Submission Worksheet

Submission Data

Course: IT114-003-F2025
Assignment: IT114 Milestone 2 - RPS
Student: Laura L. (lsl8)
Status: Submitted | **Worksheet Progress:** 100%
Potential Grade: 10.00/10.00 (100.00%)
Received Grade: 0.00/10.00 (0.00%)
Started: 11/24/2025 10:26:49 PM
Updated: 11/25/2025 12:08:19 AM
Grading Link: https://learn.ethereallab.app/assignment/v3/IT114-003-F2025/it114-milestone-2-rps/grading/lsl8
View Link: https://learn.ethereallab.app/assignment/v3/IT114-003-F2025/it114-milestone-2-rps/view/lsl8

Instructions

1. Refer to Milestone2 of Rock Paper Scissors
    1. Complete the features
2. Ensure all code snippets include your ucid, date, and a brief description of what the code does
3. Switch to the `Milestone2` branch
    1. `git checkout Milestone2`
    2. `git pull origin Milestone2`
4. Fill out the below worksheet as you test/demo with 3+ clients in the same session
5. Once finished, click "Submit and Export"
6. 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 Milestone2`
    4. On Github merge the pull request from `Milestone2` to `main`
7. Upload the same PDF to Canvas
8. Sync Local
    1. `git checkout main`
    2. `git pull origin main`

Section #1: (1 pt.) Payloads

Progress: 100%

☰ Task #1 (1 pt.) - Show Payload classes and subclasses

Progress: 100%

Details:

- Reqs from the document
    - Provided Payload for applicable items that only need client id, message, and type
    - PointsPayload for syncing points of players

- Each payload will be presented by debug output (i.e, properly override the toString() method like the lesson examples)

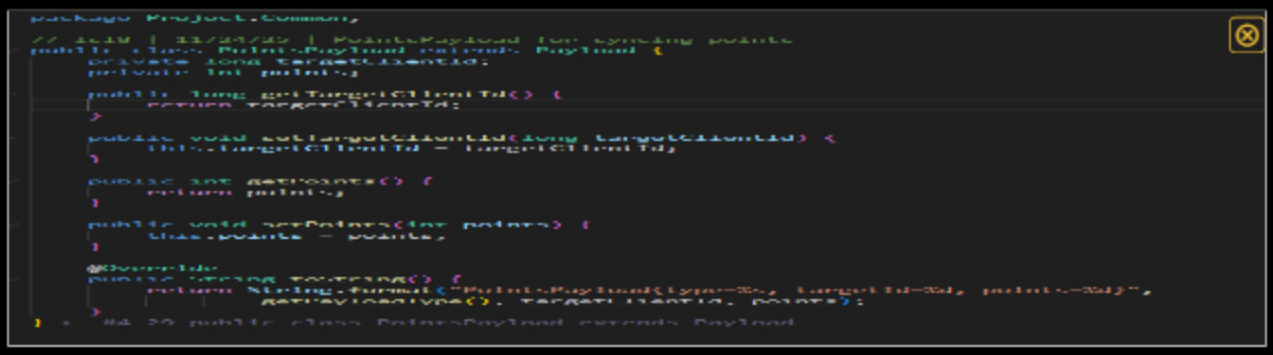
Part 1:

Progress: 100%

Details:

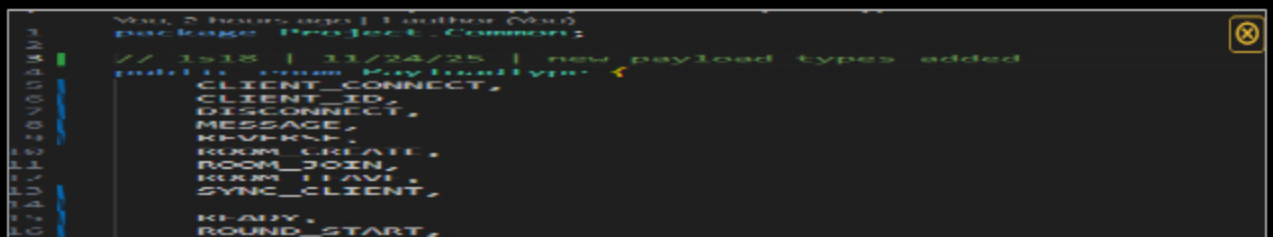
- Show the code related to your payloads (Payload, PointsPayload, and any new ones added)
- Each payload should have an overriden toString() method showing its internal data

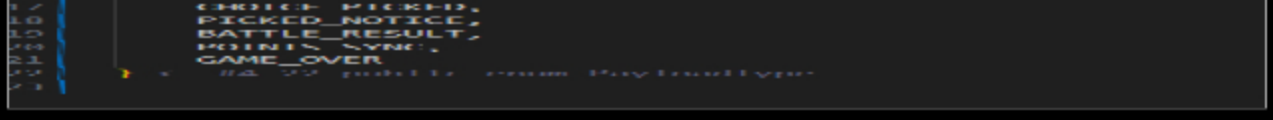
Payload.java



PointsPayload.java

ConnectionPayload.java





PayloadTypes.java

Saved: 11/24/2025 10:33:11 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the purpose of each payload shown in the screenshots and their properties

Your Response:

Each payload in my project has a specific purpose that helps the server and clients communicate clearly during the game. The basic Payload class is used for most actions, like sending messages, readying up, making a pick, or receiving round start and game-over notifications. It contains just three main pieces of information: the type of action, the client ID, and an optional message. The ConnectionPayload is used when information about a player's identity or room status needs to be shared. This payload sends things like the player's name, join/leave notifications, and the list of users in a room so everyone stays in sync. The PointsPayload was added for Milestone 2 and is used only for updating player scores after each battle. It contains the target client's ID and their current point total so every client can show the correct scoreboard. All payloads also override toString() so the server and client logs display helpful debug information showing the internal data, making it easier to understand what messages are flowing through the system.

Saved: 11/24/2025 10:33:11 PM

Section #2: (4 pts.) Lifecycle Events

Progress: 100%

Task #1 (0.80 pts.) - GameRoom Client Add/Remove

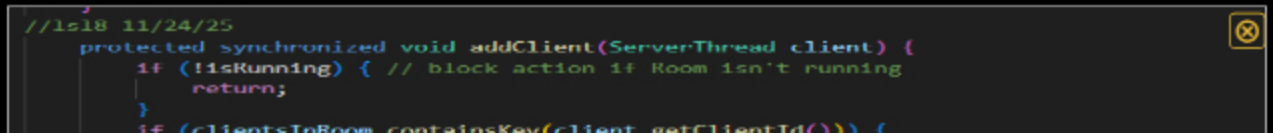
Progress: 100%

Part 1:

Progress: 100%

Details:

- Show the `onClientAdded()` code
- Show the `onClientRemoved()` code



```
            info("Attempting to add a client that already exists in the room");
            return;
        }
        clientsInRoom.put(client.getClientId(), client);
        client.setCurrentRoom(this);
        client.sendResetUserList();
        syncExistingClients(client);
        // notify clients of someone joining
        joinStatusRelay(client, true);
```

addClient

```
//1s18 11/24/25       You, 4 hours ago • Uncommitted changes
    protected synchronized void removeClient(ServerThread client) {
        if (!isRunning) { // block action if Room isn't running
            return;
        }
        if (!clientsInRoom.containsKey(client.getClientId())) {
            info("Attempting to remove a client that doesn't exist in the room");
            return;
        }
        ServerThread removedClient = clientsInRoom.get(client.getClientId());
        if (removedClient != null) {
            // notify clients of someone joining
            joinStatusRelay(removedClient, false);
            clientsInRoom.remove(client.getClientId());
            autoCleanup();
        } <- #60-65 if (removedClient != null)
    } <- #51-66 protected synchronized void removeClient(ServerThread client)
```

removeClient

Saved: 11/24/2025 10:40:40 PM

Part 2:

Progress: 100%

Details:

- Briefly note the actions that happen in `onClientAdded()` (app data should at least be synchronized to the joining user)
- Briefly note the actions that happen in `onClientRemoved()` (at least should handle logic for an empty session)

Your Response:

In my project, the logic for adding and removing clients is handled inside the Room class. When a new client joins a room, the addClient() method acts like an onClientAdded() callback by registering them in the room's client list, updating their current room, syncing them with existing users, and notifying everyone that they joined. When a client leaves, the removeClient() method serves as the onClientRemoved() logic by removing them from the room's client list, broadcasting a leave message, and automatically cleaning up the room if it becomes empty. Together, these functions manage room membership in a clean and consistent way.

Saved: 11/24/2025 10:40:40 PM

Task #2 (0.80 pts.) - GameRoom Session Start

Progress: 100%

Details:

- Reqs from document

- First round is triggered
- Reset/set initial state

Part 1:

Progress: 100%

Details:

- Show the snippet of `onSessionStart()`

```
private void startRound() {
    choices.clear();
    phase = GamePhase.CHOOSING;

    if (roundTimer != null) roundTimer.cancel();
    roundTimer = new Timer(true);
    roundTimer.schedule(new TimerTask() {
        @Override
        public void run() {
            synchronized (GameRoom.this) {
                endRound(); // Condition 1: timer expires
            }
        } <- #91-95 public void run()
    }, 15000); // 15 second pick window <- #89-96 roundTimer.schedule

    clientsInRoom.values().forEach(ServerThread::sendRoundStart);
} <- #83-99 private void startRound()
```

startRound

Saved: 11/24/2025 10:46:44 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the logic that occurs here (i.e., setting up initial session state for your project) and next lifecycle trigger

Your Response:

The onSessionStart() logic in my project is handled inside the startRound() method of the GameRoom class. This is where the game resets everything needed for a new session: it clears all previous choices, sets the game phase to CHOOSING, and starts a round timer. After the state is initialized, the server sends a ROUND_START payload to all connected clients telling them the round has begun and they can now make their pick. This method acts exactly like an onSessionStart() function because it prepares all players and the game state for the first round of a new session.

Saved: 11/24/2025 10:46:44 PM

Task #3 (0.80 pts.) - GameRoom Round Start

Progress: 100%

Details:

- Reqs from Document
  - Initialize remaining Players' choices to null (not set)
  - Set Phase to "choosing"
  - GameRoom round timer begins

Part 1:

Progress: 100%

Details:

- Show the snippet of `onRoundStart()`

```
private void startRound() {
    choices.clear();
    phase   GamePhase.CHOOSING;

    if (roundTimer != null) roundTimer.cancel();
    roundTimer = new Timer(true);
    roundTimer.schedule(new TimerTask() {
        @Override
        public void run() {
            synchronized (GameRoom.this) {
                endRound(); // Condition 1: timer expires
            }
        } <- #91-95 public void run()
    }, 15000); // 15 second pick window <- #89-96 roundTimer.schedule

    clientsInRoom.values().forEach(ServerThread::sendRoundStart);
} <- #83-99 private void startRound()
```

startRound

Saved: 11/24/2025 11:06:21 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the logic that occurs here (i.e., setting up the round for your project)

Your Response:

The onRoundStart() logic happens inside the startRound() method. This is where the game resets every active player's choice (sets them back to null), switches the game phase to CHOOSING, and starts the 15-second round timer. After the setup is complete, the server sends a ROUND_START message to every connected client to let them know they can now make their pick. My implementation combines the logic for both session start and round start into the same method, startRound(). This works because the requirements for beginning a session (after all players press /ready) and starting a new round are identical: reset player choices, set the phase to "choosing", start the round timer, and notify all clients. Therefore, onSessionStart simply calls startRound(), and so does any subsequent round.

Saved: 11/24/2025 11:06:21 PM

Task #4 (0.80 pts.) - GameRoom Round End

Progress: 100%

Details:

- Reqs from Document
  - **Condition 1:** Round ends when round timer expires
  - **Condition 2:** Round ends when all active Players have made a choice
  - All Players who are not eliminated and haven't made a choice will be marked as eliminated
  - Process Battles:
    - Round-robin battles of eligible Players (i.e., Player 1 vs Player 2 vs Player 3 vs Player 1)
      - Determine if a Player loses if they lose the "attack" or if they lose the "defend" (since each Player has two battles each round)
        - Give a point to the winning Player
        - Points will be stored on the Player/User object
        - Sync the points value of the Player to all Clients
      - Relay a message stating the Players that competed, their choices, and the result of the battle
      - Losers get marked as eliminated (Eliminated Players stay as spectators but are skipped for choices and for win checks)
    - Count the number of non-eliminated Players
      - If one, this is your winner (onSessionEnd())
      - If zero, it was a tie (onSessionEnd())
      - If more than one, do another round (onRoundStart())

Part 1:

Progress: 100%

Details:

- Show the snippet of `onRoundEnd()`

```
private void endRound() {
    if (gameState != GameState.RESOLVING) return;
    phase = GamePhase.RESOLVING;

    if (roundTimer != null) {
        roundTimer.cancel();
        roundTimer = null;
    }

    for (ServerThread st : clientsInRoom.values()) {
        long id = st.getClientId();
        if (!eliminated.contains(id) && !choices.containsKey(id)) {
            eliminated.add(id);
        }
    }

    processBattles();
    syncAllPoints();

    int active = countActivePlayers();
    if (active == 1) {
        long winner = remainingPlayerId();
        gameOver("Winner: " + name(winner));
        resetSession();
    } else if (active == 0) {
        gameOver("No players remain. It's a tie.");
        resetSession();
    } else {
        phase = GamePhase.NULL;
        startRound();  // next round automatically
    }
}
```

endRound()

Saved: 11/24/2025 11:11:25 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the logic that occurs here (i.e., cleanup, end checks, and next lifecycle events)

Your Response:

When the round ends, the game performs a cleanup and moves into the next part of the session lifecycle. First, the phase switches from choosing to resolving and the round timer is stopped so no more picks can be made. Then the server checks which players never submitted a pick and automatically eliminates them from the rest of the session. After that, all valid players participate in the round-robin battle sequence, and the server updates everyone's points so each client knows the current scoreboard. Once battles are finished, the server checks how many players are still active: if only one player is left, that person is declared the winner; if nobody is left, the session ends in a tie; and if multiple players are still alive, the game automatically starts a new round. This ensures each round fully cleans up, checks win/tie conditions, and sets up the correct next step in the game flow.

Saved: 11/24/2025 11:11:25 PM

Task #5 (0.80 pts.) - GameRoom Session End

Progress: 100%

Details:

- Reqs from Document
  - **Condition 1:** Session ends when one Player remains (they win)
  - **Condition 2:** Session ends when no Players remain (this is a tie)
  - Send the final scoreboard to all clients sorted by highest points to lowest (include a game over message)
  - Reset the player data for each client server-side and client-side (do not disconnect them or move them to the lobby)
  - A new ready check will be required to start a new session

Part 1:

Progress: 100%

Details:

- Show the snippet of `onSessionEnd()`

```
private void gameOver(String msg) {
    StringBuilder sb = new StringBuilder();
    sb.append(msg).append("\n Final scores:\n");

    points.entrySet().stream()
            .sorted((a, b) -> Integer.compare(b.getValue(), a.getValue()))
            .forEach(e -> sb.append(String.format("%s: %d\n", nameOf(e.getKey()), e.getValue())));
```

```
        String text = sb.toString();
        clientsInRoom.values().forEach(st -> st.sendGameOver(text));
    } <- #189-199 private void gameOver(String msg)
    private void resetSession() {
        choices.clear();
        eliminated.clear();
        points.clear();
        readySet.clear();
        phase = GamePhase.ENDED;
        syncAllPoints();
    } <- #200-207 private void resetSession()
```

resetSession

```
    if (roundTimer != null) roundTimer.cancel();
    roundTimer = new Timer(true);
    roundTimer.schedule(new TimerTask() {
        @Override
        public void run() {
            synchronized (GameRoom.this) {
                endRound(); // Condition 1: timer expires
            }
        } <- #91-95 public void run()
    }, 15000); // 15 second pick window <- #89-96 roundTimer.schedule
```

Condition 1

```
    int active = countActivePlayers();
    if (active == 1) {
        long winner = remainingPlayerId();
        gameOver("Winner: " + nameOf(winner));
        resetSession();
    } else if (active == 0) {
        gameOver("No players remain. It's a tie.");
        resetSession();
    } else {
        phase = GamePhase.ENDED;
```

Condition 2

Saved: 11/24/2025 11:24:14 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the logic that occurs here (i.e., cleanup/reset, next lifecycle events)

Your Response:

When the session ends, the GameRoom runs cleanup steps to finish the current game and prepare for a new one. First, the gameOver() method creates a final scoreboard by sorting all players based on their points and sending a "game over" message with the results to every client. After announcing the winner or declaring a tie, the resetSession() method clears all game-related data, including each player's choices, elimination status, and point totals, so everyone starts fresh. The game phase is set back to ENDED, and the server sends a points reset to all clients so their displays go back to zero. Finally, the ready list is cleared, which means players must type ready again before a new session can begin. This ensures the system fully resets the game state without disconnecting or moving anyone, and the lifecycle cleanly returns to the waiting-for-ready phase.

Saved: 11/24/2025 11:24:14 PM

Section #3: (4 pts.) Gameroom User Action And State

Progress: 100%

Task #1 (2 pts.) - Choice Logic

Progress: 100%

Details:

- Reqs from document
  - Command: `/pick <[r,p,s]>` (user picks one)
    - GameRoom will check if it's a valid option
    - GameRoom will record the choice for the respective Player
    - A message will be relayed saying that "X picked their choice"
    - If all Players have a choice the round ends

Part 1:

Progress: 100%

Details:

- Show the code snippets of the following, and clearly caption each screenshot
- Show the Client processing of this command (process client command)
- Show the ServerThread processing of this command (process method)
- Show the GameRoom handling of this command (handle method)
- Show the sending/syncing of the results of this command to users (send/sync method)
- Show the ServerThread receiving this data (send method)
- Show the Client receiving this data (process method)

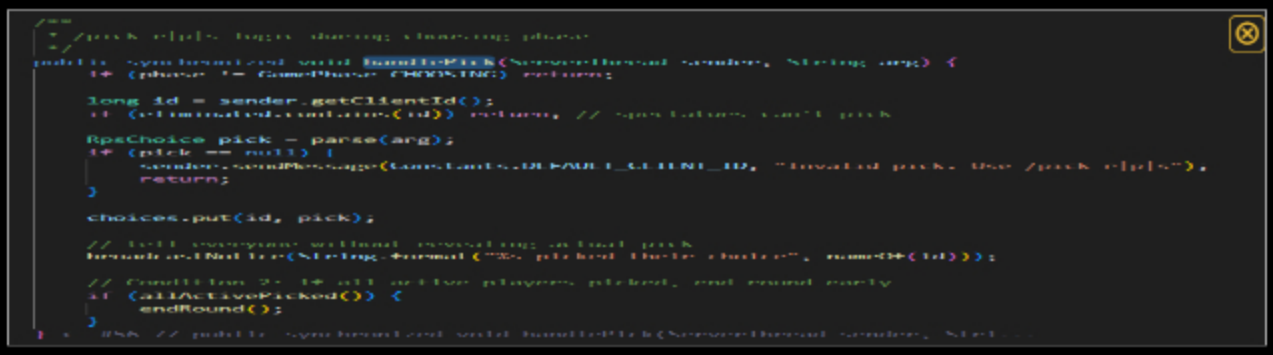
```
} else if (text.startsWith("pick ")) {        You, 3 hours ago • U
    String arg = text.substring(5).trim().toLowerCase();
    Payload p = new Payload();
    p.setPayloadType(PayloadType.CHOICE_PICKED);
    p.setMessage(arg);
    sendToServer(p);
    wasCommand = true;
```

Client processing /pick

```
case CHOICE_PICKED:
    if (currentRoom instanceof GameRoom) {
```

```
    ((GameRoom) currentRoom).handlePick(this, incoming.getMessage());
    } else {
        sendMessage(Constants.DEFAULT_CLIENT_ID, "You must be in a game room to /pick");
    }
    break;
```

ServerThread processing of /pick



GameRoom handling of /pick

```
private void broadcastNotice(String msg) {
    clientsInRoom.values().forEach(st -> st.sendPickedNotice(msg));
}
```

Sending/Synching results to users

```
// 1s18 | 11/24/25 |notice helper
protected boolean sendPickedNotice(String msg) {
    Payload p = new Payload();
    p.setPayloadType(PayloadType.PICKED_NOTICE);
    p.setClientId(Constants.DEFAULT_CLIENT_ID);
    p.setMessage(msg);
    return sendToClient(p);
} <- #153-159 protected boolean sendPickedNotice(String msg)
```

ServerThread sending the data

```
case CHOICE_PICKED:
// Server says someone picked (without revealing their choice)
    System.out.println(
    TextFX.colorize(payload.getMessage(), Color.YELLOW));
    break;
```

Client receiving the data

Saved: 11/24/2025 11:47:13 PM

Part 2:

Progress: 100%

Details:

- Briefly explain/list in order the whole flow of this command being handled from the client-side to the server-side and back

Your Response:

When a player types the /pick command on the client, the client first recognizes it as a command and sends a special payload to the server that contains the player's choice (r, p, or s). The server receives this payload inside ServerThread, and then forwards it to the correct game room. Inside the game room, the server checks if the pick is valid and if the round is currently in the choosing phase. If it is valid, the pick gets saved for that player, and the server immediately sends a message to all connected clients saying that the player "picked their choice" without revealing what they picked. Every client receives this update and prints it on their screen. The game room then checks if all active players have made a pick. If they have, the round ends and the server continues with battle processing, awarding points, syncing the updated scores, and possibly sending a game-over message. Finally, all clients receive the results and update their displays accordingly. This creates a complete loop where the command starts on the client, gets processed fully on the server, and returns visual feedback back to all clients.

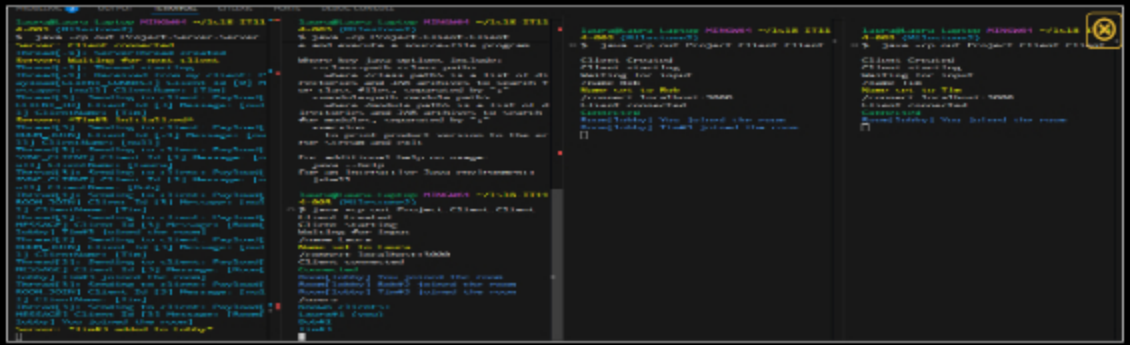
Saved: 11/24/2025 11:47:13 PM

Task #2 (2 pts.) - Game Cycle Demo

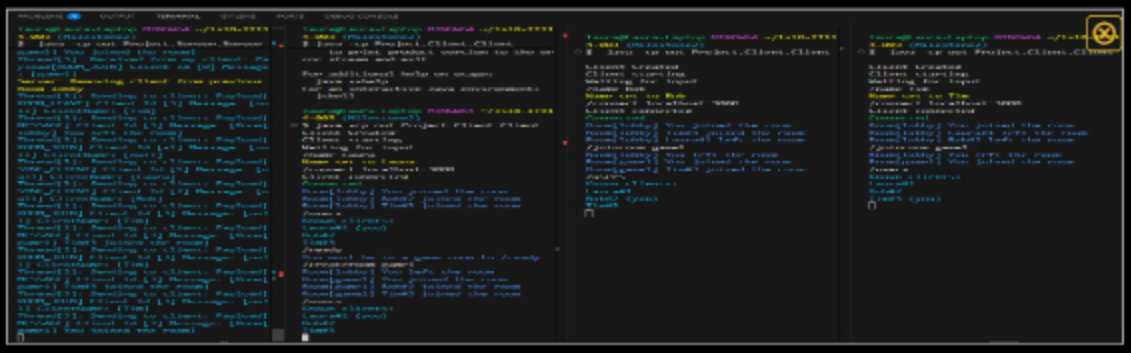
Progress: 100%

Details:

- Show examples from the terminal of a full session demonstrating each command and progress output
- This includes battle outcomes, scores and scoreboards, etc
- Ensure at least 3 Clients and the Server are shown
- Clearly caption screenshots



3 Clients and Server



Game room "game1"

All 3 clients have to be ready to start the game

Game in process

Saved: 11/24/2025 11:59:12 PM

Section #4: (1 pt.) Misc

Progress: 100%

Task #1 (0.33 pts.) - Github Details

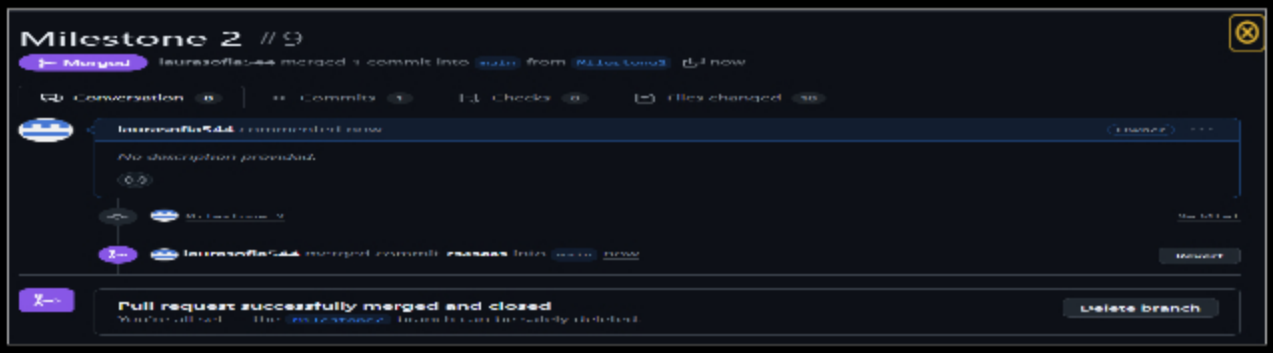
Progress: 100%

Part 1:

Progress: 100%

Details:

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



Commit history

Saved: 11/25/2025 12:08:19 AM

Part 2:

Progress: 100%

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

URL #1
https://github.com/laurasofia544/lsl8-IT114-003/

URL
https://github.com/laurasofia544/

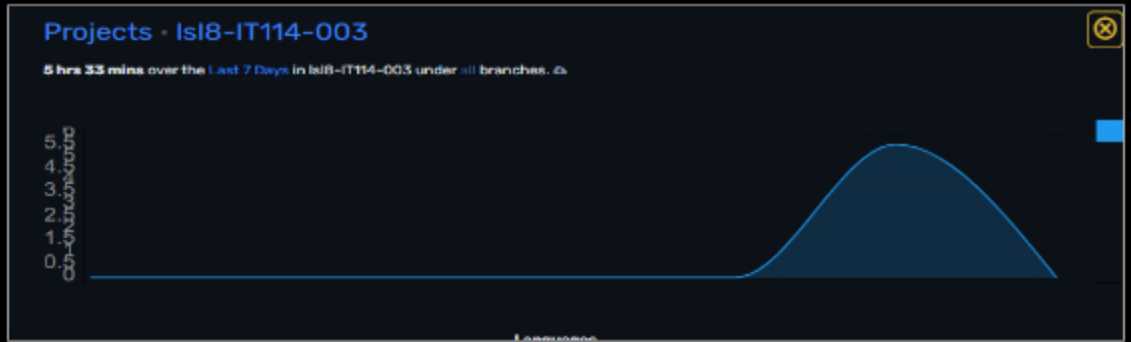
Saved: 11/25/2025 12:08:19 AM

Task #2 (0.33 pts.) - WakaTime - Activity

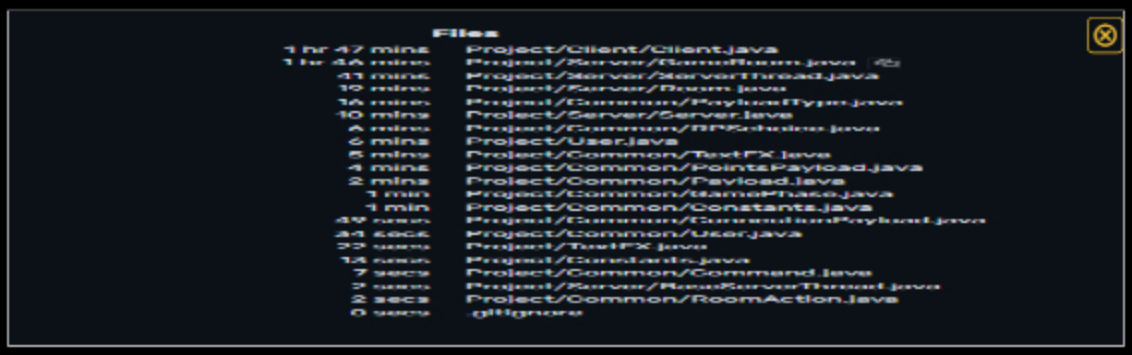
Progress: 100%

Details:

- Visit the WakaTime.com Dashboard
- Click `Projects` and find your repository
- Capture the overall time at the top that includes the repository name
- Capture the individual time at the bottom that includes the file time
- Note: The duration isn't relevant for the grade and the visual graphs aren't necessary



overall



Individual

Saved: 11/25/2025 12:03:30 AM

Task #3 (0.33 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 full client-server game system works from start to finish. I learned how the client sends commands, how the server receives and processes them, and how GameRoom controls all the game logic like ready checks, round timers, choices, battles, scoring, and resetting the session. I also learned how different payload types are used to communicate between the client and server, and how important it is for both sides to handle messages correctly so the game runs smoothly.

Saved: 11/25/2025 12:04:47 AM

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 for me was setting up the basic commands and getting the client and server to communicate using simple payloads like /name, /connect, and basic chat messages. These parts were straightforward because they followed the same structure as the lesson examples, so I already understood how to send a payload, how the server processes it, and how the client displays the result. Once those pieces were in place, extending the logic for things like joining and creating rooms also felt easier because it reused the same patterns

Saved: 11/25/2025 12:05:48 AM

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 definitely implementing the full game logic and lifecycle for Rock–Paper–Scissors, especially making sure it worked correctly with multiple clients at the same time. Tracking which players were active, which were eliminated, when everyone had picked, and when a round or a whole session should end required a lot of coordination between the client, the ServerThread, and the GameRoom. The timing logic, ready checks, round resets, elimination rules, and point-syncing all had to work together without breaking the rest of the server. Another difficult part was making sure every payload type was handled on both sides so nothing triggered an "Unhandled payload type"

Saved: 11/25/2025 12:06:52 AM