

# Submission Worksheet

**CLICK TO GRADE**

<https://learn.ethereallab.app/assignment/IT114-005-F2024/it114-module-5-project-milestone-1/grade/kr553>

Course: IT114-005-F2024

Assignment: [IT114] Module 5 Project Milestone 1

Student: Kush R. (kr553)

## Submissions:

Submission Selection

1 Submission [submitted] 10/21/2024 11:09:02 PM

## Instructions

[^ COLLAPSE ^](#)

Overview Video: <https://youtu.be/A2yDMS9TS1o>

1. Create a new branch called Milestone1
2. At the root of your repository create a folder called Project if one doesn't exist yet
  1. You will be updating this folder with new code as you do milestones
  2. You won't be creating separate folders for milestones; milestones are just branches
3. Copy in the code from Sockets Part 5 into the Project folder (just the files)
  2. <https://github.com/MattToegel/IT114/tree/M24-Sockets-Part5>
4. Fix the package references at the top of each file (these are the only edits you should do at this point)
5. Git add/commit the baseline and push it to github
6. Create a pull request from Milestone1 to main (don't complete/merge it yet, just have it in open status)
7. Ensure the sample is working and fill in the below deliverables 1. Note: Don't forget the client commands are /name and /connect
8. Generate the output file once done and add it to your local repository
9. Git add/commit/push all changes
10. Complete the pull request merge from the step in the beginning
11. Locally checkout main
12. git pull origin main

Branch name: Milestone1

**Group**

Group: Start Up

Tasks: 2

Points: 3

[^ COLLAPSE ^](#)**Task**

Group: Start Up

Task #1: Start Up

Weight: ~50%

Points: ~1.50

[^ COLLAPSE ^](#)**i Details:**

**Important:** Code screenshots should be fairly concise (try to show only the sections of code relevant to the question)

Capturing all possible code (i.e., including a lot of irrelevant code) can lead to a reduced grade.

Columns: 1

**Sub-Task**

Group: Start Up

Task #1: Start Up

Sub Task #1: Show the Server starting via command line and listening for connections

**Task Screenshots**

Gallery Style: 2 Columns

[4](#)   [2](#)   [1](#)

```
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL SEARCH ERROR
Kushal@Kushals-MacBook-Pro:~/Projects/Kotlin-1.3.16-MINIS (HelloWorld)
$ java Project.Server
Server starting
Listening on port: 3000
Room[Lobby] created
Created new Room lobby
Waiting for next client
[]
```

Server listing connections

**Caption(s) (required) ✓**Caption Hint: *Describe/highlight what's being shown***Sub-Task**

Group: Start Up

Task #1: Start Up

Sub Task #2: Show the Server Code that listens for connections

**Task Screenshots**

4 2 1

```

//socket server code
private void start(int port) {
    this.port = port;
    // server listening
    System.out.println("listening on port " + this.port);
    try (ServerSocket serverSocket = new ServerSocket(port)) {
        createClientSocket(serverSocket); // create the first socket
        while (true) {
            System.out.println("Waiting for next client");
            Socket incomingClient = serverSocket.accept(); // blocking action, waits for a client
            System.out.println("Client connected");
            // wrap socket in a ServerThread, pass a callback to notify the Server they're
            // initialized
            ServerThread aClient = new ServerThread(incomingClient, this::onClientInitialized);
            aClient.start(); // start thread (typically an external entity manages the lifecycle and we
            // don't have the thread start itself)
        }
    } catch (IOException e) {
        System.out.println("Error accepting connection");
        e.printStackTrace();
    } finally {
        shutdown();
        System.out.println("Closing server socket");
    }
}

```

this is the server code that initializes on start and listnes any connections

**Caption(s) (required)** ✓

**Caption Hint:** *Describe/highlight what's being shown (uid/date must be present)*

## ≡, Task Response Prompt

*Briefly explain the code related to starting up and waiting for connections*

**Response:**

A server opens a ServerSocket and provides the service of waiting for connection requests from a client. Each connection makes a new ServerThread that will handle client requests and therefore it includes multiple clients. Frequency issues during connection are recorded while the server goes on to accept other clients.

**Sub-Task**

Group: Start Up

100%

Task #1: Start Up

Sub Task #3: Show the Client starting via command line

## ❑ Task Screenshots

4 2 1



KushR@acerNitro5: MINGW64 ~/Projects/kr553-TT114-R05 (Milestone1)\$ \$ java Project.Client Client Created Client starting Waiting for input

Client starting

**Caption(s) (required)** ✓

**Caption Hint:** *Describe/highlight what's being shown*

**Sub-Task**

Group: Start Up

100%

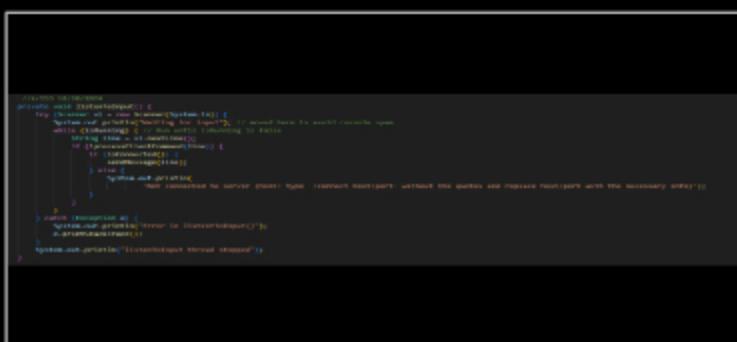
Task #1: Start Up

Sub Task #4: Show the Client Code that prepares the client and waits for user input

## Task Screenshots

Gallery Style: 2 Columns

4 2 1



```
private void startClient() {
    try {
        System.out.println("Starting the client..."); // Used here to avoid character issues
        Scanner scanner = new Scanner(System.in);
        while (true) {
            IsRunning = true;
            if (scanner.hasNextLine()) {
                if (scanner.nextLine().equals("connect")) {
                    connect();
                } else if (scanner.nextLine().equals("quit")) {
                    quit();
                }
            }
        }
    } catch (Exception e) {
        System.out.println("An error occurred while connecting to the server. Please try again later or report the issue to the administrator.");
    }
}
```

this function prepares client and wait for the input.

**Caption(s) (required)** ✓

Caption Hint: *Describe/highlight what's being shown (ucid/date must be present)*

## Task Response Prompt

*Briefly explain the code/logic/flow leading up to and including waiting for user input*

Response:

By use of `CompletableFuture` the client's `start()` method invokes `listenToInput()` that reads user input under the help of `Scanner`. Input such as `/connect` is executed by the bot whilst all other inputs are forwarded as messages up to the moment the client disconnects (`IsRunning` is `False`).

End of Task 1

Task



Group: Start Up

Task #2: Connecting

Weight: ~50%

Points: ~1.50

 COLLAPSE 

 Details:

Important: Code screenshots should be fairly concise (try to show only the sections of code relevant to the question)

Capturing all possible code (i.e., including a lot of irrelevant code) can lead to a reduced grade.

Columns: 1

Sub-Task

Group: Start Up

Task #2: Connecting

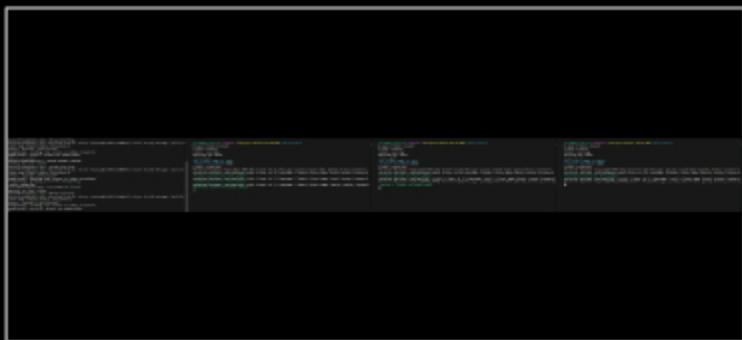
Sub Task #1: Show 3 Clients connecting to the Server

## Task Screenshots

## Task Screenshots

Gallery Style: 2 Columns

4 2 1

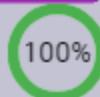


3 clients connected to the server

Caption(s) (required) ✓

Caption Hint: *Describe/highlight what's being shown*

Sub-Task



Group: Start Up

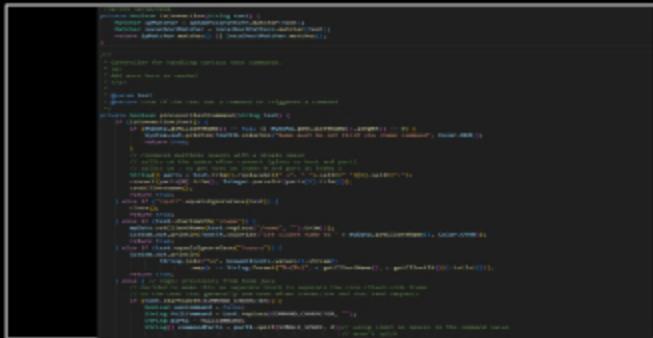
Task #2: Connecting

Sub Task #2: Show the code related to Clients connecting to the Server (including the two needed commands)

## Task Screenshots

Gallery Style: 2 Columns

4 2 1



The /connect command is validated with regex before calling connect().

```
//ke553 10/20/2024
private boolean connect(String address, int port) {
    try {
        server = new Socket(address, port);
        // channel to send to server
        out = new ObjectOutputStream(server.getOutputStream());
        // channel to listen to server
        in = new ObjectInputStream(server.getInputStream());
        System.out.println("Client connected");
        // Use CompletableFuture to run listenToServer() in a separate thread
        CompletableFuture.runAsync(this::listenToServer);
    } catch (UnknownHostException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return isConnected();
}
```

It sets up the socket and input/output streams for communication.

Caption(s) (required) ✓

Caption Hint: *Describe/highlight what's being shown (ucid/date must be present)*

## Task Response Prompt

*Briefly explain the code/logic/flow*

Response:

The client connects to the server via /connect [host]:[port]. The command string is validated and checked in the 'processClientCommand()' function while socket connection in the server is done by 'connect()'. Upon connection sendClientName() is invoked and upon entering/quit, the connection is closed with close().

Group



Group: Communication

Tasks: 2

Points: 3

[^ COLLAPSE ^](#)

Task



Group: Communication

Task #1: Communication

Weight: ~50%

Points: ~1.50

[^ COLLAPSE ^](#)

**i** Details:

Important: Code screenshots should be fairly concise (try to show only the sections of code relevant to the question)



Capturing all possible code (i.e., including a lot of irrelevant code) can lead to a reduced grade.

Columns: 1

Sub-Task



Group: Communication

Task #1: Communication

Sub Task #1: Show each Client sending and receiving messages

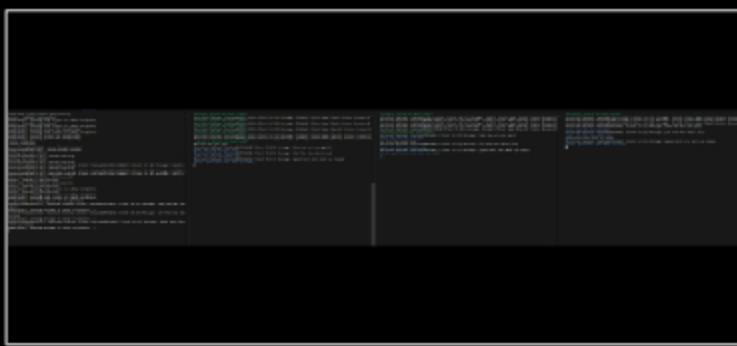
## Task Screenshots

Gallery Style: 2 Columns

4

2

1



On the server side and all the clients side all can hear  
messages

**Caption(s) (required) ✓**

Caption Hint: *Describe/highlight what's being shown*

[See Task](#)

**Sub-Task**

Group: Communication

Task #1: Communication

Sub Task #2: Show the code related to the Client-side of getting a user message and sending it over the socket

**Task Screenshots**

Gallery Style: 2 Columns

4

2

1

```
private void startServer() {
    try {
        System.out.println("Server is listening for clients");
        while (true) {
            if (serverSocket != null) {
                if (serverSocket.isClosed()) {
                    System.out.println("Server has closed");
                } else {
                    System.out.println("Not connected to server. Object type " + serverSocket.getRemoteSocketAddress());
                }
            }
        }
    } catch (IOException e) {
        System.out.println("Error in listening(" + e.getMessage() + ")");
    }
    System.out.println("Client accepted(" + clientSocket.getInetAddress() + ")");
}
```

```
private void sendMessage(String message) {
    Payload p = new Payload();
    p.setPayloadType(PayloadType.MESSAGE);
    p.setMessage(message);
    send(p);
}
```

This method listens for user input through the console and processes the input.

This method packages the user message into a Payload object, sets the payload type to MESSAGE, and then sends it to the server

```
private void send(Payload p) {
    try {
        out.writeObject(p);
        out.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
// end send methods
```

The send() method writes the Payload object to the ObjectOutputStream connected to the server.

**Caption(s) (required)** ✓

Caption Hint: *Describe/highlight what's being shown (ucid/date must be present)*

**=, Task Response Prompt**

*Briefly explain the code/logic/flow involved*

Response:

The listenToInput() method takes care of checking the user input, whether it be a connection string, such as connect, or a message to be sent to the connected server, while sendMessage() builds up a Payload of type MESSAGE and forwards it to the server so that the door to constant communication between the client and server is open.

**Sub-Task**

Group: Communication

Task #1: Communication

Sub Task #3: Show the code related to the Server-side receiving the message and relaying it to each connected Client

**Task Screenshots**

4 2 1

```
//kris 10/20/2024
@Override
protected void processPayload(Payload payload) {
    try {
        switch (payload.getPayloadType()) {
            case CLIENT_CONNECT:
                ConnectionPayload cp = (ConnectionPayload) payload;
                cp.setClientName(ep.getClientName());
                break;
            case MESSAGE:
                CurrentRoom.sendMessage(this, payload.getMessage());
                break;
            case ROOM_CREATE:
                currentRoom.handleCreateRoom(this, payload.getMessage());
                break;
            case ROOM_JOIN:
                currentRoom.handleJoinRoom(this, payload.getMessage());
                break;
            case DISCONNECT:
                currentRoom.disconnect(this);
                break;
            default:
                break;
        }
    } catch (Exception e) {
        System.out.println("Could not process Payload: " + payload);
        e.printStackTrace();
    }
}
```

The server calls sendMessage() to broadcast MESSAGE payloads to other connected clients.

```
//kris 10/20/2024
public boolean sendMessage(long senderId, String message) {
    Payload p = new Payload();
    p.setClientId(senderId);
    p.setMessage(message);
    p.setPayloadType(PayloadType.MESSAGE);
    return send(p);
}
```

sendMessage() forwards the message to all clients except the original sender.

### Caption(s) (required) ✓

Caption Hint: *Describe/highlight what's being shown (ucid/date must be present)*

## Task Response Prompt

*Briefly explain the code/logic/flow involved*

Response:

The ServerThread interacts with client's messages and pass it to the room's sendMessage() method. The room also informs the other clients of the messages in the room excluding the sender in real time hence the communication in the room.

### Sub-Task

Group: Communication

100%

Task #1: Communication

Sub Task #4: Show the code related to the Client receiving messages from the Server-side and presenting them

## Task Screenshots

4 2 1

```
//kris 10/20/2024
private void listenToServer() {
    try {
        while (!isRunning && isConnected()) {
            Payload fromServer = (Payload) in.readObject(); // blocking read
            if (fromServer != null) {
                if (fromServer instanceof RoomServer)
                    processPayload((RoomServer) fromServer);
                else {
                    System.out.println("Server disconnected");
                    break;
                }
            } else {
                System.out.println("Error reading object as specified type: " + cce.getMessage());
                cce.printStackTrace();
            }
        }
        if (!isRunning)
            System.out.println("Connection dropped");
    } finally {
        closeServerConnection();
    }
    System.out.println("ListenToServer thread stopped");
}
```

The method continuously listens, blocking on input, then processes received server messages.

```
//kris 10/20/2024
private void processPayload(Payload payload) {
    try {
        System.out.println("Received payload: " + payload);
        switch (payload.getPayloadType()) {
            case CONNECTION:
                ConnectionPayload cp = (ConnectionPayload) payload;
                processClientInfo(cp.getClientId(), cp.getClientName());
                break;
            case MESSAGE:
                cp = (MessagePayload) payload;
                processClientInfo(cp.getClientId(), cp.getClientName());
                break;
            case ROOM_TYPE:
                cp = (RoomTypePayload) payload;
                processClientInfo(cp.getClientId(), cp.getClientName());
                break;
            case ROOM_MESSAGE:
                cp = (RoomMessagePayload) payload;
                processClientInfo(cp.getClientId(), cp.getClientName(), cp.isConnect());
                break;
            case ROOM_LEAVE:
                cp = (RoomLeavePayload) payload;
                processClientInfo(cp.getClientId(), cp.getClientName(), cp.getReason());
                break;
            case ROOM_MESSAGE_RECEIVED:
                processMessage(payload.getClientId(), payload.getMessage());
                break;
            default:
                break;
        }
    } catch (Exception e) {
        System.out.println("Could not process payload: " + payload);
        e.printStackTrace();
    }
}
```

The client processes payloads, extracts and displays message content if type MESSAGE.

```
//kris 10/20/2024
private void processMessage(long clientId, String message) {
    String name = RoomClients.createKey(clientId);
    RoomClients.get(clientId).get(name).set("name", message);
    System.out.println(String.format("%s: %s", name, message));
    RoomClients.get(clientId).remove(name);
}
```

The `processMessage()` method takes the message and the sender's information, then prints it to the console.

**Caption(s) (required)** ✓

Caption Hint: *Describe/highlight what's being shown (ucid/date must be present)*

## Task Response Prompt

*Briefly explain the code/logic/flow involved*

Response:

The `listenToServer()` method corresponds to the listening of message from the server and; When the server has something to send through the `messageParameter` it gives it a `Payload` before sending it to the `processPayload()` method. If the received message is of type `MESSAGE` then `processMessage()` function message is shown to the client. This configuration means that the client and the server can easily exchange data in real time so messages passed and presented whenever they are received.

End of Task 1

Task

Group: Communication



Task #2: Rooms

Weight: ~50%

Points: ~1.50

[▲ COLLAPSE ▲](#)

**i** Details:

Important: Code screenshots should be fairly concise (try to show only the sections of code relevant to the question)

Capturing all possible code (i.e., including a lot of irrelevant code) can lead to a reduced grade.



Columns: 1

Sub-Task

Group: Communication



Task #2: Rooms

Sub Task #1: Show Clients can Create Rooms

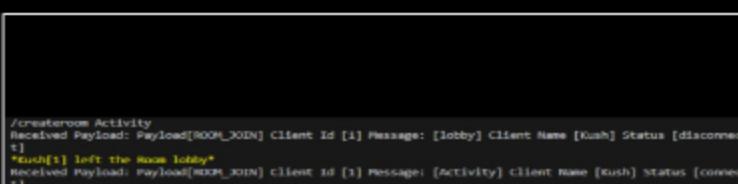
## Task Screenshots

Gallery Style: 2 Columns

4

2

1



"Dash[1] joined the Room Activity"

Created room and joined

**Caption(s) (required)** ✓

Caption Hint: *Describe/highlight what's being shown*

**Sub-Task**

Group: Communication

Task #2: Rooms

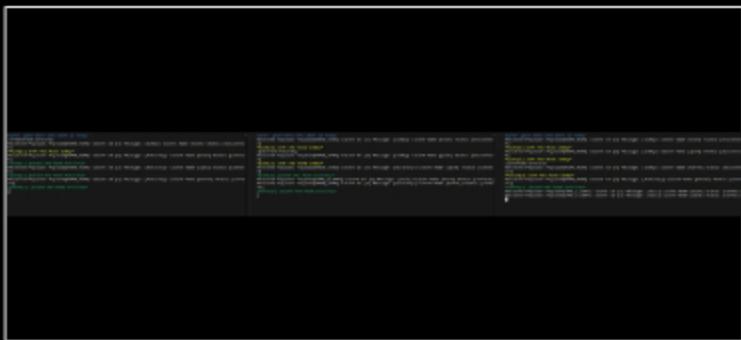
Sub Task #2: Show Clients can Join Rooms (leave/join messages should be visible)

100%

## Task Screenshots

Gallery Style: 2 Columns

4 2 1



Clients joining other room than the lobby created by client 1

**Caption(s) (required)** ✓

Caption Hint: *Describe/highlight what's being shown*

**Sub-Task**

Group: Communication

Task #2: Rooms

Sub Task #3: Show the Client code related to the create/join room commands

100%

## Task Screenshots

Gallery Style: 2 Columns

4 2 1

```
String[] commandParts = parts.split(SINGLE_SPACE, 2); // string limit so spaces in the command value
final String command = commandParts[0];
final String commandValue = commandParts.length > 2 ? commandParts[1] : "";
switch (command) {
    case CREATE_ROOM:
        sendCreateRoom(commandValue);
        wasCommand = true;
        break;
    case JOIN_ROOM:
        sendJoinRoom(commandValue);
        wasCommand = true;
        break;
    // Note: these are to disconnect, they're not for changing room
    case DISCONNECT:
    case LOGOUT:
    case LOGON:
        disconnect();
        wasCommand = true;
        break;
}
return wasCommand;
```

```
private final String CREATE_ROOM = "createroom";
private final String JOIN_ROOM = "joinroom";
```

Client code for create and join rooms

initialized here

**Caption(s) (required)** ✓

Caption Hint: *Describe/highlight what's being shown (ucid/date must be present)*

## ≡ Task Response Prompt

Briefly explain the code/logic/flow involved

Response:

We will also provide in the `processClientCommand()` method the control for such client commands provided for example with `/createroom` or `/joinroom`. Depending upon the command it sends either `sendCreateRoom()` or `sendJoinRoom()` which in return creates a Payload consisting of name of room which is then send to the server.

Sub-Task

Group: Communication

100%

Task #2: Rooms

Sub Task #4: Show the ServerThread/Room code handling the create/join process

## ■ Task Screenshots

Gallery Style: 2 Columns

4

2

1

```
public synchronized void processPayload(Payload payload) {
    try {
        switch (payload.getPayloadType()) {
            case CLIENT_COMMAND:
                ConnectionPayload cp = (ConnectionPayload) payload;
                setClientName(cp.getClientName());
                break;
            case MESSAGE:
                Room room = sendMessage(this, payload.getMessage());
                break;
            case ROOM_CREATE:
                createRoom.handleCreateRoom(this, payload.getMessage());
                break;
            case ROOM_JOIN:
                createRoom.handleJoinRoom(this, payload.getMessage());
                break;
            case DISCONNECT:
                createRoom.disconnect(this);
                break;
            default:
                break;
        }
    } catch (Exception e) {
        System.out.println("Could not process Payload: " + payload);
        e.printStackTrace();
    }
}
```

1. Handling Room Creation (ROOM\_CREATE):

```
// receive data from ServerThread
protected void handleCreateRoom(ServerThread sender, String room) {
    if (!Server.INSTANCE.createRoom(room)) {
        sender.joinRoom(room, sender);
    } else {
        sender.sendMessage(String.format("Room %s already exists", room));
    }
}

protected void handleJoinRoom(ServerThread sender, String room) {
    if (!Server.INSTANCE.joinRoom(room, sender)) {
        sender.sendMessage(String.format("Room %s doesn't exist", room));
    }
}
```

handling creating and joining room

```
protected synchronized void handleClient(ServerThread client) { [Ctrl (CTRL + T) / F4 (CTRL + 1)]
    if (timedOut) { // block action if main isn't running
        return;
    }
    if (clientsInRoom.containsKey(client.getClientId())) {
        logger.info("Attempting to add a client that already exists in the room");
        return;
    }
    clientsInRoom.put(client.getClientId(), client);
    client.setCurrentRoom(this);

    // modify clients of someone leaving
    sendRoomStatus(client.getClientId(), client.getClientName(), true);
    // sync room state to joiner
    syncRoomList(client);

    infoString = String.format("%s(%s) joined the room(%s)", client.getClientName(), client.getClientId(), getName());
}
```

handles the joining/leaving room messages

Caption(s) (required) ✓

Caption Hint: *Describe/highlight what's being shown (ucid/date must be present)*

## ≡ Task Response Prompt

Briefly explain the code/logic/flow involved

Response:

The `processPayload()` method serves clients' requests of creating a new room or joining an existing room or booth. The `handleCreateRoom()` method creates a new room and sends a message to the client requesting room creation. The `handleJoinRoom()` method assigns the client to an existing room or sends a message if the room is unavailable.

**Sub-Task**

Group: Communication

Task #2: Rooms

Sub Task #5: Show the Server code for handling the create/join process

100%

**Task Screenshots**

Gallery Style: 2 Columns

4 2 1

```
//k-111 10/26/2021
private void removeRoom(String name, ServerThread client) {
    final String roomCheck = name.substring(0,1);
    if (!rooms.containsKey(roomCheck)) {
        return false;
    }
    Room current = rooms.getCurRoom();
    if (current != null) {
        current.removeClient(client);
    }
    Room newRoom = rooms.get(roomCheck);
    newRoom.addClient(client);
    return true;
}

public void removeRoom(Room room) {
    rooms.remove(room.getName());
    System.out.println("Server removed room " + room.getName());
}

public static void main(String[] args) {
    System.out.println("Server starting");
    Room room = new Room("Server room");
    int port = 6666;
    try {
        room.setPort(port);
        room.start();
    } catch (IOException e) {
        // can ignore, will either be index out of bounds or type mismatch
        // will handle in the server class prior to this implemented
    }
    System.out.println("Server Started");
}
```

Server code for handling the create/join process

**Caption(s) (required)** ✓Caption Hint: *Describe/highlight what's being shown (ucid/date must be present)***Task Response Prompt***Briefly explain the code/logic/flow involved*

Response:

Most of the actions are similar to `joinRoom()`, which transfers clients to different rooms; `removeRoom()` erases a room from the server. The `main()` same as the previous program method initializes the server and assigns a port, which waits for connections quickly.

**Sub-Task**

Group: Communication

100%

Task #2: Rooms

Sub Task #6: Show that Client messages are constrained to the Room (clients in different Rooms can't talk to each other)

4 2 1

**Task Screenshots**

Gallery Style: 2 Columns

4 2 1

```
//k-111 10/23/2021
protected synchronized void sendMessage(ServerThread sender, String message) {
    if (!isRunning) { // block action if Room isn't running
        return;
    }

    // note: any desired changes to the message must be done before this section
    long senderId = sender == null ? ServerThread.DEFAULT_CLIENT_ID : sender.getClientId();

    // loop over clients and send out the message; reuse client if message failed
    // to be sent
    // Note: this uses a lambda expression for each item in the values() collection,
    // it's one way we can safely remove items during iteration
    info(String.format("sending message to %s recipients: %s", getRoomName(), clientsInRoom.size()), message);
    clientsInRoom.values().removeIf(client -> {
        if (client.getClientId() == senderId) {
            client.sendMessage(senderId, message);
        }
        if (client.isDisconnected()) {
            info(String.format("removing disconnected client[%s] from list", client.getClientId()));
            disconnect(client);
        }
    });
    return failedToSend;
}

// and send data to client(s)
```

This method sends a message to all clients currently in the room (clientsInRoom)

**Caption(s) (required)** ✓

Caption Hint: *Describe/highlight what's being shown*

## Task Response Prompt

*Briefly explain why/how it works this way*

Response:

This preserves messages to a room by only sending messages to clients in that room thus preventing cross-room communications. Pros include methods like thread-safe, synchronized methods which do not allow problems that exist in many-thread application.

End of Task 2

End of Group: Communication

Task Status: 2/2

Group

Group: Disconnecting/Termination

Tasks: 1

Points: 3

100%

▲ COLLAPSE ▲

Task

Group: Disconnecting/Termination

Task #1: Disconnecting

Weight: ~100%

Points: ~3.00

100%

▲ COLLAPSE ▲

Details:

Important: Code screenshots should be fairly concise (try to show only the sections of code relevant to the question)

Capturing all possible code (i.e., including a lot of irrelevant code) can lead to a reduced grade.

Columns: 1

Sub-Task

Group: Disconnecting/Termination

Task #1: Disconnecting

Sub Task #1: Show Clients gracefully disconnecting (should not crash Server or other Clients)

100%

## Task Screenshots

Gallery Style: 2 Columns

```

/**
 * Tells the client information about a disconnect (similar to leaving a room)
 * @param clientId their unique identifier
 * @param clientName their name
 * @return success of sending the payload
 */
//kr553 10/21/2024
public boolean sendDisconnect(long clientId, String clientName) {
    ConnectionPayload cp = new ConnectionPayload();
    cp.setPayloadType(PayloadType.DISCONNECT);
    cp.setConnect(false);
    cp.setClientId(clientId);
    cp.setClientName(clientName);
    return send(cp);
}

```

```

@Override
protected void disconnect() {
    //sendDisconnect(clientId, clientName);
    super.disconnect();
}
// handle received message from the Client
//kr553 10/20/2024

```

method sends a disconnect payload to other clients in the room, informing them that the client has left.

### Caption(s) (required) ✓

Caption Hint: *Describe/highlight what's being shown*

**Sub-Task**

Group: Disconnecting/Termination

100%

Task #1: Disconnecting

Sub Task #2: Show the code related to Clients disconnecting

## Task Screenshots

Gallery Style: 2 Columns

4

2

1

```

/**
 * Takes a ServerThread and removes them from the Server
 * Adding the synchronized keyword ensures that only one thread can execute
 * these methods at a time,
 * preventing concurrent modification issues and ensuring thread safety
 *
 * @param client
 */
//kr553 10/21/2024
protected synchronized void disconnect(ServerThread client) {
    if (!isRunning) [] // block action if Room isn't running
    []
    long id = client.getClientId();
    sendDisconnect(client);
    client.disconnect();
    // removedClient(client); // -- use this just for normal room leaving
    clientsInRoom.remove(client.getClientId());
    // Improved logging with user data
    info(String.format("%s[%s] disconnected", client.getClientName(), id));
}

```

```

// send/sync data to client(s)

/**
 * Sends to all clients details of a disconnect client
 * @param client
 */
//kr553 10/21/2024
protected synchronized void sendDisconnect(ServerThread client) {
    info(String.format("sending disconnect status to %s recipients", getRoom().clientsInRoom.size()));
    clientsInRoom.values().removeIf(clientInRoom -> {
        boolean failedToSend = !clientInRoom.sendDisconnect(client.getClientId(), client.getClientName());
        if (failedToSend) {
            info(String.format("Removing disconnected client[%s] from list", client.getClientId()));
            disconnect(client);
        }
    });
    return #failedToSend;
}

```

server notifies other clients in the room via sendDisconnect() and safely removes the client from the room.

method sends a disconnection notification to all clients

```

//kr553 10/21/2024
protected synchronized void removedClient(ServerThread client) {
    if (!isRunning) {} // block action if Room isn't running
    []
    // notify remaining clients of someone leaving
    // happens before removal so leaving client gets the data
    sendRoomStatus(client.getClientId(), client.getClientName(), false);
    clientsInRoom.remove(client.getClientId());
    info(String.format("%s[%s] left the room", client.getClientName(), client.getClientId(), getRoom()));
    autoCleanup();
}

```

method removes the client from the room and notifies other clients.

### Caption(s) (required) ✓

Caption Hint: *Describe/highlight what's being shown (ucid/date must be present)*

## Task Response Prompt

Briefly explain the code/logic/flow involved

Response:

The disconnect() method – clears clients and, after that, sendDisconnect(); removedClient() – cleans the room after clients left. Synchronized methods help in making the required facilities safe from the interference of other threads running during some client activities.

**Sub-Task**

Group: Disconnecting/Termination



Task #1: Disconnecting

Sub Task #3: Show the Server terminating (Clients should be disconnected but still running)

## Task Screenshots

Gallery Style: 2 Columns

4 2 1

```
/*
 * Gracefully disconnect clients
 */
//kr553 10/21/2024
private void shutdown() {
    try {
        //choose removeIf over forEach to avoid potential ConcurrentModificationException
        //since empty rooms tell the server to remove themselves
        rooms.values().removeIf(room -> {
            room.disconnectAll();
            return true;
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}

//kr553 10/21/2024
private Server(){
    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        System.out.println("JVM is shutting down. Perform cleanup tasks.");
        shutdown();
    }));
}
```

Disconnects clients, removes rooms, and allows clients to continue. the shutdown process is triggered

**Caption(s) (required) ✓**

Caption Hint: *Describe/highlight what's being shown*

**Sub-Task**

Group: Disconnecting/Termination



Task #1: Disconnecting

Sub Task #4: Show the Server code related to handling termination

## Task Screenshots

Gallery Style: 2 Columns

4 2 1

```
//kr553 10/21/2024
private Server(){
    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        System.out.println("JVM is shutting down. Perform cleanup tasks.");
        shutdown();
    }));
}

/*
 * Gracefully disconnect clients
 */
//kr553 10/21/2024
private void shutdown() {
    try {
        //choose removeIf over forEach to avoid potential ConcurrentModificationException
        //since empty rooms tell the server to remove themselves
        rooms.values().removeIf(room -> {
            room.disconnectAll();
            return true;
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

shutdown() method is called to disconnect clients and clean up rooms.

The shutdown() method disconnects all clients via room.disconnectAll()

**Caption(s) (required) ✓**

Caption Hint: *Describe/highlight what's being shown (ucid/date must be present)*

## Task Response Prompt

Briefly explain the code/logic/flow involved

Response:

The server side's shutdown hook calls function `shutdown()` that removes rooms and disconnects all the clients and informs all the clients about the termination so they can run locally without getting crashed.

End of Task 1

End of Group: Disconnecting/Termination

Task Status: 1/1

Group

Group: Misc

Tasks: 3

Points: 1

100%

▲ COLLAPSE ▲

Task

Group: Misc

Task #1: Add the pull request link for this branch

Weight: ~33%

Points: ~0.33

100%

## Task URLs

URL #1

<https://github.com/KushDev19/kr553-IT114-005/pull/6>

URL

<https://github.com/KushDev19/kr553-IT114-005/>

End of Task 1

Task

Group: Misc

Task #2: Talk about any issues or learnings during this assignment

Weight: ~33%

Points: ~0.33

100%

### Details:

Few related sentences about the Project/sockets topics



# Task Response Prompt

Response:

there was this issue where i changes the packages first and then compiling using javac, class files were still with module 5 part 5 package name and it was not changing i just simply exhausted trying it numerous times and then i quit the application, after like 3 hours i tries to do it again and it worked i dont know if it was some bug due to which this problem was existing but yeah other then that there were no issues.

End of Task 2

## Task



Group: Misc

Task #3: WakaTime Screenshot

Weight: ~33%

Points: ~0.33

[▲ COLLAPSE ▾](#)

### Details:

Grab a snippet showing the approximate time involved that clearly shows your repository.

The duration isn't considered for grading, but there should be some time involved.



## Task Screenshots

Gallery Style: 2 Columns

4                    2                    1



wakatime

End of Task 3

End of Group: Misc

Task Status: 3/3

