

Submission Worksheet

Submission Data

Course: IT114-450-M2025

Assignment: IT114 Milestone 1

Student: Graham B. (gb373)

Status: Submitted | **Worksheet Progress:** 100%

Potential Grade: 10.00/10.00 (100.00%)

Received Grade: 0.00/10.00 (0.00%)

Started: 6/30/2025 3:33:44 PM

Updated: 6/30/2025 7:33:51 PM

Grading Link: <https://learn.ethereallab.app/assignment/v3/IT114-450-M2025/it114-milestone-1/grading/gb373>

View Link: <https://learn.ethereallab.app/assignment/v3/IT114-450-M2025/it114-milestone-1/view/gb373>

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

```
graham@DESKTOP-TIERSKK4: MINGW64 ~/repo/gb373-11114-458 (Milestone1)
$ javac Project\Server.Server.java
error: file not found: Project\Server.Server.java
Usage: javac <options> <source files>
use --help for a list of possible options

graham@DESKTOP-TIERSKK4: MINGW64 ~/repo/gb373-11114-458 (Milestone1)
$ java Project.Server.Server 3001
graham@DESKTOP-TIERSKK4: MINGW64 ~/repo/gb373-11114-458 (Milestone1)
> $ java Project.Server.Server 3001
Server: Starting
Server: Listening on port 3001
Room[lobby]: Created
Server: Created new Room lobby
Server: Waiting for next client
□
```

Output

```
    /**
     * Starts the server and listens on the specified port and listens for incoming client connections.
     */
    private void start(int port) {
        this.port = port;
        this.isRunning = true;
        info("Listening on port " + this.port);
        // Accept client connection
        try {
            ServerSocket serverSocket = new ServerSocket(port);
            createRooms();
            connect(); // Create the room when listening
            while (true) { // Waiting for next client
                Socket incomingClient = serverSocket.accept(); // Blocking call to wait for a client connection
                info("Client connected");
                // Upon connection establishment, pass it to the room to handle their connection
                if (this.rooms.size() == 0) {
                    ServerThread serverThread = new ServerThread(incomingClient, this.createServerThreadInListener);
                    serverThread.start();
                    // Note: don't add the ServerThread reference to our connectToClients map
                    // as it will still be running
                } else {
                    for (Room room : rooms) {
                        if (room.getConnectedClients().contains(incomingClient)) {
                            room.connectClient();
                            break;
                        }
                    }
                }
            }
        } catch (IOException e) {
            error("Error starting server");
        }
    }
```

Code referencing Section 1

≡, Part 2:

Progress: 100%

Details:

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

Your Response:

The server uses a socket to connect to port 3001. It then uses a while loop to continuously check for client connections. Then, when the client connects, a new thread is created and puts the client into the default lobby.



Saved: 6/30/2025 3:36:28 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

A screenshot of a terminal window showing the server's response to multiple client connections. The server is listening on port 3001 and has accepted three clients. The output shows the server sending messages to clients and receiving responses from them. The terminal window has a yellow 'X' icon in the top right corner.

Server Output for section 2

Three screenshots of terminal windows showing the server's response to multiple client connections. Each window displays a different set of client interactions. The windows have yellow 'X' icons in the top right corner.

3 clients connected to server

```
//UCID: 61079
//Doku: 6/30/2025
//Summary: Starts the server on the specified port and listens for incoming client connections.
private void start(int port) {
    try {
        // Starts listening on port "port"
        info("Listening on port: " + port);
        // Accepts client connections from the specified port
        UnixServerSocket serverSocket = new UnixServerSocket(port);
        createThreads((ServerThread) serverSocket.accept()); // Create the first client thread
        while (true) { // Waiting for next client
            ServerIncomingClient serverIncomingClient = serverSocket.accept(); // Blocking call until a client connection is received ("Client connected")
            info("Client connected");
            if (serverIncomingClient != null) { // If the client is initialized
                ServerThread serverThread = new ServerThread(serverIncomingClient); // This creates a ServerThread instance
                if (serverThread != null) { // If the thread is initialized correctly, it will manage the connection and
                    serverThread.start(); // and the ServerThread reference to our connectableClient
                }
            }
        }
    } catch (IOException e) {
        error("IOException exception: " + e);
        e.printStackTrace();
    } finally {
        info("Closing server socket");
    }
}
```

Code for section 2 showing is running

```
//UCID: 61079
//Doku: 6/30/2025
//Summary: A branch method that is invoked when a ServerThread has finished
//The method takes a unique Client ID, which is then used by the ServerThread, and adds the
//Client to the lobby queue.
private void onServerThreadInitialized(ServerThread serverThread) {
    String currentClientId = serverThread.getCurrentClientId();
    int newClientId = Math.max(currentClientId, 100);
    serverThread.setClientId(newClientId);
    serverThread.send("Client initialized"); // Notify the client of the state
    // Adding the client to the lobby
    info(String.format("New Client added", serverThread.getDisplayName()));
    try {
        info(String.format("Server Thread initialized", serverThread.getDisplayName()));
        info(String.format("New client in lobby", serverThread.getDisplayName()));
    } catch (IOException e) {
        error("IOException exception: " + e);
        e.printStackTrace();
    }
}
```

Code for setting client ids

Saved: 6/30/2025 3:49:53 PM

Part 2:

Progress: 100%

Details:

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

Your Response:

The server uses the `is` running to make connections for the client and loops it for each client. This means that the server creates a new thread for each client. With this, the server can run multiple clients at once with different threads.

Saved: 6/30/2025 3:49:53 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)

```
Received message[0]: Created room myroom
Server: Created new Room myroom
Server: Removing client from previous Room lobby
Thread[1]: Sending to client: Payload[ROOM_LEAVE] Client Id [1] Message: (Room lobby) ClientName: [Oswaldo]
Thread[2]: Sending to client: Payload[ROOM_LEAVE] Client Id [2] Message: (Room lobby) ClientName: [Oswaldo]
Thread[3]: Sending to client: Payload[MESSAGE] Client Id [1] Message: (Room lobby) ClientName: [Oswaldo] You left the room
Thread[4]: Sending to client: Payload[ROOM_LEAVE] Client Id [3] Message: (Room lobby) ClientName: [Oswaldo] You left the room
Thread[5]: Sending to client: Payload[ROOM_LEAVE] Client Id [4] Message: (Room lobby) ClientName: [Oswaldo] You left the room
Thread[6]: Sending to client: Payload[ROOM_LEAVE] Client Id [5] Message: (Room lobby) ClientName: [Oswaldo] You left the room
Thread[7]: Sending to client: Payload[MESSAGE] Client Id [1] Message: (Room myroom) you joined the room
Thread[8]: Sending to client: Payload[MESSAGE] Client Id [2] Message: (Room myroom) you joined the room
Server: Removing client from previous Room myroom
Thread[9]: Sending to client: Payload[ROOM_LEAVE] Client Id [1] Message: (Room myroom) You left the room
Thread[10]: Removing room myroom
Room[myroom] closed
Thread[11]: Sending to client: Payload[ROOM_JOIN] Client Id [1] Message: (null) ClientName: [Oswaldo]
Thread[12]: Sending to client: Payload[MESSAGE] Client Id [1] Message: (Room myroom) You joined the room
Thread[13]: Sending to client: Payload[MESSAGE] Client Id [2] Message: (null) ClientName: [Oswaldo]
Thread[14]: Sending to client: Payload[ROOM_JOIN] Client Id [2] Message: (null) ClientName: [Oswaldo]
Thread[15]: Sending to client: Payload[MESSAGE] Client Id [3] Message: (null) ClientName: [Oswaldo]
Thread[16]: Sending to client: Payload[MESSAGE] Client Id [4] Message: (null) ClientName: [Oswaldo]
Thread[17]: Sending to client: Payload[MESSAGE] Client Id [5] Message: (null) ClientName: [Oswaldo]
Thread[18]: Sending to client: Payload[MESSAGE] Client Id [1] Message: (Room lobby) uraham joined the room
Thread[19]: Sending to client: Payload[MESSAGE] Client Id [2] Message: (Room lobby) uraham joined the room
Thread[20]: Sending to client: Payload[MESSAGE] Client Id [3] Message: (Room lobby) uraham joined the room
Thread[21]: Sending to client: Payload[MESSAGE] Client Id [4] Message: (Room lobby) uraham joined the room
Thread[22]: Sending to client: Payload[MESSAGE] Client Id [5] Message: (Room lobby) uraham joined the room
||
```

Server side of room creation, joining, and removal.

```
✓/OCLD_gblaz# YOU: 33 minutes ago - Uncommitted changes
✓/Date: 6/14/2025
// Summary: Creates a new room with the specified name if it doesn't already exist.
protected void joinRoom(String name, ServerThread client) throws RoomNotFoundException {
    if (clients.containsKey(name)) {
        throw new DuplicateRoomException(String.format("Room %s already exists.", name));
    }
    Room room = new Room(name);
    rooms.add(room);
    Info("Creating room " + room.getName() + "!");
    System.out.println("Created new Room " + name);
}
1 ✓ 0:00 0:17 previous test failed: expected(String, Room) but was(String, Room)
||
```

create room code

```
✓/OCLD_gblaz# YOU: 33 minutes ago - Uncommitted changes
✓/Date: 6/14/2025
// Summary: Moves a client to the specified room, removing them from their current.
// room if they are already in one.
protected void joinRoom(String name, ServerThread client) throws RoomNotFoundException {
    if (!clients.containsKey(name)) {
        throw new RoomNotFoundException(String.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 = clients.get(name);
    next.addClient(client);
}
1 ✓ 0:00 0:16 previous test failed: expected(String, Room) but was(String, Room)
||
```

join/leave room code

```
✓/OCLD_gblaz# YOU: 33 minutes ago - Uncommitted changes
✓/Date: 6/14/2025
// Summary: Removes a room from the collection of rooms.
protected void removeRoom(Room room) {
    rooms.remove(room.getName().toLowerCase());
    Info(String.format("Removed room %s.", room.getName()));
}
||
```

Remove Room Code

 Saved: 6/30/2025 4:59:45 PM

≡ Part 2:

Progress: 100%

Details:

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

Your Response:

First, the default lobby is called lobby, and this is made when the server thread is initialized. Using createRoom, it creates a new room and checks if the room already exists. Clients can then use joinroom to join the room that has been created. This method handles the joining of rooms and the leaving of rooms by removing them from the previous one and removing them when the leave room is typed. When removeroom is used, this is typically when a room is empty and no client is in it.

 Saved: 6/30/2025 4:59:45 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



Client Connect and Name Output and Successful Connection

```
// Client code
// Summary: Client Command Handler. This handles the /name command
// which outputs the current client name to the user
// Example: /name <name>
// Test: length(<name>) == 0 <br/>
// Response: User's name is <name>. This command initializes or changes the current name. <br/>
// Parameters: <name>
// Returns: User's name
// Author: CustomRBD
// Revision: 1.0
// Date: 2025-03-30
// Description: This command handles the /name command. It takes a string parameter <name> and sets it as the current name. It then returns the current name.
```

Code for /name in Client

```
// Client code
// Summary: Handles the /connect command from the client. This command connects the client to the server.
// Parameters: None
// Returns: Connection status
// Author: CustomRBD
// Revision: 1.0
// Date: 2025-03-30
// Description: This command connects the client to the server. It sends a connection request to the server and waits for a response. If the response is successful, it prints a success message. If it fails, it prints an error message.
```

code for /connect in client

```
// Client code
// Summary: Confirms the connection from the client. Checks if the client is connected.
// Parameters: Client ID (payLoad.get("clientID"))
// Returns: Client ID (payLoad.get("clientID"))
// Author: CustomRBD
// Revision: 1.0
// Date: 2025-03-30
// Description: This command confirms the connection from the client. It checks if the client ID in the payLoad matches the client ID in the system. If they match, it prints a confirmation message and returns the client ID.
```

Code for conformation

 Saved: 6/30/2025 4:55:41 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 a client uses /name, it will get rid of the / and use the name that is given, and set it as the name of the client. For the /connect, the client would connect to the local host and find the port that the server is running on, which in this case is 3001. It then opens a socket for connection 5 to the server, and if it is a success, the client will be notified that it is successful. The confirmation part works like it gets the client ID ready, then processes the client data then prints in green that they are connected to the server.



Saved: 6/30/2025 4:55:41 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

```
Gratuitous-FACTORY-TRAVIS-CI-MINIMAW ~ /tmp/gb52v TT114 458 (idle) [root@localhost ~]
$ ./join.py --client-client
CLIENT Created
CLIENT starting
socket.createSocket()
/room/CREATE
Room[Today] you joined the room
Room[Today] you created the room
Room[Today] Bob123 joined the room
Room[Today] Room1
Room[Today] you left the room
Room[Today] you joined the room
/room/CREATE
Room[Today] you left the room
Room[Today] you joined the room
[]
```

output for client creating room and joining room and sucess for both scenarios

```
Gratuitous-FACTORY-TRAVIS-CI-MINIMAW ~ /tmp/gb52v TT114 458 (idle) [root@localhost ~]
$ ./join.py --client-client
CLIENT Created
CLIENT starting
socket.createSocket()
/room/CREATE
Room[Today] you joined the room
Room[Today] you created the room
Room[Today] Bob123 joined the room
Room[Today] Room1
Room[Today] you left the room
Room[Today] you joined the room
/room/CREATE
Room[Today] you left the room
Room[Today] you joined the room
[]
```

Function definitions for the room creation and joining commands

```
1 view AF_ClientsCreateAndJoinCommand.CREATE_ROOM_Command{ } {
    2 AF_ClientsCreateAndJoinCommand.CREATE_ROOM_Command{ } = new AF_ClientsCreateAndJoinCommand();
    3 AF_ClientsCreateAndJoinCommand.CREATE_ROOM_Command{ } = AF_ClientsCreateAndJoinCommand.create();
    4 AF_ClientsCreateAndJoinCommand.CREATE_ROOM_Command{ } = AF_ClientsCreateAndJoinCommand.create();
    5 AF_ClientsCreateAndJoinCommand.CREATE_ROOM_Command{ } = AF_ClientsCreateAndJoinCommand.create();
    6 AF_ClientsCreateAndJoinCommand.CREATE_ROOM_Command{ } = AF_ClientsCreateAndJoinCommand.create();
    7 AF_ClientsCreateAndJoinCommand.CREATE_ROOM_Command{ } = AF_ClientsCreateAndJoinCommand.create();
    8 AF_ClientsCreateAndJoinCommand.CREATE_ROOM_Command{ } = AF_ClientsCreateAndJoinCommand.create();
    9 AF_ClientsCreateAndJoinCommand.CREATE_ROOM_Command{ } = AF_ClientsCreateAndJoinCommand.create();
}
```

code for both commands

```
/* Room.java: 16/5/25
 * Name: P.M. Muthu
 * Summary: Handles room action to the server, such as creating, joining or
 * leaving a room.
 */
public void handleRoomAction(String command, RoomUtil roomUtil) throws IOException {
    Payload payload = new Payload();
    payload.setMessage(command);
    switch (command) {
        case RoomUtil.CREATE:
            payload.setPayloadType(PayloadType.ROOM_CREATE);
            break;
        case RoomUtil.JOIN:
            payload.setPayloadType(PayloadType.ROOM_JOIN);
            break;
        case RoomUtil.LEAVE:
            payload.setPayloadType(PayloadType.ROOM_LEAVE);
            break;
        default:
            System.out.println("Received invalid room action: " + command);
    }
    sendServer(payload);
}
```

code for sending any room actions to server

```
/* RoomUtil.java: 16/5/25
 * Name: P.M. Muthu
 * Summary: Handles room action to the server, such as creating, joining or
 * leaving a room.
 */
public void handleRoomAction(String command) throws IOException {
    Payload payload = new Payload();
    payload.setMessage(command);
    if (isConnected()) {
        sendServer(payload);
    } else {
        System.out.println("User is not connected to server. Only user messages themselves" +
                           " receives their messages. The user and message from others will be discarded." +
                           " User: " + nickname);
    }
}
```

code for sending payloads to server

 Saved: 6/30/2025 5:30:06 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:

The /createroom and /joinroom work with the client command where the client puts it in, then it sends the payload type to the corresponding command, where it will send it through the sendToServer method. Which will work if the client is connected. If the server receives the payload, it will send the messages back and create the room or have the client join the room.

 Saved: 6/30/2025 5:30:06 PM

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

Progress: 100%

☰ Task #1 (1 pt.) - Evidence

Progress: 100%

▣ Part 1:

Progress: 100%

Details:

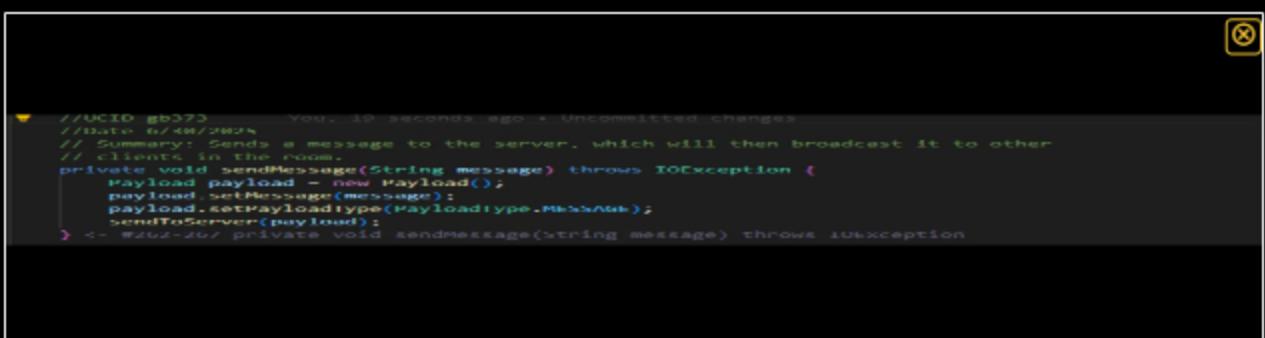
- Show the terminal output of a few messages from each of 3 clients
- Include examples of clients grouped into other rooms
- Show the relevant snippets of code that handle the message process from client to server-side and back



Three terminal windows showing client messages. The first window shows a client named 'Client 1' sending a message. The second window shows a client named 'Client 2' sending a message. The third window shows a client named 'Client 3' sending a message.

```
Client 1: 
Client 2: 
Client 3: 
```

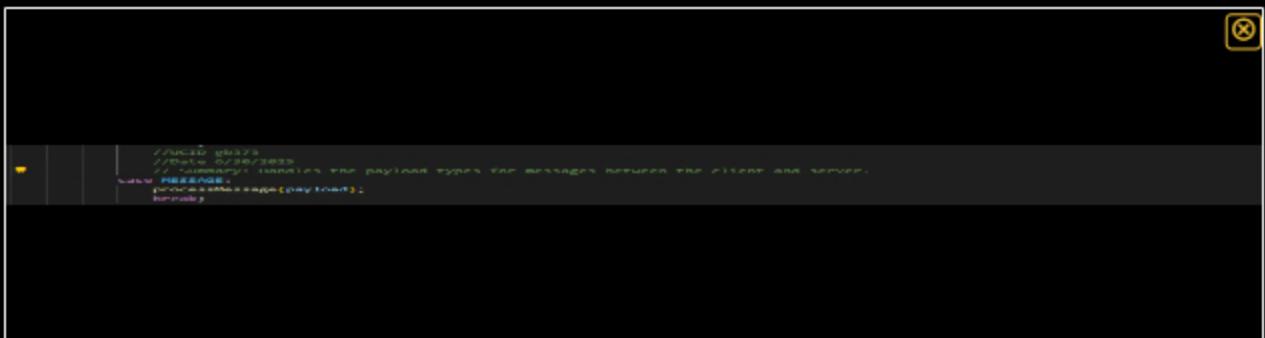
Messages from clients and also some that are in different rooms



Java code for client-side message sending. The code defines a method `sendMessage` that sends a message to the server, which then broadcasts it to other clients in the room. The code uses a `Payload` object to store the message payload.

```
// UCID: gb323 You, 39 seconds ago + Uncommitted changes
// Date: 6/26/2025
// Summary: Sends a message to the server, which will then broadcast it to other
// clients in the room.
private void sendMessage(String message) throws IOException {
    Payload payload = new Payload();
    payload.setMessage(message);
    payload.setPayloadType(PayloadType.Message);
    sendToServer(payload);
}
<- #262-267 private void sendMessage(String message) throws IOException 
```

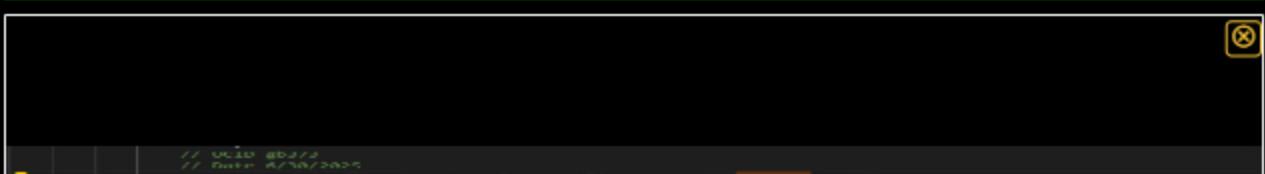
code for client side sending messages



Java code for payload handling. The code defines a `Payload` class with a static method `processMessage` that processes messages between the client and server. The code also includes a `MESSAGE` constant and a `payload` variable.

```
// UCID: gm223
// Date: 6/26/2025
// Summary: Handles the payload types for messages between the client and server.
public static void processMessage(Payload payload) {
    // ...
}
```

payload code for messages



Java code for client-side message sending. The code defines a method `sendMessage` that sends a message to the server, which then broadcasts it to other clients in the room. The code uses a `Payload` object to store the message payload.

```
// UCID: gb223
// Date: 6/26/2025 
```

case incremental: HandshakeCase when a client sends a message to the server, break;

code for the case of message server side

```
/** UDP UDPServer
 * Summary: sends a message to all clients in the room, handling disconnections
 * and sending thread safety
 * preconditions: internal static relay<CommonThread> sender, String message, int
 *    if (CLOSING) { // blank action if room isn't running
 *      return;
 *    }
 *    // never any method changes to the message must be done before this line
 *    String formattedMessage = format("formatRoom[%s]", message);
 *    final long senderId = sender == null ? Constants.DEFAULT_CLIENT_ID : sender.getSenderId();
 *    // Never FormatAndSendMessage result be fired (or effectively final) since update
 *    // would then update the value in the variable but also update the value in the list
 *    final String formattedMessage = String.format("%s %s", senderId, message);
 *    loop over clients and send out the message: receiver instance of message
 *    via its send()
 *    Note: this uses a lambda expression for each item in the ArrayList collection,
 *    it's some way we can safely remove items during iteration
 *    After that, if the client disconnects, we have to remove it
 *    final ClientRelay<CommonThread> receiver = receiverList.remove();
 *    receiver.synchronized void relay(CommonThread sender, String message) {
 *      // can't use Collection.remove because it's not synchronized
 *      synchronized (receiver) {
 *        receiver.notifyAll();
 *        receiver.setLastActivityTime();
 *      }
 *      // remove the client from the list
 *      CommonThread.removeByName(senderName);
 *    }
 *  
```

code for sending message to all clients in a certain room.



Saved: 6/30/2025 5:58:26 PM

Part 2:

Progress: 100%

Details:

- Briefly explain how the message code flow works

Your Response:

When a client enters a message, it calls a send message payload with the type Message. On the server side, the server thread processes the payload and then identifies it as a message. It will then call the handleMessage method in which room that the client is currently in. For the room code, it uses a relay to send the message to all the clients in the same room as the sender.



Saved: 6/30/2025 5:58:26 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 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

This screenshot shows a terminal window with four panes. The top-left pane displays a sequence of messages from a client, including "SESSION ID: 1234567890" and "SESSION NAME: testSession". The top-right pane shows a response from the server. The bottom-left pane is mostly blank, while the bottom-right pane contains server log entries.

output for a client disconnecting

This screenshot shows a terminal window with four panes. The top-left pane displays a sequence of messages from a client. The top-right pane shows a response from the server indicating a connection has been closed. The bottom-left and bottom-right panes contain server log entries.

output for server disconnecting

This screenshot shows a terminal window with four panes. The top-left pane displays a sequence of messages from a client. The top-right pane shows a response from the server indicating a connection has been closed. The bottom-left and bottom-right panes contain server log entries.

output of reconnection after server comes back online

This screenshot shows a terminal window with four panes. The bottom-left pane displays a snippet of Java code. The code includes imports for `java.util.concurrent`, `java.util.function`, and `java.util.concurrent.CompletableFuture`. It defines a class `Client` with a method `handleDisconnect` that takes a `CompletableFuture<String>` and a `String` parameter. The method checks if the command is `DISCONNECT` and sets a flag `wasCommand = true`.

```
/*
 * UCED - MR#74
 * // handles the DISCONNECT command to DISCONNECT from the server
 */
public void handleDisconnect(CompletableFuture<String> future, String command) {
    if (command.equals(Command.DISCONNECT)) {
        wasCommand = true;
    }
}
```

code for command on client side for disconnect

```

// LCTD ghs75
// Date 6/30/2025
// Summary: Sends a disconnect payload to the server, indicating that the
// client wishes to disconnect.
private void sendDisconnect() throws IOException {
    payLoad.setPayloadType(PayLoadType.DISCONNECT);
    payLoad.setPayLoadType(PayLoadType.DISCONNECT);
    sendToServer(payLoad);
}

// #250-254 private void sendDisconnect() throws IOException

```

sends disconnect payload to server from client side

```

// LCTD ghs75
// Date 6/30/2025
// Summary: Handles incoming message from the server and handles disconnections.
private void listenToServer() {
    try {
        while (!isRunning || !isConnected) {
            if (isConnected) {
                Thread.sleep(1000); // Blocking read
            } else {
                processPayload(disconnected);
            }
        }
        if (isRunning & !isConnected) {
            System.out.println("Server disconnected");
            break;
        }
    } catch (ClassNotFoundException | ClassNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        if (isRunning) {
            System.out.println("Connection dropped");
            processDisconnect();
        }
    } finally {
        closeServerConnection();
    }
}

// #255-256 private void listenToServer()

```

code for messages to do with disconnections

```

// LCTD ghs75
// Date 6/30/2025
// Summary: Closes the server connection and unregisters the connection, sending a disconnect
// shutdown.
private void closeServerConnection() {
    try {
        if (out != null) {
            System.out.println("Closing output stream");
            out.close();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    try {
        if (in != null) {
            System.out.println("Closing input stream");
            in.close();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    try {
        if (server != null) {
            server.shutdown();
            System.out.println("Closed socket");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// #257-258 private void closeServerConnection()

```

code for closing the client connection

```

// LCTD ghs75
// Date 6/30/2025
// Summary: Disconnects from the server and sends a disconnect frame from generic message type disconnect or terminate.
private void disconnect() {
    Room room = Room.getRoom();
    room.getRoom().sendShutdownFrame("Termination");
    room.disconnect();
}

// #259-260 private void disconnect()

```

code on server side for shutdown hook for JVM shutdown

```

// LCTD ghs75
// Date 6/30/2025
// Summary: shuts down the server by disconnecting all clients and removing rooms.
private void shutdown() {
    try {
        // chose removeif over for each to avoid potential
        // concurrentmodificationexception
        // #261-262 removes all the server to remove themselves
        rooms.values().removeIf(room -> {
            room.disconnect();
            return true;
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// #263-264 private void shutdown()

```

code for server side for when shutdown happens, it disconnects clients and removes rooms.



Saved: 6/30/2025 6:28:40 PM

≡, Part 2:

Progress: 100%

Details:

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

Your Response:

On the client side, the client uses /disconnect command to start disconnect. When this command is made, it closes client streams, socket connections, and any threads that are associated with it or listening. It will then send a disconnect payload to the server, and then message to the user that the connection has been dropped. On the server side, when the server disconnects, it registers the hook for the JVM shutdown. It then uses the shutdown method to close all rooms and disconnect every client, but the clients are still active. Messages are sent to confirm shutdown, and the server will be disconnected.



Saved: 6/30/2025 6:28:40 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%



proper workspace structure for my project folder



Saved: 6/30/2025 6:29:58 PM

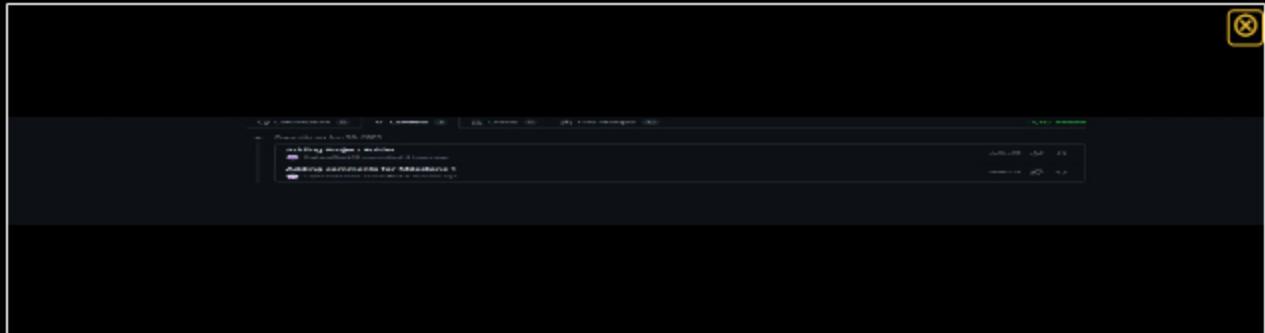
- ≡ Task #2 (0.25 pts.) - Github Details

Part 1:

Progress: 100%

Details:

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



commits for milestone 1



Saved: 6/30/2025 6:32:09 PM

Part 2:

Progress: 100%

Details:

Include the link to the Pull Request (should end in `/pull/*`)

URL #1

<https://github.com/GrahamBlack10/gb373-IT114-450/pull/6>



URL

<https://github.com/GrahamBlack1>



Saved: 6/30/2025 6:32:09 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 - gb373-IT114-450

2 hrs 32 mins over the last 7 days in gb373-IT114-450 under all branches.

overall time

```
git log --oneline --graph --all
```



individual time



Saved: 6/30/2025 6:37:08 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 learn more about sockets and the different features that they can do. I learned more about advanced features of the server, as well as how clients can create their rooms and the server can hold them. I found out about how clients can be created with their thread and their own name to have differences from other clients. I also learn about how servers can run multiple clients and how they can disconnect, but the clients will still be active.



Saved: 6/30/2025 7:26:39 PM

≡ Task #2 (0.33 pts.) - What was the easiest part of the assignment?

Details:

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

Your Response:

The easiest part of this assignment was understanding the connection between client and server. What I mean by that is how they connected through a port and how the code did it. This was mainly because this was in the previous socket parts. I also found that the payloads and how they are created, and each type, were easy to understand, and the concept behind them was easy to understand, too.



Saved: 6/30/2025 7:31:08 PM

=> Task #3 (0.33 pts.) - What was the hardest part of the assignment?

Details:

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

Your Response:

The hardest part of the assignment was understanding exceptions and the differences between them. I found that I had trouble understanding why there was an exception when the client disconnected, and I had to learn more about it. Also, I had some trouble figuring out the process of sending messages. This included the process of sending messages between client and server, and how the room was put into effect, meaning where a sender would only send a message to clients in the same room.



Saved: 6/30/2025 7:33:51 PM