined" /joinroom is here

```
Server: Removed room pizza
Room[pizza]: closed
Thread[19]: Sending to client: Payload[ROOM_JOIN] Clie
nt Id [-1] Message: [null] ClientName: [null]
Thread[19]: Sending to client: Payload[ROOM_JOIN] Clie
nt Id [19] Message: [null] ClientName: [lily]
Thread[19]: Sending to client: Payload[MESSAGE] Client
 Id [19] Message: [Room[lobby] You joined the room]
```

```
Connected
Room[lobby] You joined the room
/createroom pizza
Room[lobby] You left the room
Room[pizza] You joined the room
/joinroom lobby
Room[pizza] You left the room
Room[lobby] You joined the room
```

This is the output of creation/join

```
        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", C
            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", C
            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;
```

This is the code for room creation and joining.

Saved: 7/26/2025 12:11:50 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:

When a client uses /createroom roomName, the client code creates a CREATE_ROOM payload with the room name and sends it to the server. The server checks if the room already exists; if not, it creates the room and adds the client to it. For /joinroom roomName, the client sends a JOIN_ROOM payload. The server checks if the room exists and moves the client from their current

room (if any) to the new one. Both commands rely on sending payloads from client to server, and server logic handles room setup or switching.

Saved: 7/26/2025 12:11:50 AM

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

Progress: 100%

Task #1 (1 pt.) - Evidence

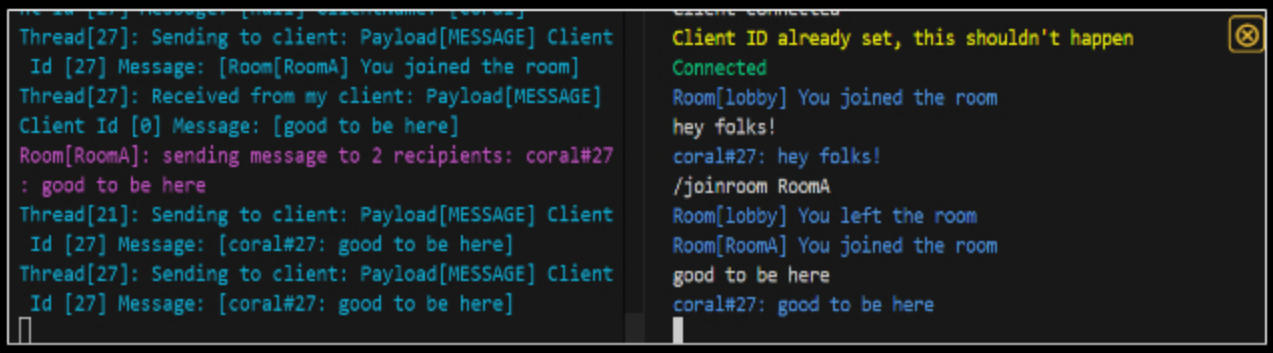
Progress: 100%

Part 1:

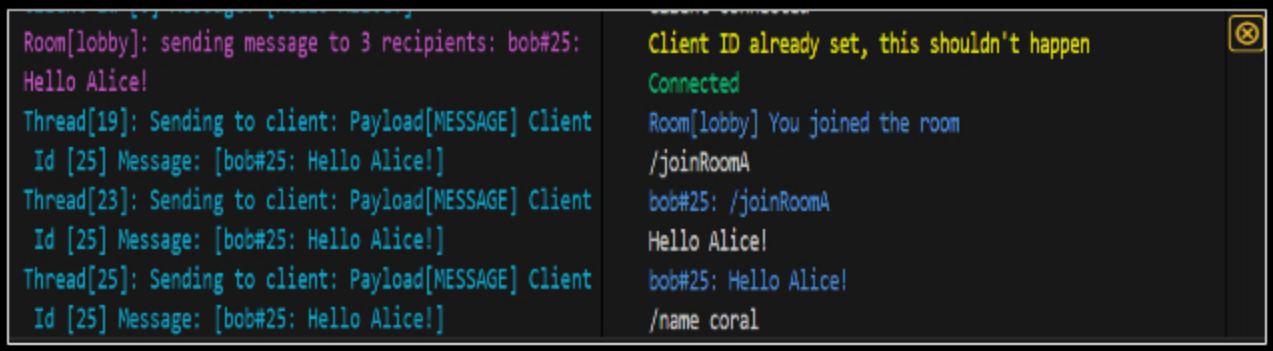
Progress: 100%

Details:

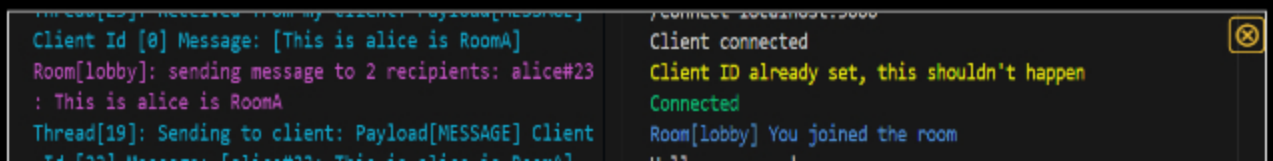
- Show the terminal output of a few messages from each of 3 clients
- Include examples of clients grouped into other rooms
- Show the relevant snippets of code that handle the message process from client to server-side and back

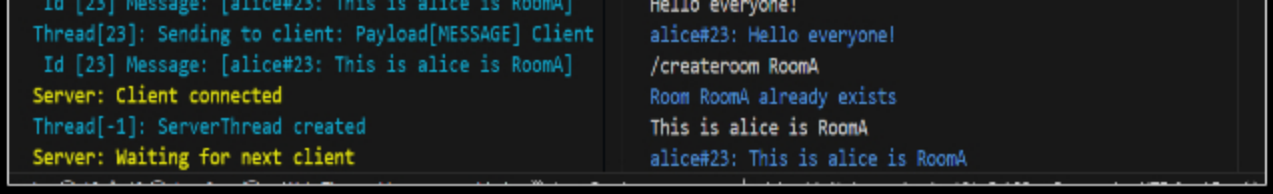


This is a few messages from client 1

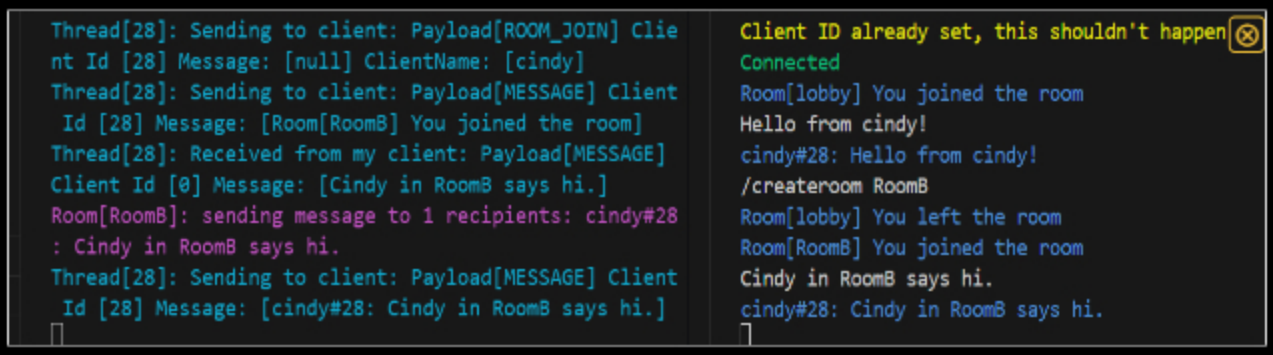


This is a few messages from client 2





This is a few messages from client 3



Clients join different rooms; messages are only received by users in the same room, ensuring room-specific chat

```java
/**
 * Sends a message to the server
 *
 * @param message
 * @throws IOException
 */
private void sendMessage(String message) throws IOException {      You, 2 days ago • Project
    Payload payload = new Payload();
    payload.setMessage(message);
    payload.setPayloadType(PayloadType.MESSAGE);
    sendToServer(payload);
}
```

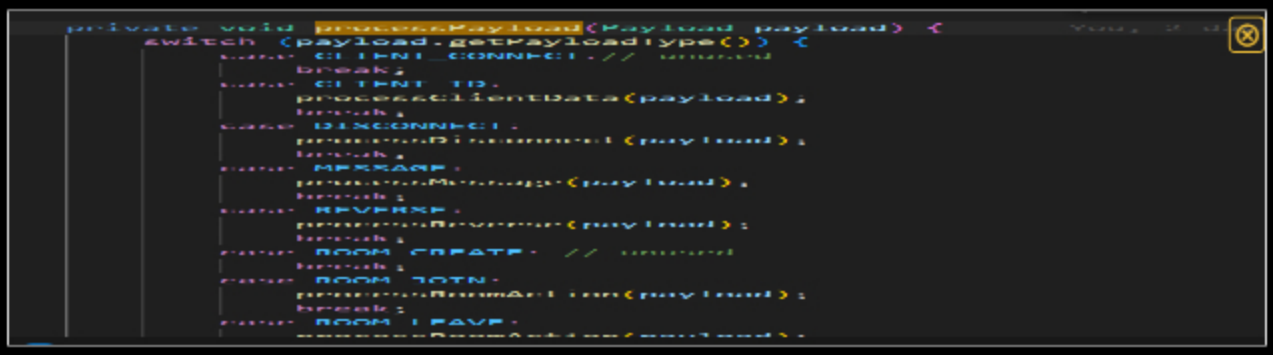
Part 1 of code that handles process from client to server

```java
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(
                x:"Not connected to server (hint: type `/connect host:port` without the quotes and replace ho
    }
}
```

Part 2 of code that handles process from client to server

```java
private void listenToServer() {      You, 2 days ago • Project
    try {
        while (isRunning && isConnected()) {
            Payload fromServer = (Payload) in.readObject(); // blocking read
            if (fromServer != null) {
                processPayload(fromServer);
            } else {
                System.out.println(x:"Server disconnected");
                break;
            }
        }
    } catch (ClassCastException | ClassNotFoundException cce) {
        System.err.println("Error reading object as specified type: " + cce.getMessage());
        cce.printStackTrace();
    } catch (IOException e) {
        if (isRunning) {
            System.out.println(x:"Connection dropped");
            e.printStackTrace();
        }
    } finally {
        closeServerConnection();
    }
```

Part 3 of code that handles process from client to server



Part 4 of code that handles process from client to server

```
private void processMessage(Payload payload) {      You, 2 days ago • Project
    System.out.println(TextFX.colorize(payload.getMessage(), Color.BLUE));
}
```

Part 5 of code that handles process from client to server

Saved: 7/26/2025 12:38:22 AM

Part 2:

Progress: 100%

Details:

- Briefly explain how the message code flow works

Your Response:

When a user types a message in the client, the client creates a Payload object containing the message and marks it as a message type. This payload is then sent through the network to the server using the client's output stream. The server receives this payload, recognizes it as a message, and forwards it only to those clients who are in the same chat room as the sender. On the receiving end, each client listens for incoming payloads from the server. When a message payload arrives, the client processes it and displays the message in the user's console. This flow ensures that messages are shared in real time but only among clients who belong to the same chat room.

Saved: 7/26/2025 12:38:22 AM

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

Progress: 100%

≔ Task #1 (1 pt.) - Evidence

Task #1 (1 pt.) - Evidence

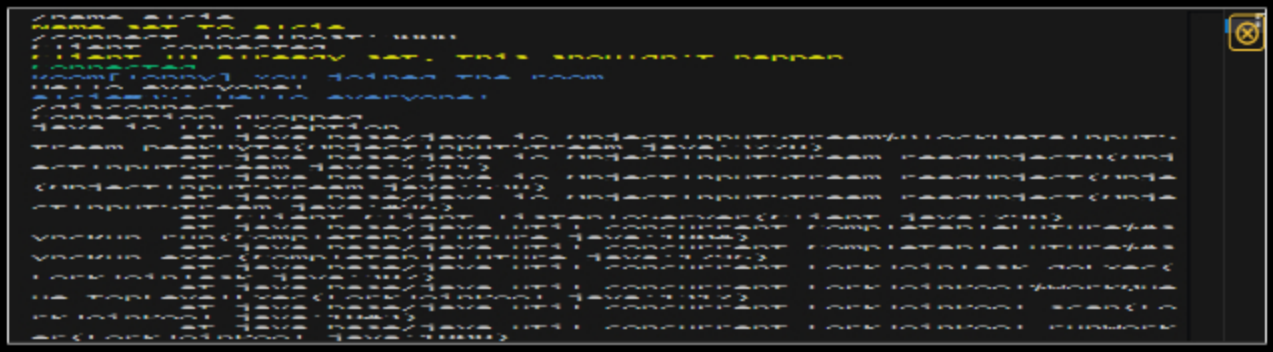
Progress: 100%

Part 1:

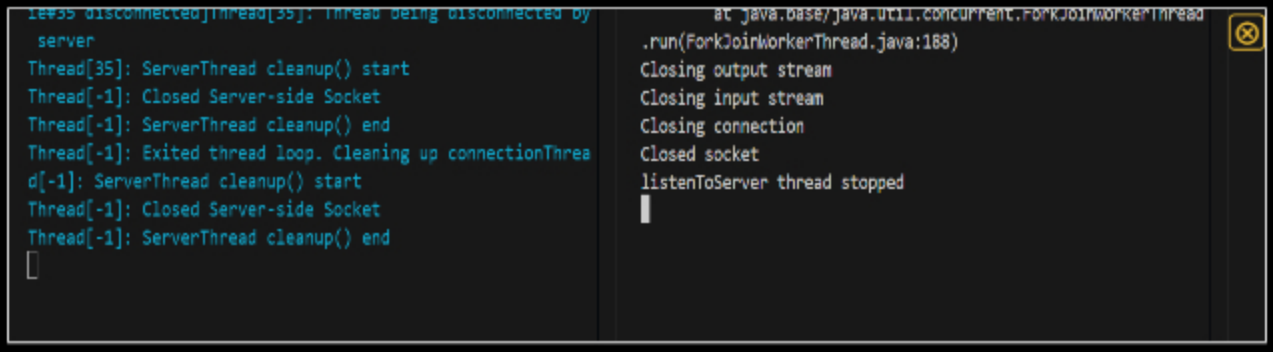
Progress: 100%

Details:

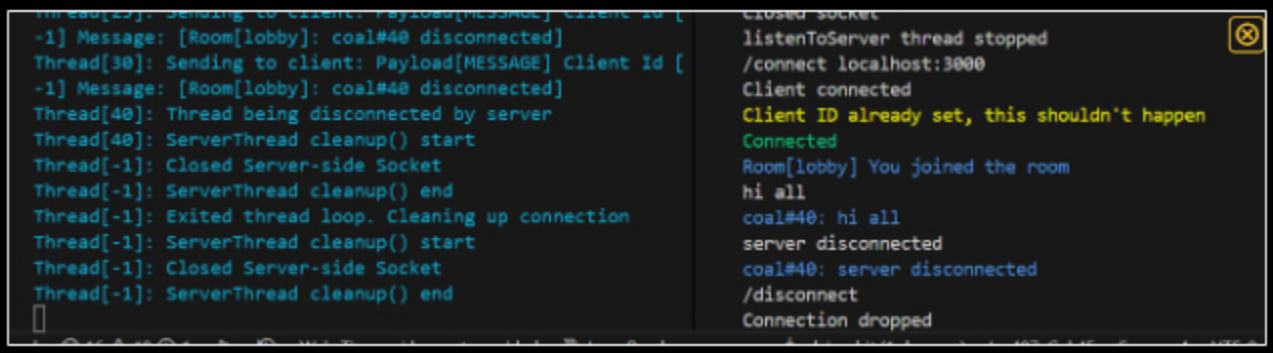
- Show examples of clients disconnecting (server should still be active)
- Show examples of server disconnecting (clients should be active but disconnected)
- Show examples of clients reconnecting when a server is brought back online
- Examples should include relevant messages of the actions occuring
- 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



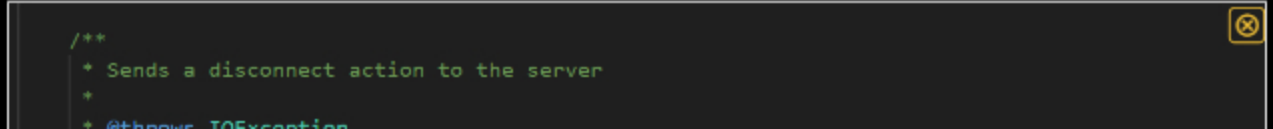
Example of client disconnection part 1



example of client disconnection part 2



Example of server disconnecting



```java
    @throws IOException
 */
private void sendDisconnect() throws IOException {      You, 2 days ago • Project
    Payload payload = new Payload();
    payload.setPayloadType(PayloadType.DISCONNECT);
    sendToServer(payload);
}
```

Part 1 of client side code

```java
 */
private boolean processClientCommand(String text) throws IOException {      You, 2 days ago • Project
    boolean wasCommand = false;
    if (text.startsWith(Constants.COMMAND_TRIGGER)) {
        text = text.substring(beginIndex:1); // remove the /
        // System.out.println("Checking command: " + text);
        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
                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(re
            connect(parts[0].trim(), Integer.parseInt(parts[1].trim()));
            sendClientName(myUser.getClientName());// sync follow-up data (handshake)
```

Part 2 of client side code

```java
 */
private void listenToServer() {      You, 2 days ago • Project
    try {
        while (isRunning && isConnected()) {
            Payload fromServer = (Payload) in.readObject(); // blocking read
            if (fromServer != null) {
                processPayload(fromServer);

            } else {
                System.out.println(x:"Server disconnected");
                break;
            }
        }
    } catch (ClassCastException | ClassNotFoundException cce) {
        System.err.println("Error reading object as specified type: " + cce.getMessage());
        cce.printStackTrace();
    } catch (IOException e) {
        if (isRunning) {
            System.out.println(x:"Connection dropped");
            e.printStackTrace();
        }
    } finally {
        closeServerConnection();
    }
    System.out.println(x:"listenToServer thread stopped");
}
```

Part 3 of client side code

```java
private void processDisconnect(Payload payload) {      You, 2 days ago • Project
    if (payload.getClientId() == myUser.getClientId()) {
        knownClients.clear();
        myUser.reset();
        System.out.println(TextFX.colorize(text:"You disconnected", Color.RED));
    } else if (knownClients.containsKey(payload.getClientId())) {
        User disconnectedUser = knownClients.remove(payload.getClientId());
        if (disconnectedUser != null) {
            System.out.println(TextFX.colorize(String.format(format:"%s disconnected", disconnectedUser.getDi
                    Color.RED));
        }
    }

}
```

Part 4 of client side code

```java
private void closeServerConnection() {
    try {
        if (out != null) {
            System.out.println(x:"Closing output stream");
            out.close();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    try {
        if (in != null) {
            System.out.println(x:"Closing input stream");
            in.close();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    try {
        if (server != null) {
            System.out.println(x:"Closing connection");
            server.close();
            System.out.println(x:"Closed socket");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Part 5 of client side code

```
Thread[25]: Client already closed
Thread[25]: ServerThread cleanup() end
Thread[38]: My Client disconnected
Thread[38]: Exited thread loop. Cleaning up connection
Thread[38]: ServerThread cleanup() start
Thread[38]: Client already closed
Thread[38]: ServerThread cleanup() end
```

```
Client connected
Client ID already set, this shouldn't happen
Connected
Room[lobby] You joined the room
Exception in thread "main" java.util.concurrent.CompletionExce
ption: java.util.NoSuchElementException: No line found
PS C:\It1114\IT114-2025-Module3-Homework\Projects>
```

client disconnection part 3

```java
private Server() {      You, 2 days ago • Project
    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        info(message:"JVM is shutting down. Perform cleanup tasks.");
        shutdown();
    }));
}
```

Server side code part 1

```java
 */
private void shutdown() {
    try {
        // chose removeIf over forEach to avoid potential
        // ConcurrentModificationException
        // since empty rooms tell the server to remove themselves
        rooms.values().removeIf(room -> {
            room.disconnectAll();
            return true;
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Server side code part 2

Saved: 7/26/2025 1:22:26 AM

Part 2:

Progress: 100%

Details:

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

Your Response:

The client politely tells the server before closing its connection and cleans up its resources, while the server ensures all clients are disconnected and resources are freed during shutdown, so both sides close their connections smoothly and safely.

Saved: 7/26/2025 1:22:26 AM

Section #8: (1 pt.) Misc

Progress: 37%

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

Progress: 0%

Missing Caption

Not saved yet

Task #2 (0.25 pts.) - Github Details

Progress: 50%

Part 1:

Progress: 0%

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

Missing Caption

Saved: 7/26/2025 1:23:36 AM

Part 2:

Progress: 100%

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

URL #1
https://github.com/alvina-bit/IT1114.git

URL
https://github.com/alvina-bit/IT11

Saved: 7/26/2025 1:23:36 AM

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

Progress: 0%

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

Missing Caption

Not saved yet

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:

Through this milestone, I learned how to set up a basic networked application using Java sockets. I now understand how a server listens for incoming client connections and how multiple clients can communicate with the server at the same time using threads. I also learned the importance of organizing a project into logical packages like Client, Server, Common, and Exceptions—it really helps keep the code clean and easier to manage. On top of that, I got better at using Git for version control, especially when it comes to creating branches, committing changes, and merging through GitHub pull requests. I also got practice using terminal commands to start the server and client, and I gained more confidence in reading and understanding networked code logic. This project helped me connect what I've learned in class with real-world coding practices, especially around how communication between a server and multiple clients actually works.

Saved: 7/6/2025 6:43:34 PM

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 of this milestone was copying the Part5 folder, renaming it to Project, and organizing the files into the correct packages (Client, Server, Common, and Exceptions). Once I reviewed the lessons and understood what each class was responsible for, sorting them into the right package felt straightforward. I also found it easy to use Git to create the Milestone1 branch and push my changes to GitHub. The structure of the assignment was well-organized, and having clear instructions made it easier to follow along. Using VS Code's file explorer helped a lot in moving things quickly, and the package system made it easier to keep everything clean and easy to find.

Saved: 7/6/2025 6:41:42 PM

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)

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

Your Response:

The hardest part was setting up the server to handle multiple clients at the same time and making sure the room system worked correctly. I had to understand how threads work in Java because each client needs to be handled separately, and if not done right, it can crash the whole server or make it stop responding. It was also challenging to test this part—I had to open multiple terminal windows, run multiple clients, and make sure they were all able to connect, join or create rooms, and send messages without issues. Another tough part was debugging the room logic. For example, if a user left a room or disconnected, I had to make sure the room was updated correctly on the server-side without affecting other clients. Keeping track of who was in what room, and making sure users couldn't join or create rooms in the wrong order, took a lot of testing and attention to detail.

Saved: 7/6/2025 6:42:45 PM