Submission Worksheet

Submission Data

Course: IT114-003-F2025
Assignment: IT114 Milestone 1
Student: Laura L. (lsl8)
Status: Submitted | **Worksheet Progress:** 100%
Potential Grade: 9.96/10.00 (99.60%)
Received Grade: 0.00/10.00 (0.00%)
Started: 11/3/2025 8:22:29 PM
Updated: 11/3/2025 11:36:49 PM
Grading Link: https://learn.ethereallab.app/assignment/v3/IT114-003-F2025/it114-milestone-1/grading/lsl8
View Link: https://learn.ethereallab.app/assignment/v3/IT114-003-F2025/it114-milestone-1/view/lsl8

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

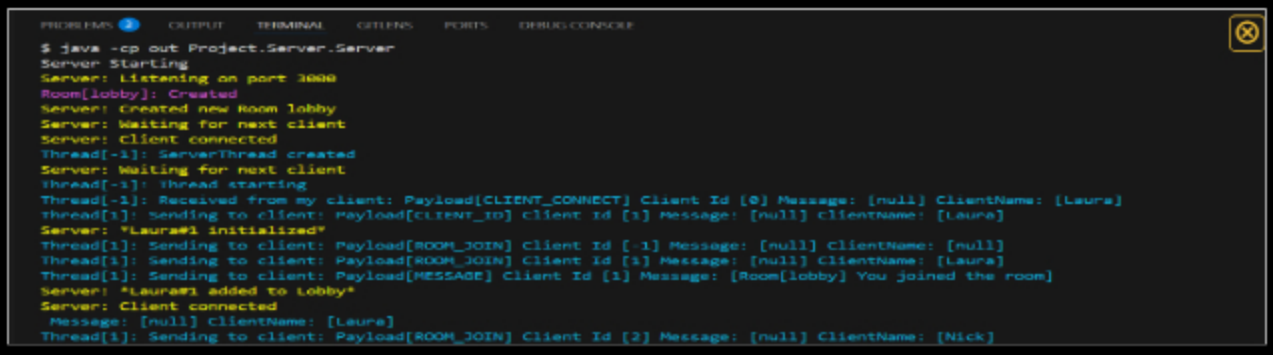
Progress: 100%

Part 1:

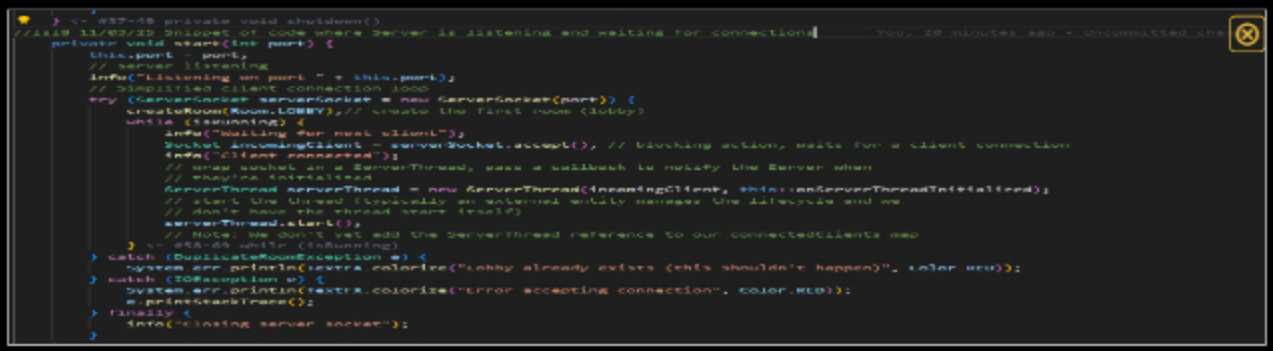
Progress: 100%

Details:

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



Terminal Output



Code

Saved: 11/3/2025 8:30:22 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 specific port, it then waits (blocks) until a client tries to connect and waits for the client using accept(). When a client connects, the server accepts the connection and returns a Socket object that represents a specific client. Each connected client runs on its own Thread so that the server can handle multiple clients at once without stopping the main program from listening for a new connection.

Saved: 11/3/2025 8:30:22 PM

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

Progress: 96%

Task #1 (1 pt.) - Evidence

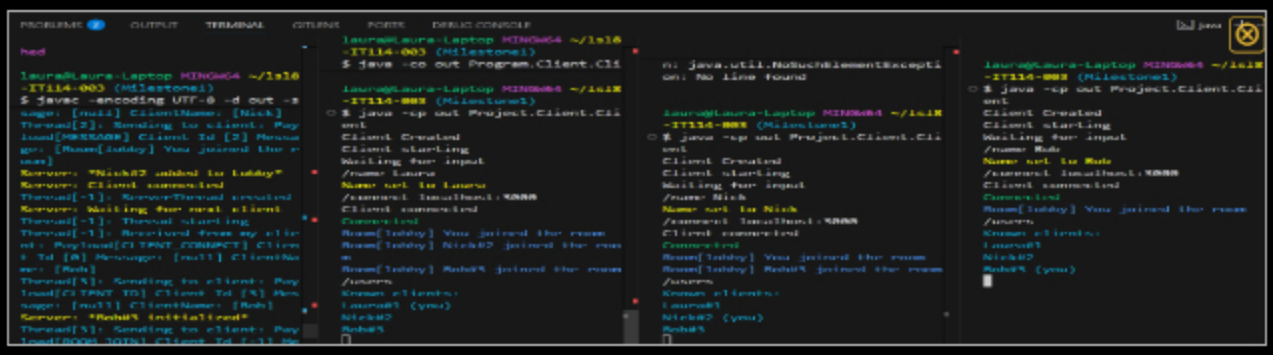
Progress: 96%

Part 1:

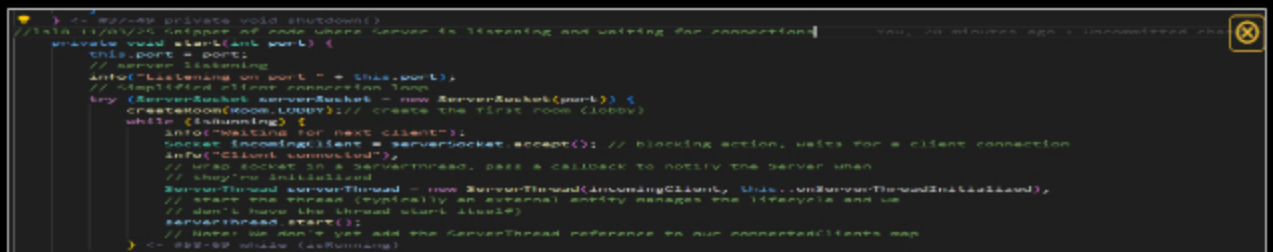
Progress: 93%

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



Missing Caption



Accepts multiple clients

Handling each client

Relay messages to clients

Saved: 11/3/2025 10:15:41 PM

Part 2:

Progress: 100%

Details:

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

Your Response:

When a new client connects, the server's main loop (Server.java) calls serverSocket.accept() and creates a new ServerThread just for that client. Each ServerThread runs in parallel, it listens for messages from its specific client using a blocking readObject() loop in BaseServerThread.java. Because every client has its own thread, they can all send and receive messages at the same time without interfering with one another. The server then uses shared, thread-safe structures to manage and broadcast messages to all connected clients efficiently.

Saved: 11/3/2025 10:15:41 PM

Section #3: (2 pts.) Feature: Server Will

Implement The Concept Of Rooms (With The Default Being "lobby")

Progress: 100%

Task #1 (2 pts.) - Evidence

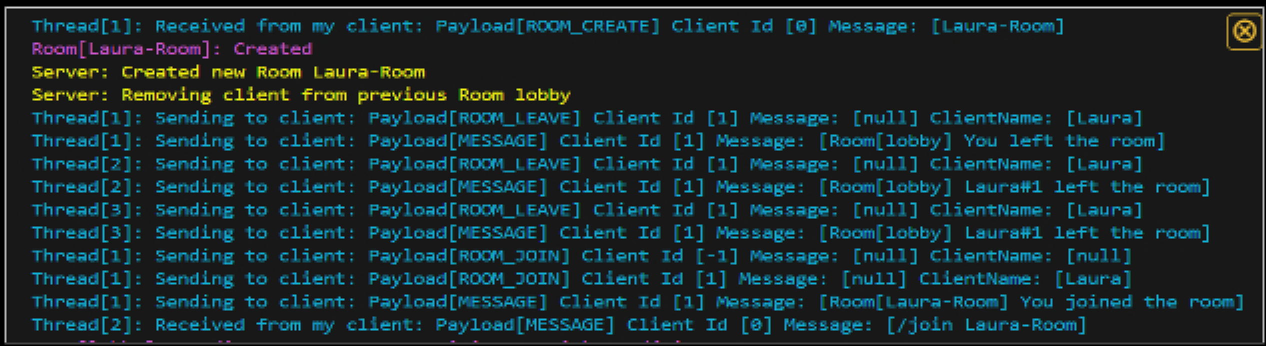
Progress: 100%

Part 1:

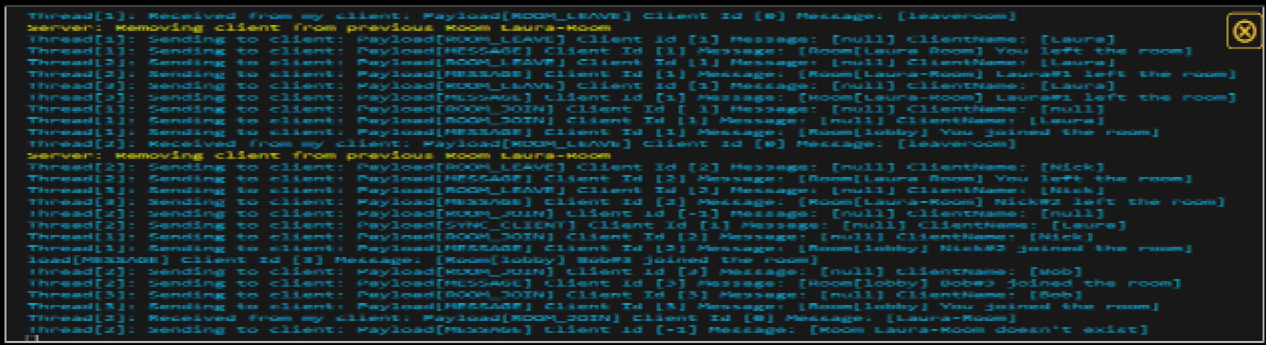
Progress: 100%

Details:

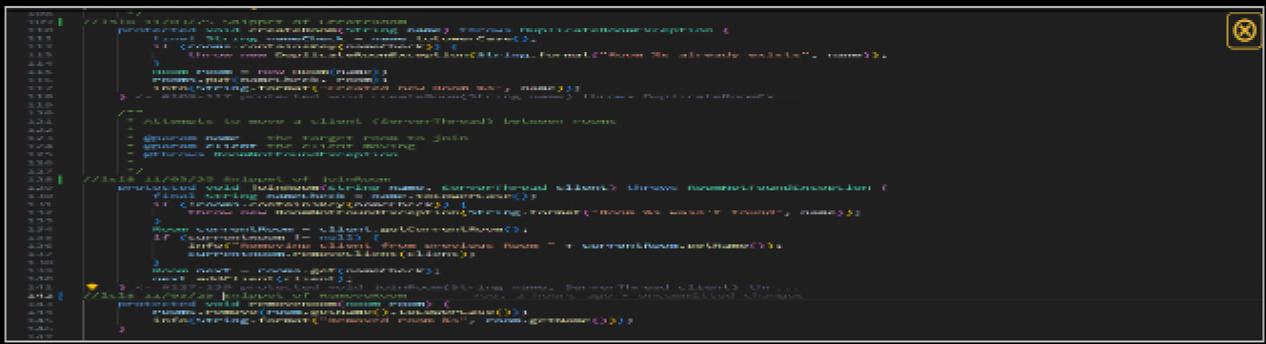
- Show the terminal output of rooms being created, joined, and removed (server-side)
- Show the relevant snippets of code that handle room management (create, join, leave, remove) (server-side)



Created a room and Joined



Removed the room



Code

Saved: 11/3/2025 10:23:52 PM

Part 2:

Progress: 100%

Details:

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

Your Response:

The server side handles room creation, joining/leaving, and removal through through logic between the Server and Room classes. When a client sends a request to join or create a room, the Server checks if the room already exists, if the room doesn't exist then the Server calls createRoom() to make a new Room object and store it. When joining, the Server calls joinRoom() which removes the client from the default room (Lobby) and adds them to the new one. Once the last client leaves the room that was created, the room is automatically removed and no longer exists once a client tries to join back.

Saved: 11/3/2025 10:23:52 PM

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

Progress: 100%

Task #1 (1 pt.) - Evidence

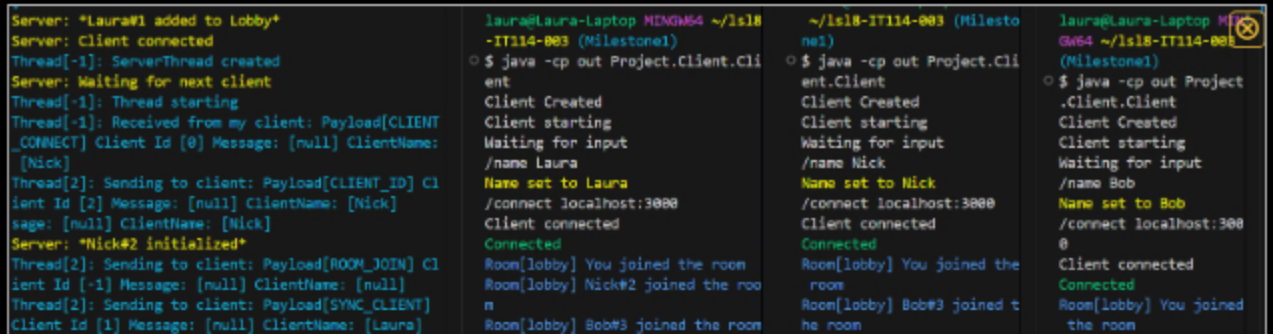
Progress: 100%

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



Terminal

```
//ls18 11/03/25 Handling the user's name before connecting to the server      You, 2 hours ago • Uncommitted changes
    private boolean processClientCommand(String text) throws IOException {
        boolean wasCommand = false;
        if (text.startsWith(Constants.COMMAND_TRIGGER)) {
            text = text.substring(1); // remove the /
            // System.out.println("Checking command: " + text);
            if (isConnection("/" + text)) {
                if (myUser.getClientName() == null || myUser.getClientName().isEmpty()) {
                    System.out.println(
                            TextFX.colorize("Please set your name via /name <name> before connecting", Color.RED));
                    return true;
                } // else we do have myUser.getClientName() == null || myUser.getClientName()
                // replaces multiple spaces with a single space
                // splits on the space after connect (gives us text and port)
                // splits on : to get host as index 0 and port as index 1
                String[] parts = text.trim().replaceAll(" +", " ").split(" ")[1].split(":");
                connect(parts[0].trim(), Integer.parseInt(parts[1].trim()));
                sendClientName(myUser.getClientName());// sync follow-up data (handshake)
                wasCommand = true;
            } else if (text.startsWith(Command.NAME.command)) {
                text = text.replace(Command.NAME.command, "").trim();
                if (text == null || text.length() == 0) {
                    System.out.println(TextFX.colorize("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("Name set to %s", myUser.getClientName()),
                        Color.YELLOW));
                wasCommand = true;
```

/name and /connect

```
//ls18 11/03/25 Client receives confirmation and logs "connected"
    // Start process*() methods
    private void processClientData(Payload payload) {
        if (myUser.getClientId() != Constants.DEFAULT_CLIENT_ID) {
            System.out.println(TextFX.colorize("Client ID already set, this shouldn't happen", Color.YELLOW));

        }
        myUser.setClientId(payload.getClientId());
        myUser.setClientName(((ConnectionPayload) payload).getClientName());// confirmation from Server
        knownClients.put(myUser.getClientId(), myUser);
        System.out.println(TextFX.colorize("Connected", Color.GREEN));
    } <- #354-363 private void processClientData(Payload payload)
```

Confirmation

Saved: 11/3/2025 10:53:33 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 types /name followed by their desired nickname, the Client recognizes it and uses it as a command and extracts the text after the "/name". This value is stored in the Client's User object as their display name. The program then confirms this change by printing a message like "Name set to Laura," letting the user know their name is saved locally. This step is required before connecting to the server. When the user later types the /connect command (for example, localhost:3000), the program first verifies that the input follows a valid pattern for a host and port. It then checks whether the user has already set a name. If not, it prevents the connection and prompts the user to use /name first. Once everything is valid, the client extracts the host and port values and calls the connect() method to establish a socket connection to the server. Immediately after connecting, the client sends a handshake message using the sendClientName() method, which packages the client's name into a ConnectionPayload and sends it to the server. On the server side, once the connection is received, the server assigns the client a unique ID and sends back a confirmation message containing this ID and the verified

username. The client then processes this message inside the processClientData() method and updates its own user data with the server-assigned ID

Saved: 11/3/2025 10:53:33 PM

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

Progress: 100%

Task #1 (2 pts.) - Evidence

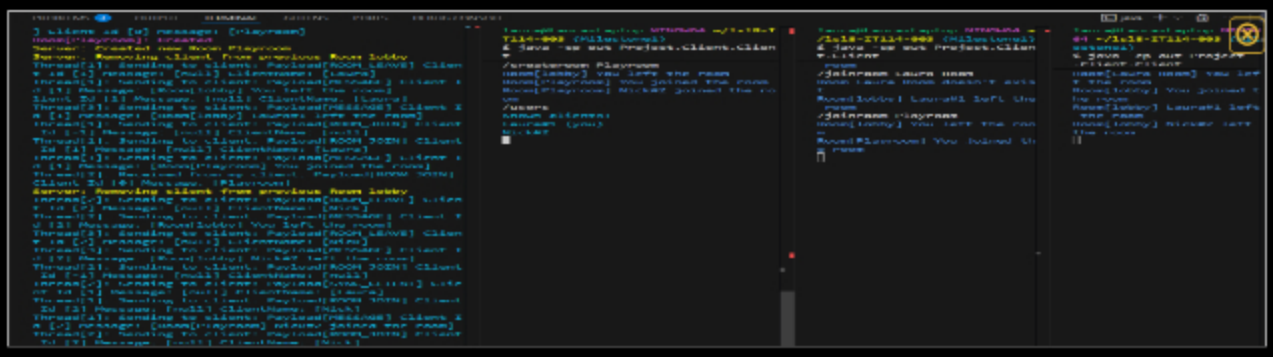
Progress: 100%

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 and /joinroom

```
//Isik 11/01/25 Create and Join Room Commands
    } else if (text.startsWith(Command.CREATE_ROOM.command)) {
        text = text.replace(Command.CREATE_ROOM.command, "").trim();
        if (text == null || text.length() == 0) {
            System.out.println(TextFX.colorize("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, "").trim();
        if (text == null || text.length() == 0) {
            System.out.println(TextFX.colorize("This command requires a room name as an argument", Color.RED));
            return true;
        }
        sendRoomAction(text, RoomAction.JOIN);
        wasCommand = true;
```

Create_Room and Join_Room

Saved: 11/3/2025 11:00:26 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 or /joinroom , the Client.java class detects these commands through its processClientCommand() method. On the client side, the input text is checked against the Command.CREATE_ROOM and Command.JOIN_ROOM keywords. If it matches, the command text is stripped from the input, and the sendRoomAction() method is called with the room name and the corresponding RoomAction enum (either CREATE or JOIN). The method then builds a Payload object, sets the message to the room name, and assigns the PayloadType to either ROOM_CREATE or ROOM_JOIN. This payload is serialized and sent to the server via sendToServer(payload)

Saved: 11/3/2025 11:00:26 PM

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

Progress: 100%

Task #1 (1 pt.) - Evidence

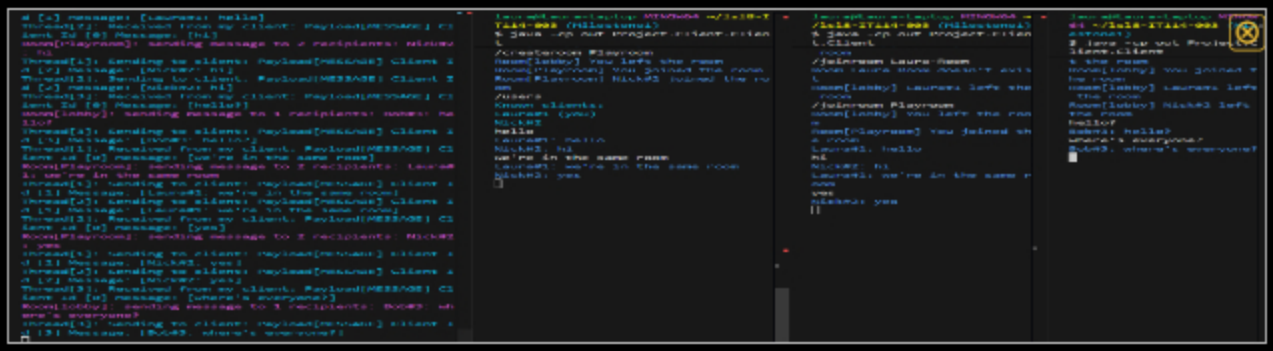
Progress: 100%

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



Messages between clients in the same and separate rooms

```java
//lsl8 11/03/25 String Message
    /**
     * Sends a message to the server
     *
     * @param message
     * @throws IOException
     */
    private void sendMessage(String message) throws IOException {
        Payload payload = new Payload();
        payload.setMessage(message);
        payload.setPayloadType(PayloadType.MESSAGE);
        sendToServer(payload);
    } <- #249-254 private void sendMessage(String message) throws IOException
```

String Message

```java
//lsl8 11/03/25 Replay Messages      You, 3 hours ago • Uncommitted changes
    protected synchronized void handleMessage(ServerThread sender, String text) {
        relay(sender, text);
    }
    // end handle methods
} <- #13-269 public class Room implements AutoCloseable
```

Relay Messages

Sending Messages

Saved: 11/3/2025 11:12:27 PM

Part 2:

Progress: 100%

Details:

- Briefly explain how the message code flow works

Your Response:

It works through a continuous exchange of Payload objects between the client and server using object streams. On the client side, when a user types a message that is not a command (like /connect or /name), the program creates a new Payload object in the sendMessage() method. This payload includes the user's message and is tagged with the type PayloadType.MESSAGE. On the server side, each client runs on its own ServerThread, which constantly listens for incoming objects using an input stream inside a loop in BaseServerThread.run(). When a message payload

is received, it is passed to ServerThread.processPayload(). If the payload type is MESSAGE, it calls the room's handleMessage() method, which in turn calls relay(). The relay() method in Room.java loops through all connected clients in that room, creating a new message payload for each and sending it out using sendToClient(). This allows every participant to receive the message simultaneously. Back on the client side, each client runs listenToServer() in a background thread that continuously reads payloads from the input stream. When a MESSAGE payload arrives, it is handled by processPayload() and passed to processMessage(), which displays the formatted text in the console.

Saved: 11/3/2025 11:12:27 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

disconnecting and connecting back

```
//lsl8 11/03/25 Sends the disconnect action to the server        You, 3 hours ago • Uncom
    /**
     * Sends a disconnect action to the server
     *
     * @throws IOException
     */
    private void sendDisconnect() throws IOException {
        Payload payload = new Payload();
        payload.setPayloadType(PayloadType.DISCONNECT);
        sendToServer(payload);
```

sendDisconnect

```java
//ls18 11/03/25 how the client processes disconnect
private void processDisconnect(Payload payload) {
    if (payload.getClientId() == myUser.getClientId()) {
        knownClients.clear();
        myUser.reset();
        System.out.println(TextFX.colorize("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("%s disconnected", disconnectedUser.getDisplayName()),
                    Color.RED));
        }
    } <- #372-378 else if (knownClients.containsKey(payload.getClientId()))
```

client processes disconnect

```java
//ls18 11/03/25 disconnect payload
        case DISCONNECT:
            currentRoom.handleDisconnect(this);
            break;
```

disconnect payload

```java
private synchronized void disconnect(ServerThread client) {
    if (!isRunning) { // block action if Room isn't running
        return;
    }
    ServerThread disconnectingServerThread = clientsInRoom.remove(client.getClientId());
    if (disconnectingServerThread != null) {
        clientsInRoom.values().removeIf(serverThread -> {
            if (serverThread.getClientId() == disconnectingServerThread.getClientId()) {
                return true;
            }
            boolean failedToSend = !serverThread.sendClientInfo(disconnectingServerThread.getClientId(),
                    disconnectingServerThread.getClientName(), RoomAction.LEAVE);
            if (failedToSend) {
                System.out.println(
                        String.format("Removing disconnected %s from list", serverThread.getDisplayName()));
                disconnect(serverThread);
            } <- #160-172 if (failedToSend)
            return failedToSend;
        }); <- #162-174 clientsInRoom.values().removeIf
        relay(null, disconnectingServerThread.getDisplayName() + " disconnected");
        disconnectingServerThread.disconnect();
    } <- #148-177 if (disconnectingServerThread != null)
    autoCleanup();
} <- #135-179 private synchronized void disconnect(ServerThread client)
```

Room.java forwards disconnect

```java
/**
 * Terminates the server-side of the connection
 */
//ls18 11/03 termination
protected void disconnect() {
    if (!isRunning) {
        // prevent multiple triggers if this gets called consecutively
        return;
    }
    info("Thread being disconnected by server");
    isRunning = false;
    this.interrupt(); // breaks out of blocking read in the run() method
    cleanup(); // good practice to ensure data is written out immediately
```

termination

Saved: 11/3/2025 11:27:09 PM

Part 2:

Progress: 100%

Details:

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

Your Response:

When a client disconnects, the program makes sure everything shuts down cleanly on both sides. On the client side, typing /disconnect sends a message to the server to let it know the user is leaving. The client then stops running, closes its input and output streams, and shuts down the socket so no background processes are left open. On the server side, each client runs on its own thread, so when the server receives a disconnect signal, it removes that user from the chat room, sends a message to the other clients letting them know the user left, and then stops and cleans up the thread connected to that client. This process ensures that both the client and server end their connections safely without errors.

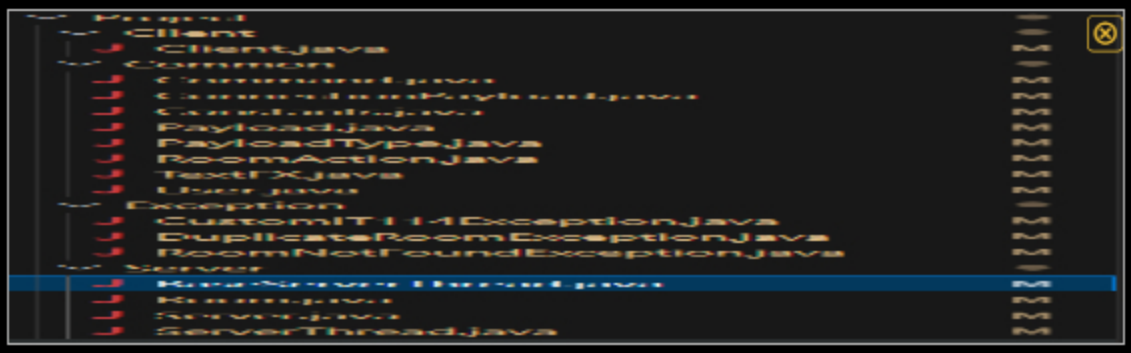
Saved: 11/3/2025 11:27:09 PM

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%



Workspace structure

Saved: 11/3/2025 11:27:37 PM

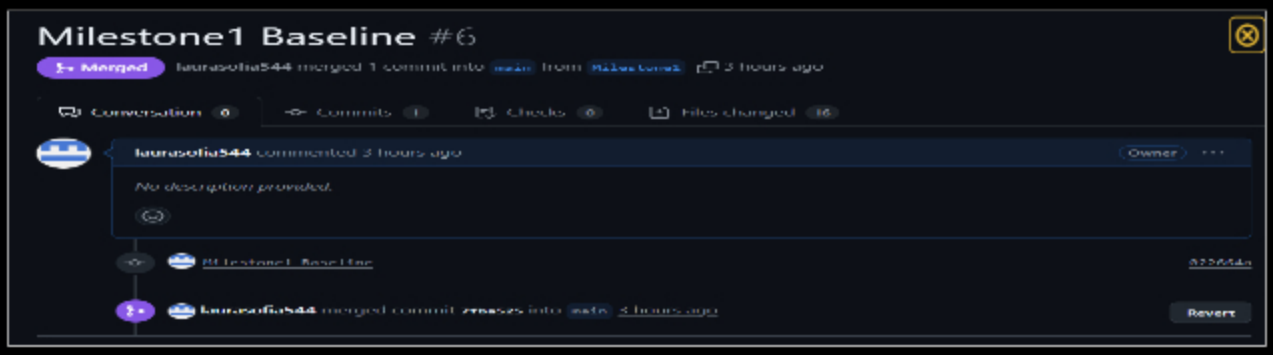
Task #2 (0.25 pts.) - Github Details

Progress: 100%

Part 1:

Progress: 100%

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



Accidentally closed the merge before adding the comments

#2

Saved: 11/3/2025 11:31:41 PM

Part 2:

Progress: 100%

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

URL #1
https://github.com/laurasofia544/lsl8-IT11[illegible]/

URL
https://github.com/laurasofia544/

Saved: 11/3/2025 11:31:41 PM

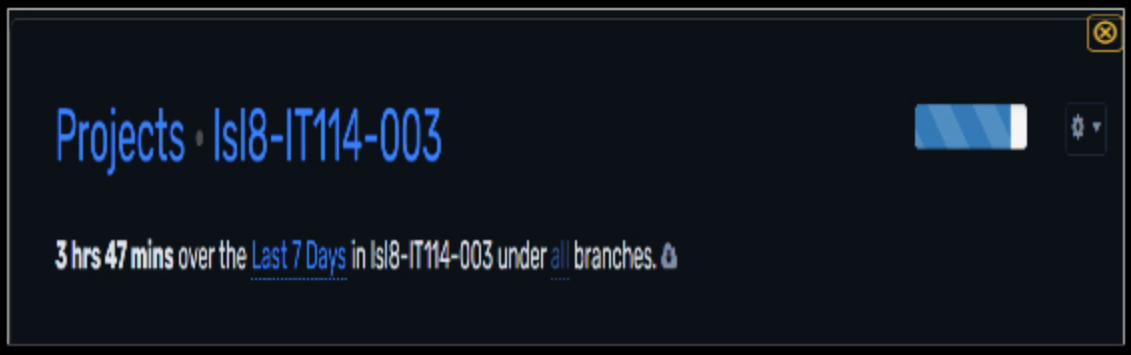
Task #3 (0.25 pts.) - WakaTime - Activity

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



Overall time

Individual time

Saved: 11/3/2025 11:33:13 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:

Overall, I learned how a client-server chat program works from start to finish. I saw how the server waits for incoming connections and creates a new thread for each client so multiple users can be connected at the same time. I learned how commands like /connect, /name, /createroom, and /joinroom are handled and how rooms are created, joined, and managed on the server side. I also learned how both the client and server handle disconnections in a safe way by closing sockets, stopping threads, and cleaning up resources

Saved: 11/3/2025 11:35:00 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 understanding how the client connects to the server and sends basic messages. Once the connection was established, it was pretty straightforward to see how data was sent and received through the input and output streams. Running simple commands like /connect, /name, and sending regular chat messages helped me clearly see how the communication worked between both sides

Saved: 11/3/2025 11:35:48 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 everything to compile and work together correctly, especially with all the different files and classes that depended on each other. Making sure the imports, packages, and file paths were set up properly was challenging and small mistakes would cause a lot of errors. It was also tricky to understand how multiple threads worked at the same time when several clients were connected, and how the server handled each one separately

Saved: 11/3/2025 11:36:49 PM