Submission Worksheet

Submission Data

Course: IT114-450-M2025
Assignment: IT114 Milestone 1
Student: Mukaddis I. (mi348)
Status: Submitted | **Worksheet Progress:** 100%
Potential Grade: 10.00/10.00 (100.00%)
Received Grade: 0.00/10.00 (0.00%)
Started: 7/6/2025 1:25:48 PM
Updated: 7/6/2025 3:40:41 PM
Grading Link: https://learn.ethereallab.app/assignment/v3/IT114-450-M2025/it114-milestone-1/grading/mi348
View Link: https://learn.ethereallab.app/assignment/v3/IT114-450-M2025/it114-milestone-1/view/mi348

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

```
07/06/2025 13:25:24 [Project.Server.Server] (INFO):
> Server Starting
07/06/2025 13:25:24 [Project.Server.Server] (INFO):
> Server: Listening on port 3000
07/06/2025 13:25:24 [Project.Server.Room] (INFO):
> Room[lobby]: Created
07/06/2025 13:25:24 [Project.Server.Server] (INFO):
> Server: Created new Room lobby
07/06/2025 13:25:24 [Project.Server.Server] (INFO):
> Server: Waiting for next client
```

Image of Server Waiting for Client

```
//UCID MI348
    private void start(int port) {
        this.port = port;
        info("Listening on port " + this.port); //UCID:Mi348
        try (ServerSocket serverSocket = new ServerSocket(port)) { //Opens the server on the given port.
            createRoom(Room.LOBBY); //UCID:Mi348 - Create default lobby
            while (isRunning) {
                info(message:"Waiting for next client");
                Socket incomingClient = serverSocket.accept(); // blocks until client connects
                info(message:"Client connected"); //UCID:Mi348
                ServerThread serverThread = new ServerThread(incomingClient, this::onServerThreadInitialized);
                serverThread.start();
            }
```

Code showing how the code opens the server on the port and waits for client to connect

Saved: 7/6/2025 1:30:52 PM

Part 2:

Progress: 100%

Details:

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

Your Response:

The server begins by opening a ServerSocket on a specific port, which allows it to listen for incoming client connections. It enters a loop that continuously waits for clients by calling serverSocket.accept(), which blocks until a connection is made. When a client connects, the server creates a new ServerThread to handle communication with that client.

Saved: 7/6/2025 1:30:52 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

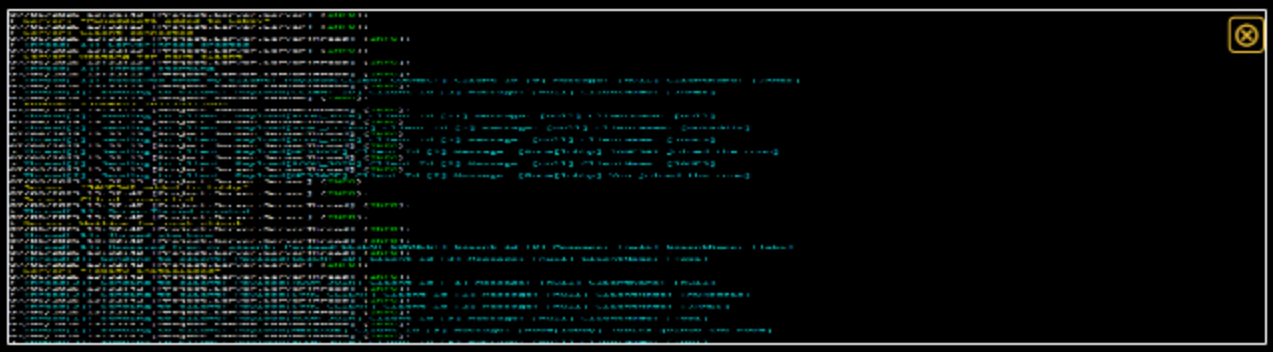
Progress: 100%

Part 1:

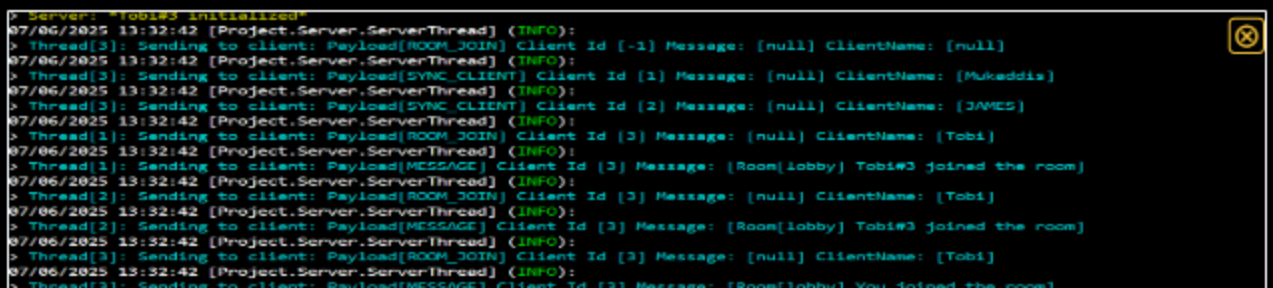
Progress: 100%

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



Shows 2 clients connecting to the server



```
07/06/2025 13:32:42 [Project.Server.Server] (INFO):
> Server: *Tobi#3 added to Lobby*
```

Shows the 3rd client connecting to the server

```
> Waiting for input
/name Tobi
07/06/2025 13:32:38 [Project.Client.Client] (INFO):
> Name set to Tobi
/connect localhost:3000
07/06/2025 13:32:42 [Project.Client.Client] (INFO):
> Client connected
07/06/2025 13:32:42 [Project.Client.Client] (INFO):
> Connected
07/06/2025 13:32:42 [Project.Client.Client] (INFO):
> Room[lobby] You joined the room
```

Shows the first client joining from the client side

```
> Client connected
07/06/2025 13:32:13 [Project.Client.Client] (INFO):
> Connected
07/06/2025 13:32:13 [Project.Client.Client] (INFO):
> Room[lobby] You joined the room
07/06/2025 13:32:42 [Project.Client.Client] (INFO):
> Room[lobby] Tobi#3 joined the room
```

Shows the second client joining from the client side

```
> Name set to Mukaddis
/connect localhost:3000
07/06/2025 13:31:43 [Project.Client.Client] (INFO):
> Client connected
07/06/2025 13:31:43 [Project.Client.Client] (INFO):
> Connected
07/06/2025 13:31:43 [Project.Client.Client] (INFO):
> Room[lobby] You joined the room
07/06/2025 13:32:13 [Project.Client.Client] (INFO):
> Room[lobby] JAMES#2 joined the room
07/06/2025 13:32:42 [Project.Client.Client] (INFO):
> Room[lobby] Tobi#3 joined the room
```

Shows the first Client connecting and can see other clients joining from the client side

```java
private void onServerThreadInitialized(ServerThread serverThread) {
    nextClientId = Math.max(++nextClientId, b:1);
    serverThread.setClientId(nextClientId);
    serverThread.sendClientId();// Send ID to client
    info(String.format(format:"*%s initialized*", serverThread.getDisplayName())); //UCID:Mi348
    try {
        joinRoom(Room.LOBBY, serverThread);
        info(String.format(format:"*%s added to Lobby*", serverThread.getDisplayName())); //UCID:Mi348
    } catch (RoomNotFoundException e) {
        info(String.format(format:"*Error adding %s to Lobby*", serverThread.getDisplayName())); //UCID:Mi348
        e.printStackTrace();
    }
```

Callback to initialize the thread after client connection

```java
while (isRunning) {
    info(message:"Waiting for next client"); // Logs waiting state
    Socket incomingClient = serverSocket.accept(); // blocks until client connects
```

```
    info(message:"Client connected"); //UCID:Mi348
    ServerThread serverThread = new ServerThread(incomingClient, this::onServerThreadInitialized);
    serverThread.start();
}
```

Loop that accepts multiple connections

Saved: 7/6/2025 1:43:04 PM

Part 2:

Progress: 100%

Details:

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

Your Response:

The server-side handles multiple connected clients by using multithreading. Each time a client connects, the Server accepts the socket and creates a new instance of ServerThread, which is a thread dedicated to communicating with that specific client. Each ServerThread runs independently, listening to and processing messages from its client.

Saved: 7/6/2025 1:43:04 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

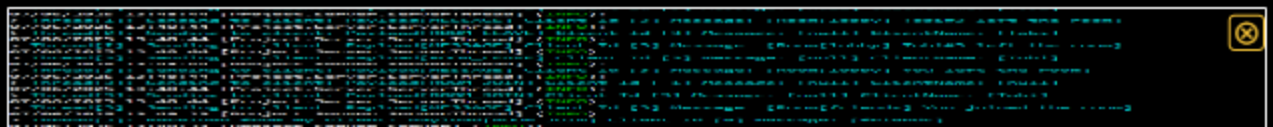
Progress: 100%

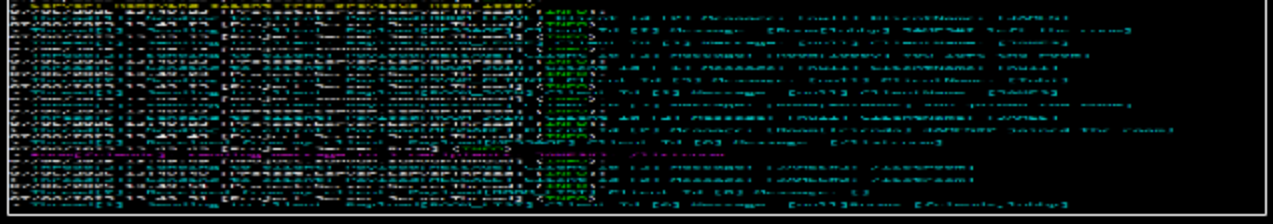
Part 1:

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)





This shows the creation of the room, leaving the current room, and joining a 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)); //UCID:Mi348
}
```

creating a room

```
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()); //UCID:Mi348
        currentRoom.removeClient(client);
    }
    Room next = rooms.get(nameCheck);
    next.addClient(client);
}
```

Joining a Room

```
protected void removeRoom(Room room) {
    rooms.remove(room.getName().toLowerCase());
    info(String.format(format:"Removed room %s", room.getName())); //UCID:Mi348
}
```

removing a room

Saved: 7/6/2025 2:05:11 PM

Part 2:

Progress: 100%

Details:

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

Your Response:

The server handles rooms by allowing clients to create, join, leave, and remove them. It creates a room if the name is unique, joins clients by moving them from their current room to a new one, and sends them back to the Lobby when leaving. Rooms can be removed unless it's the Lobby, and clients inside are handled before deletion.

Saved: 7/6/2025 2:05:11 PM

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

```
07/06/2025 14:06:48 [Project.Client.Client] (INFO):
> Waiting for input
/name Johnathan
07/06/2025 14:06:54 [Project.Client.Client] (INFO):
> Name set to Johnathan
/connect localhost:3000
07/06/2025 14:06:58 [Project.Client.Client] (INFO):
> Client connected
07/06/2025 14:06:58 [Project.Client.Client] (INFO):
> Connected
07/06/2025 14:06:58 [Project.Client.Client] (INFO):
> Room[lobby] You joined the room
```

Client 3-Johnathan Connect and Naming

```
/name Robert
07/06/2025 14:06:30 [Project.Client.Client] (INFO):
> Name set to Robert
/connect localhost:3000
07/06/2025 14:06:34 [Project.Client.Client] (INFO):
> Client connected
07/06/2025 14:06:34 [Project.Client.Client] (INFO):
> Connected
07/06/2025 14:06:34 [Project.Client.Client] (INFO):
> Room[lobby] You joined the room
```

Client 2-Robert Connect and Naming

```
> Waiting for input
/name Mukaddis
07/06/2025 14:05:45 [Project.Client.Client] (INFO):
> Name set to Mukaddis
/connect localhost:3000
07/06/2025 14:05:51 [Project.Client.Client] (INFO):
> Client connected
07/06/2025 14:05:51 [Project.Client.Client] (INFO):
> Connected
07/06/2025 14:05:51 [Project.Client.Client] (INFO):
> Room[lobby] You joined the room
```

Client 1-Mukaddis Connect and Naming

```
if (isConnection("/" + text)) {
    if (myUser.getClientName() == null || myUser.getClientName().isEmpty()) {
        LoggerUtil.INSTANCE.warning(
                TextFX.colorize(text:"Please set your name via /name <name> before connecting", Color.RED));
        return true;
    }
    String[] parts = text.trim().replaceAll(regex:" +", replacement:" ").split(regex:" ")[1].split(regex:":");
    connect(parts[0].trim(), Integer.parseInt(parts[1].trim()));
    sendClientName(myUser.getClientName());
    wasCommand = true;
```

/connect command

```
} else if (text.startsWith(Command.NAME.command)) {
    text = text.replace(Command.NAME.command, replacement:"").trim();//UCID MI348
    if (text.length() == 0) {
        LoggerUtil.INSTANCE.warning(TextFX.colorize(text:"This command requires a name as an argument", Color.RED));
        return true;
    }
    myUser.setClientName(text);
    LoggerUtil.INSTANCE.info(TextFX.colorize(String.format(format:"Name set to %s", myUser.getClientName()), Color.YELLOW));
    wasCommand = true;
```

/name COmmand

Saved: 7/6/2025 2:19:23 PM

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 enters the /name command, the client extracts and stores the given name locally. This name is required before attempting to connect. When the user enters the /connect command with a valid IP and port, the client opens a socket connection to the server and starts a listener thread to receive messages. After connecting, the client sends its name to the server.

Saved: 7/6/2025 2:19:23 PM

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 FriendlyFriends
07/06/2025 14:19:41 [Project.Client.Client] (INFO):
> Room[lobby] You left the room
07/06/2025 14:19:41 [Project.Client.Client] (INFO):
> Room[FriendlyFriends] You joined the room
07/06/2025 14:19:50 [Project.Client.Client] (INFO):
> Room[FriendlyFriends] Robert#2 joined the room
/listrooms
07/06/2025 14:20:03 [Project.Client.Client] (INFO):
> Room Results:
07/06/2025 14:20:03 [Project.Client.Client] (INFO):
> FriendlyFriends
lobby
```

creating room and seeing others join

```
> Room[lobby] Johnathan#3 left the room
/joinroom Friendlyfriends
07/06/2025 14:19:50 [Project.Client.Client] (INFO):
> Room[lobby] You left the room
07/06/2025 14:19:50 [Project.Client.Client] (INFO):
> Room[FriendlyFriends] You joined the room
/listrooms
07/06/2025 14:20:20 [Project.Client.Client] (INFO):
> Room Results:
07/06/2025 14:20:20 [Project.Client.Client] (INFO):
> FriendlyFriends
lobby
```

Friend Joining room

```
} else if (text.startsWith(Command.CREATE_ROOM.command)) {
    text = text.replace(Command.CREATE_ROOM.command, replacement:"").trim();
    if (text.length() == 0) {
        LoggerUtil.INSTANCE.warning(TextFX.colorize(text:"This command requires a room name as an argument", Color.RED));
        return true;
    }
```

```
    sendRoomAction(text, RoomAction.CREATE);
    wasCommand = true;
```

Code showing creating the room in Client.java

```
} else if (text.startsWith(Command.JOIN_ROOM.command)) {
    text = text.replace(Command.JOIN_ROOM.command, replacement:"").trim();
    if (text.length() == 0) {
        LoggerUtil.INSTANCE.warning(TextFX.colorize(text:"This command requires a room name as an argument", Color.RED));
        return true;
    }
    sendRoomAction(text, RoomAction.JOIN);
    wasCommand = true;
```

Code showing joining the room in Client.java

Saved: 7/6/2025 2:30:05 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 enters /createroom roomName, the client builds a ROOM_CREATE payload and sends it to the server, which then creates the room and confirms it. For /joinroom roomName, the client sends a ROOM_JOIN payload to the server. The server moves the client into the requested room and sends back a confirmation. The client then updates its user list and shows a message indicating the room change.

Saved: 7/6/2025 2:30:05 PM

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

Progress: 100%

Task #1 (1 pt.) - Evidence

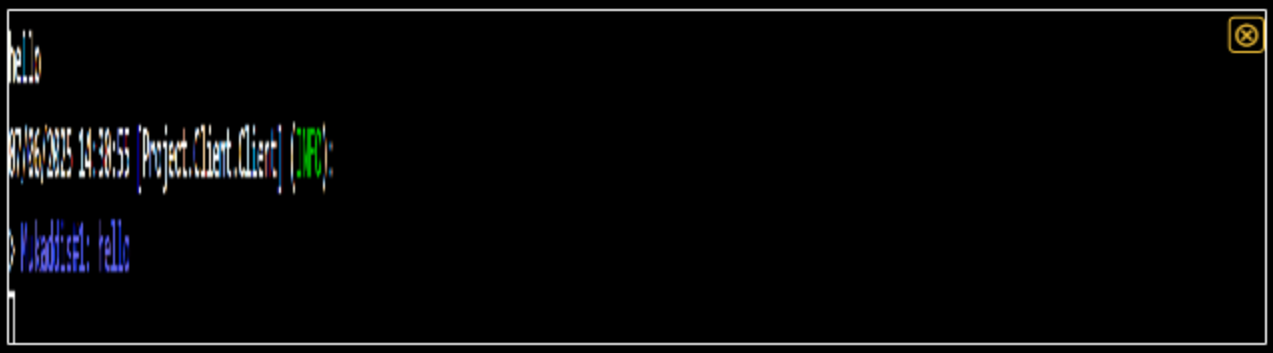
Progress: 100%

Part 1:

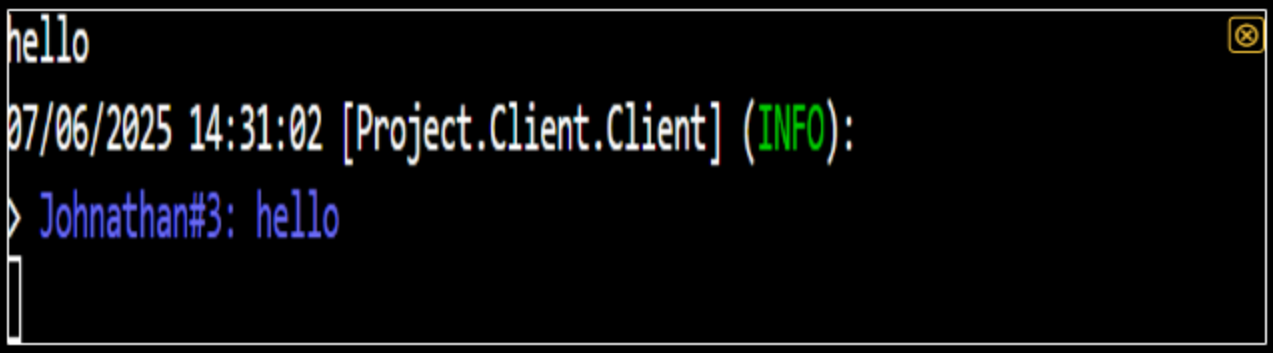
Progress: 100%

Details:

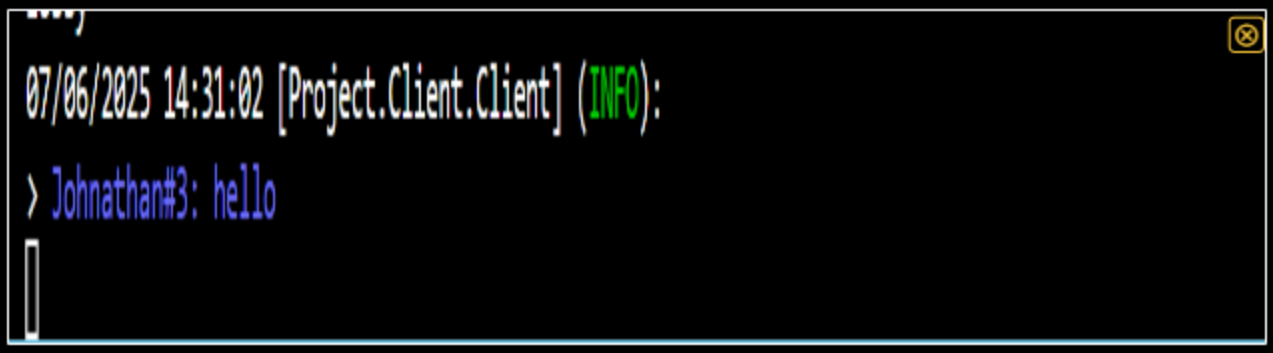
- Show the terminal output of a few messages from each of 3 clients
- Include examples of clients grouped into other rooms
- Show the relevant snippets of code that handle the message process from client to server-side and back



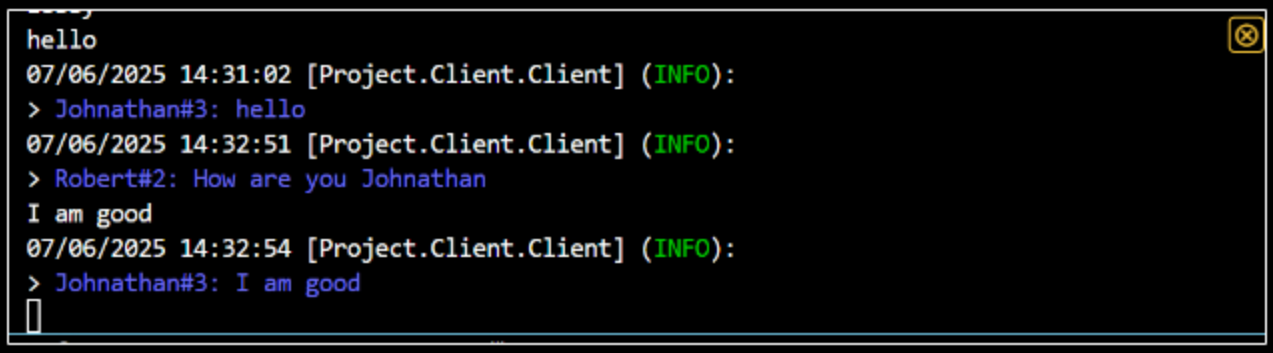
This is a client in the lobby



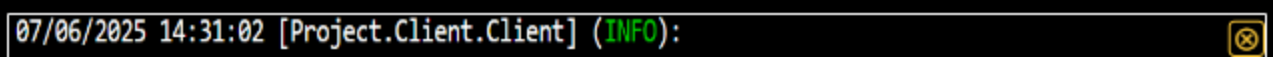
Client in private room sending hello message



Client receiving message, who is in the same private room



another example of private room Conversation



```
> Johnathan#3: hello
How are you Johnathan
07/06/2025 14:32:51 [Project.Client.Client] (INFO):
> Robert#2: How are you Johnathan
07/06/2025 14:32:54 [Project.Client.Client] (INFO):
> Johnathan#3: I am good
```

another example of private room Conversation

```java
private void sendMessage(String message) throws IOException {
    Payload payload = new Payload();
    payload.setMessage(message);
    payload.setPayloadType(PayloadType.MESSAGE);
    sendToServer(payload);
}
```

Client.java Send message

```java
protected synchronized void relay(ServerThread sender, String message) {
    if (!isRunning) return;

    String senderString = sender == null ? String.format(format:"Room[%s]", getName()) : sender.getDisplayName();
    final long senderId = sender == null ? Constants.DEFAULT_CLIENT_ID : sender.getClientId();
    final String formattedMessage = String.format(format:"%s: %s", senderString, message);

    info(String.format(format:"sending message to %s recipients: %s", clientsInRoom.size(), formattedMessage));

    clientsInRoom.values().removeIf(serverThread -> {
        boolean failedToSend = !serverThread.sendMessage(senderId, formattedMessage);
        if (failedToSend) {
            LoggerUtil.INSTANCE.warning(
                    String.format(format:"Removing disconnected %s from list", serverThread.getDisplayName()));
            disconnect(serverThread);
        }
        return failedToSend;
    });
```

Room.java Broadcasting to Other Clients in Room

```java
protected synchronized void handleMessage(ServerThread sender, String text) {
    relay(sender, text);
}
```

Broadcasting to Other Clients in Room

Saved: 7/6/2025 2:37:29 PM

Part 2:

Progress: 100%

Details:

- Briefly explain how the message code flow works

Your Response:

The message code flow starts when a client types a message (without a command prefix). The client wraps it in a Payload object with type MESSAGE and sends it to the server. On the server side, the message is received by ServerThread, which passes it to the current room's handleMessage() method. The room then relays the message to all connected clients using sendMessage(), and each receiving client displays it in their terminal.

Saved: 7/6/2025 2:37:29 PM

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

```
/disconnect
07/06/2025 14:53:28 [Project.Client.Client] (WARNING):
> Connection dropped
07/06/2025 14:53:28 [Project.Client.Client] (INFO):
> listenToServer thread stopped
```

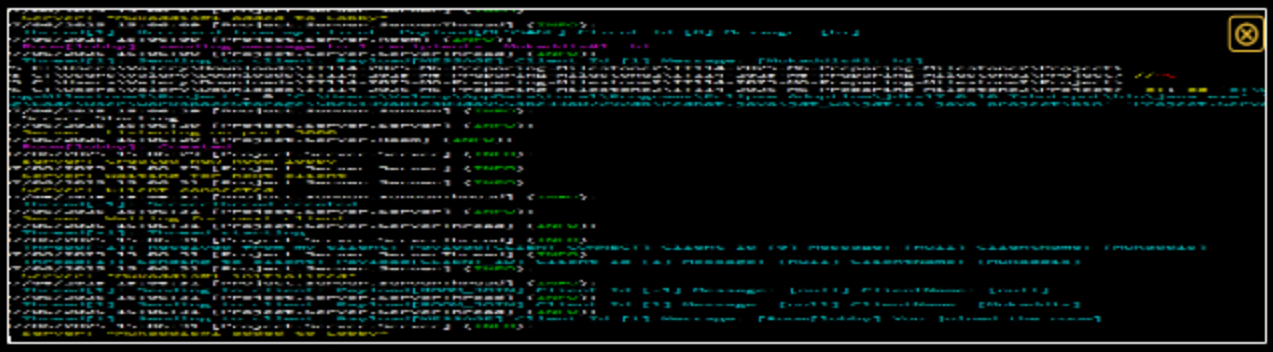
Client issuing /disconnect

```
private void sendDisconnect() throws IOException {
    Payload payload = new Payload();
    payload.setPayloadType(PayloadType.DISCONNECT);
    sendToServer(payload);
}
```

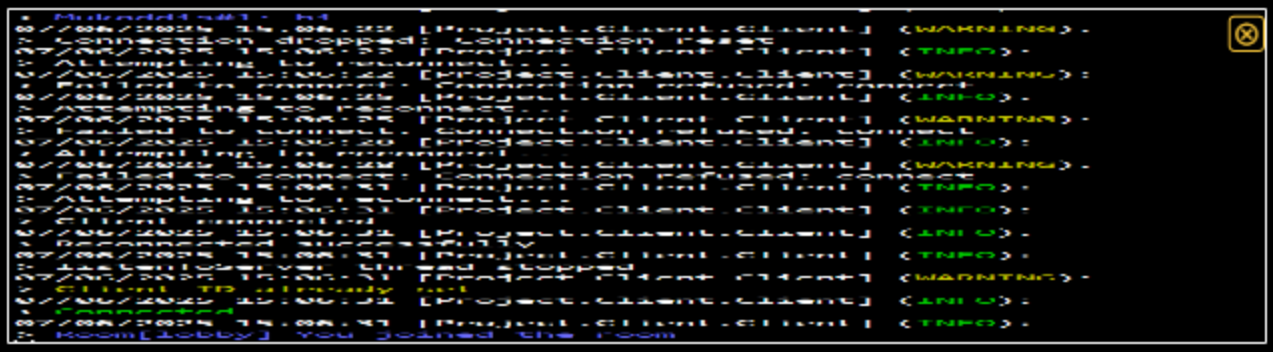
Client-side (sending disconnect request) Client.java

```
> Johnathan#3: hello
07/06/2025 14:56:36 [Project.Client.Client] (WARNING):
> Connection dropped
07/06/2025 14:56:36 [Project.Client.Client] (INFO):
> listenToServer thread stopped
```

Server stopped



This shows the client disconnecting and reconnecting at the end



This shows the client after the server has gone down, it continuously tries reconnecting to the server, then reconnects

```
private void listenToServer() {
    try {
        while (isRunning && isConnected()) {
            Payload fromServer = (Payload) in.readObject();
            if (fromServer != null) {
                processPayload(fromServer);
            } else {
                LoggerUtil.INSTANCE.info(message:"Server disconnected");
                break;
            }
        }
    } catch (Exception e) {
        LoggerUtil.INSTANCE.warning("Connection dropped: " + e.getMessage());
    } finally {
        closeServerConnection();
        if (isRunning) {
            attemptReconnect();
        }
    }
    LoggerUtil.INSTANCE.info(message:"listenToServer thread stopped");
}
```

If the connection is dropped or server is stopped, this handles the cleanup

```
private void closeServerConnection() {
    try {
        if (out != null) out.close();
```

```
        if (in != null) in.close();
        if (server != null) server.close();
    } catch (IOException e) {
        LoggerUtil.INSTANCE.severe(message:"Error closing connection", e);
    }
}
```

This cleans up the socket, input/output streams

```
private void close() {
    isRunning = false;
    closeServerConnection();
    LoggerUtil.INSTANCE.info(message:"Client terminated");
}
```

When the user types /quit, it shuts down the client fully

```
private void shutdown() {
    try {
        rooms.values().removeIf(room -> {
            room.disconnectAll();
            return true;
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

cleanly disconnect all rooms and clients

Saved: 7/6/2025 3:18:46 PM

Part 2:

Progress: 100%

Details:

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

Your Response:

The server has a shutdown hook that runs when it closes. It goes through all rooms and disconnects every client. On the client side, when the user types /disconnect or the server shuts down, the client sees the message, clears all users, and resets itself. Both sides also close any open sockets and streams to fully stop the connection.

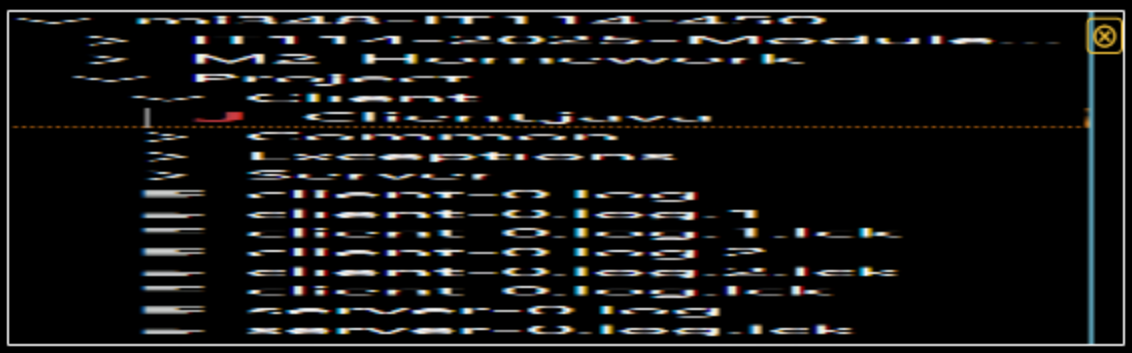
Saved: 7/6/2025 3:18:46 PM

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%



Not sure if this is correct but this is what I got

Saved: 7/6/2025 3:40:41 PM

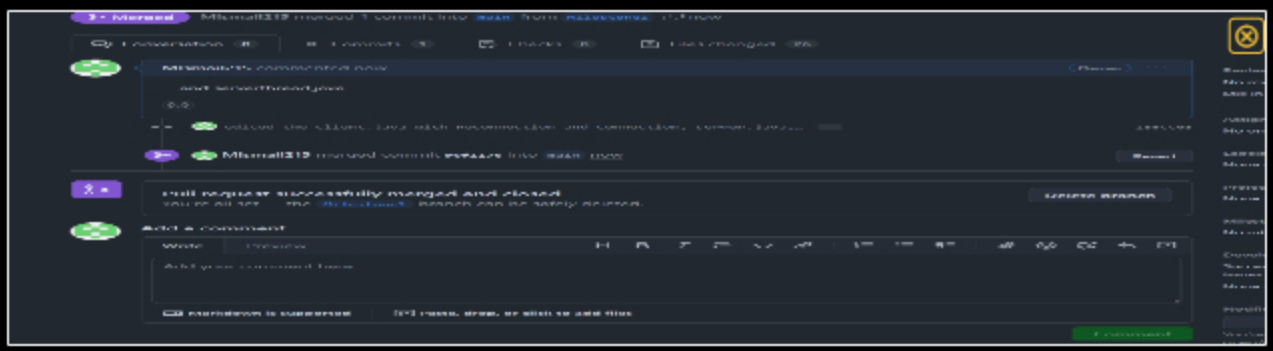
Task #2 (0.25 pts.) - Github Details

Progress: 100%

Part 1:

Progress: 100%

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



I had trouble with git so I couldnt commit regularly

Saved: 7/6/2025 3:37:03 PM

Part 2:

Progress: 100%

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

URL #1
https://github.com/MIsmail215/mi348-IT114-450/

https://github.com/MIsmail215/n

Saved: 7/6/2025 3:37:03 PM

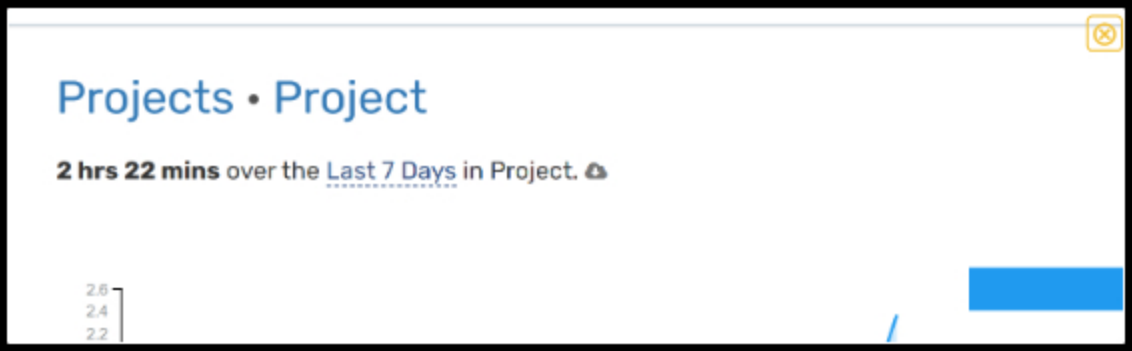
Task #3 (0.25 pts.) - WakaTime - Activity

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

Bottom of page



Top of page

Saved: 7/6/2025 3:38:24 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 client-server communication works in Java, including how clients connect to the server, join rooms, send messages, and how the server handles multiple clients using threads. I also learned how to implement commands like /nect, /name, /createroom, and how to process different types of payloads.

Saved: 7/6/2025 3:39:00 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 the assignment was using the chat commands like / nameect, and /createroom to interact with the server. Once the connection was set up, sending messages and seeing responses from the server felt straightforward and smooth. It was also easy to test basic client interactions after the initial setup.

Saved: 7/6/2025 3:39:23 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)

Your Response:

The hardest part of the assignment was getting the client-server connection to work properly and persist through disconnects. Implementing and debugging the auto-reconnect logic took time, especially making sure the client would reconnect smoothly without breaking the message flow.

Saved: 7/6/2025 3:39:54 PM