

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

```
PROBLEMS OUTPUT TERMINAL GITHUB PORTS DEBUG CONSOLE
$ java -cp out Project.Server.Server
Server Starting
Server: Listening on port 3000
Room[lobby]: Created
Server: Created new Room lobby
Server: Waiting for next client
Server: Client connected
Thread[-1]: ServerThread created
Server: Waiting for next client
Thread[-1]: Thread starting
Thread[-1]: Received from my client: Payload[CLIENT_CONNECT] Client Id [0] Message: [null] ClientName: [Laura]
Thread[-1]: Sending to client: Payload[CLIENT_ID] Client Id [x] Message: [null] ClientName: [Laura]
Server: "Laura#1 initialized"
Thread[-1]: Sending to client: Payload[ROOM_JOIN] Client Id [-1] Message: [null] ClientName: [null]
Thread[-1]: Sending to client: Payload[ROOM_JOIN] Client Id [x] Message: [null] ClientName: [Laura]
Thread[-1]: Sending to client: Payload[MESSAGE] Client Id [1] Message: [Rose[lobby] You joined the room]
Server: "Laura#1 added to lobby"
Server: Client connected
Message: [null] ClientName: [Leure]
Thread[-1]: Sending to client: Payload[ROOM_JOIN] Client Id [2] Message: [null] ClientName: [Nick]
```

Terminal Output

```
y-x@y-x-OptiPlex-5070:~/Documents$ private void shutdown() {
    // ...
}
private void start(int port) {
    this.port = port;
    // server listening
    info("Listening on port " + this.port);
    // accept connection
    try (ServerSocket serverSocket = new ServerSocket(port)) {
        createRoom(Room.Lobby); // Create the first room (lobby)
        while (true) {
            info("Waiting for next client");
            Socket client = serverSocket.accept(); // Blocking action, waits for a client connection
            Thread clientThread = new Thread(new ClientHandler(client));
            clientThread.start();
            // Once we have a ServerThread, pass a callback to notify the Server when
            // message is received
            ServerThread serverThread = new ServerThread(new ClientHandler(client), this);
            serverThread.setClient(client);
            // Once we have a ClientHandler, we can easily invoke it directly without managing the lifecycle and so
            // don't have the thread start directly
            serverThread.start();
            // Note: we don't yet add the serverThread reference to our connectedClients map
        }
    } catch (IOException e) {
        System.err.println("Server error: " + e.getMessage());
    } catch (Exception e) {
        System.err.println(TextUtil.colorize("Error accepting connection", Color.RED));
    } finally {
        info("Existing server socket");
    }
}
```

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

```
> <-- May be -M private void shutdown()
// This is a copy of code where Server is listening and waiting for connection
private void startServer() {
    this.port = 8080;
    // Start listening
    info("Listening on port " + this.port);
    // Simplified client connection loop
    try {
        ServerSocket serverSocket = new ServerSocket(port);
        createRoom(RoomType.Lobby); // Create the first room (lobby)
        while (true) {
            // Informing for next client
            Socket incomingClient = serverSocket.accept(); // Blocking action, waits for a client connection
            info("Client connected");
            // Wrap socket in a ServerThread. Pass a callback to notify the server when
            // client disconnects
            ServerThread serverThread = new ServerThread(incomingClient, this, onServerThreadInitialization);
            // Starts one thread (synthetically an external entity manages the lifecycle and we
            // don't have the thread start itself)
            serverThread.start();
            // Since we can't type add the ServerThread reference to our acceptedClients map
            acceptedClients.put(serverThread, serverThread);
        }
    } catch (IOException e) {
        error("Error starting server: " + e.getMessage());
    }
}
```

```
    } catch (IOException e) {
        System.out.println("IOException: " + e.getMessage());
    } catch (TestFXException e) {
        System.out.println("TestFX exception: " + e.getMessage());
    } finally {
        info("Cleaning server socket");
    }
}
```

Accepts multiple clients

```
// 3.5.8 31/08/25 handling each client
private void run() {
    info("Thread starting");
    try (ObjectOutputStream out = new ObjectOutputStream(client.getOutputStream());
         ObjectOutputStream in = new ObjectOutputStream(client.getInputStream())) {
        this.out = out;
        isRunning = true;
        ...
        new java.util.Timer().schedule(new java.util.TimerTask() {
            @Override
            public void run() {
                if (getClientName() == null || getClientName().isEmpty()) {
                    info("Client name not received, disconnecting");
                    disconnect();
                }
            }
        }, 1000); // 1000 ms = 1 second
        payloadFromClient();
    } catch (IOException e) {
        error("IOException: " + e.getMessage());
    } catch (TestFXException e) {
        error("TestFX exception: " + e.getMessage());
    } finally {
        if (isRunning) {
            disconnect();
        }
    }
}

// 3.5.8 31/08/25 public void run()
public void run() {
    info("Thread starting");
    try (ObjectOutputStream out = new ObjectOutputStream(client.getOutputStream());
         ObjectOutputStream in = new ObjectOutputStream(client.getInputStream())) {
        this.out = out;
        isRunning = true;
        ...
        new java.util.Timer().schedule(new java.util.TimerTask() {
            @Override
            public void run() {
                if (getClientName() == null || getClientName().isEmpty()) {
                    info("Client name not received, disconnecting");
                    disconnect();
                }
            }
        }, 1000); // 1000 ms = 1 second
        payloadFromClient();
    } catch (IOException e) {
        error("IOException: " + e.getMessage());
    } catch (TestFXException e) {
        error("TestFX exception: " + e.getMessage());
    } finally {
        if (isRunning) {
            disconnect();
        }
    }
}
```

Handling each client

```
// 3.5.8 31/08/25 Relays messages and status to all clients
private void joinStatusRelay(ServerThread client, boolean didJoin) {
    clientsInRoom.values().removeIf(serverThread -> {
        String formattedMessage = String.format("Room[%s] %s %s the room",
            client.getName(),
            client.getClientId() == serverThread.getClientId() ? "You"
                : client.getDisplayName(),
            didJoin ? "joined" : "left"); // <- H84-88 String formattedMessage = String.format(
        final long senderId = client == null ? Constants.DEFAULT_CLIENT_ID : client.getClientId();
        // Share info of the client joining or leaving the room
        boolean failedToSend = !serverThread.sendClientInfo(client.getClientId(),
            client.getClientName(), didJoin ? RoomAction.JOIN : RoomAction.LEAVE);
        // Send the server generated message to the current client
        boolean failedToSend = !serverThread.sendMessage(senderId, formattedMessage);
        if (failedToSend || failedToSend) {
            System.out.println(
                String.format("Removing disconnected %s from list", serverThread.getDisplayName()));
            disconnect(serverThread);
        } // <- #85-99 if (failedToSend || failedToSend)
    });
    return failedToSend;
} // <- #82-101 clientsInRoom.values().removeIf
} // <- #82-102 private void joinStatusRelay(ServerThread client, boolean did...
```

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)

```
Thread[1]: Received from my client: Payload[ROOM_CREATE] Client Id [0] Message: [Laura-Room]
Room[Laura-Room]: Created
Server: Created new Room Laura-Room
Server: Removing client from previous Room lobby
Thread[1]: Sending to client: Payload[ROOM_LEAVE] Client Id [1] Message: [null] ClientName: [Laura]
Thread[1]: Sending to client: Payload[MESSAGE] Client Id [1] Message: [Room[lobby] You left the room]
Thread[2]: Sending to client: Payload[ROOM_LEAVE] Client Id [1] Message: [null] ClientName: [Laura]
Thread[2]: Sending to client: Payload[MESSAGE] Client Id [1] Message: [Room[lobby] Laura#1 left the room]
Thread[3]: Sending to client: Payload[ROOM_LEAVE] Client Id [1] Message: [null] ClientName: [Laura]
Thread[3]: Sending to client: Payload[MESSAGE] Client Id [1] Message: [Room[lobby] Laura#1 left the room]
Thread[1]: Sending to client: Payload[ROOM_JOIN] Client Id [-1] Message: [null] ClientName: [null]
Thread[1]: Sending to client: Payload[ROOM_JOIN] Client Id [1] Message: [null] ClientName: [Laura]
Thread[1]: Sending to client: Payload[MESSAGE] Client Id [1] Message: [Room[Laura-Room] You joined the room]
Thread[2]: Received from my client: Payload[MESSAGE] Client Id [0] Message: [/join Laura-Room]
```

Created a room and Joined

Removed the room

```
    // Create a new instance of the class
    var myObject = new Object();
    myObject.name = "John";
    myObject.age = 30;
    myObject.job = "Developer";
    myObject.isMarried = false;

    // Print the object to the console
    console.log(myObject);

    // Output: {name: "John", age: 30, job: "Developer", isMarried: false}
```

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 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

```
Server: *Laura#1 added to Lobby*
Server: Client connected
Thread[-1]: ServerThread created
Server: Waiting for next client
Thread[-1]: Thread starting
Thread[-1]: Received from my client: Payload[CLIENT_CONNECT] Client Id [0] Message: [null] ClientName: [Nick]
Thread[2]: Sending to client: Payload[CLIENT_ID] Client Id [2] Message: [null] ClientName: [Nick]
Message: [null] ClientName: [Nick]
Server: *Nick#2 initialized*
Thread[2]: Sending to client: Payload[ROOM_JOIN] Client Id [-1] Message: [null] ClientName: [null]
Thread[2]: Sending to client: Payload[SYNC_CLIENT] Client Id [1] Message: [null] ClientName: [Laura]
```

```
laura@Laura-Laptop MINGW64 ~/ls18-IT114-003 (Milestone1)
$ java -cp out Project.Client.Client
Client Created
Client starting
Waiting for input
/nick Laura
Name set to Laura
/connect localhost:3000
Client connected
Connected
Room[lobby] You joined the room
Room[lobby] Nick#2 joined the room
Room[lobby] Bob#3 joined the room
```

```
~/ls18-IT114-003 (Milestone1)
$ java -cp out Project.Client.Client
Client Created
Client starting
Waiting for input
/nick Nick
Name set to Nick
/connect localhost:3000
Client connected
Connected
Room[lobby] You joined the room
Room[lobby] Nick#2 joined the room
Room[lobby] Bob#3 joined the room
```

```
laura@Laura-Laptop MINGW64 ~/ls18-IT114-003 (Milestone1)
$ java -cp out Project.Client.Client
Client Created
Client starting
Waiting for input
/nick Bob
Name set to Bob
/connect localhost:3000
Client connected
Connected
Room[lobby] You joined the room
Room[lobby] Nick#2 joined the room
Room[lobby] Bob#3 joined the room
```

Terminal

```
// lsl8 11/03/25 Handling the user's name before connecting to the server
private boolean processClientCommand(String text) throws IOException {
    if (text.startsWith(Constants.COMMAND_TRIGGER)) {
        text = text.substring(1); // remove the /
        // System.out.println("Received command: " + text);
        if (!isConnection(text)) {
            if (myUser.getClientName() == null || myUser.getClientName().isEmpty()) {
                System.out.println(TextFX.colorize("Please set your name via /name command before connecting!", Color.RED));
                return true;
            } else if (myUser.getClientName().length() == 0) { // myUser.getClientName() ...
                // requires multiple spaces with a single space
                // splits on ' ' to get host as index 0 and port as index 1
                String[] parts = text.trim().replace(" ", " ").split(" " + Constants.DELIMITER);
                connect(parts[0], parts[1]); // sync follow-up data (handshake)
                sendClientName(myUser.getClientName()); // sync follow-up data (handshake)
            }
        }
        if (text.equals(Command.NAMP_COMMAND)) {
            text = text.replace(Command.NAMP_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));
        lastCommand = text;
    }
}
```

/name and /connect

```
// lsl8 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));
}
```

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

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/join 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

```
//LSIR 11/03/25 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 Boom and Join Boom



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

≡, Part 2:

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

```
//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

```
//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 (`/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

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 occurring
- 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

The four terminal windows show the following log output:

- Terminal 1:** Shows a client joining a room and sending a message. It then receives a room leave message from another client.
- Terminal 2:** Shows a client joining a room and sending a message. It then receives a room leave message from another client.
- Terminal 3:** Shows a client joining a room and sending a message. It then receives a room leave message from another client.
- Terminal 4:** Shows a client joining a room and sending a message. It then receives a room leave message from another client.

disconnecting and connecting back

```

//lsl8 11/03/25 Sends the disconnect action to the server side
You, 3 hours ago * Unc
    /**
     * 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

```
//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

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

disconnect payload

```
private synchronized void disconnectClient(DisconnectEvent event) {
    if (!isRunning) { // block action if Room isn't running
        return;
    }
    ServerThread disconnectingServerThread = event.getServerThread();
    ClientId client = event.getClientId();
    disconnectingServerThread = null;

    clientsInRoom.values().removeIf(serverThread -> {
        if (serverThread.getClientId() == disconnectingServerThread.getClientId()) {
            return true;
        }
        boolean failedToSend = !serverThread.sendClientInfo(disconnectingServerThread.getClientId(),
                disconnectingServerThread.getDisplayName(), KnownClient.HAVE);
        if (failedToSend) {
            System.out.println(
                    String.format("Removing disconnected %s from list", serverThread.getDisplayName()));
            disconnectingServerThread = null;
        } <- #100-172 in (failedToSend)
        return failedToSend;
    });
    <- #102-274 clientsInRoom.values().removeIf
    disconnectingServerThread.disconnect();
    <- #103-174 (disconnectingServerThread.getClientId() != null)
    autoCleanup();
}
```

Room.java forwards disconnect

```
/***
 * Terminates the server-side of the connection
 */
//is18 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%

```
Project
├── Client
│   └── Client.java
├── Common
│   ├── CommonException.java
│   ├── CommonMessagePayload.java
│   ├── CommonProtocol.java
│   ├── Payload.java
│   ├── PayloadType.java
│   ├── RoomAction.java
│   ├── TextFix.java
│   └── User.java
└── Exception
    ├── CustomMTIMException.java
    ├── DuplicateRoomException.java
    └── RoomNotFoundException.java
└── Server
    ├── RoomServerException.java
    ├── Room.java
    ├── RoomType.java
    └── ServerThread.java
```

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

Milestone1 Baseline #6

1+ Merged

lurasofia544 merged 1 commit into main from Milestone1 3 hours ago

Conversation 0

Commits 1

Checks 0

Files changed 16



lurasofia544 commented 3 hours ago

No description provided.



PR Milestone1 Baseline

Owner ...

lurasofia544

2 · lurasofia544 merged commit 7e88866 into main 3 hours ago

Revert

Accidentally closed the merge before adding the comments

Added Milestone1 Comments #7

1+ Merged

lurasofia544 merged 1 commit into main from Milestone1 now

Conversation 0

Commits 1

Checks 0

Files changed 16



lurasofia544 commented 1 minute ago

No description provided.



PR Milestone1 Comments

Owner ...

lurasofia544

2 · lurasofia544 merged commit 7e88866 into main now

Revert

#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/lurasofia544/lsl8-IT114003/>



URL

<https://github.com/lurasofia544/>



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

Projects · Isl8-IT114-003

3 hrs 47 mins over the Last 7 Days in Isl8-IT114-003 under all branches. ⏱



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