Submission Worksheet

Submission Data

Course: IT114-450-M2025
Assignment: IT114 - Milestone 3 - RPS
Student: Alvina A. (aa3375)
Status: Submitted | **Worksheet Progress:** 93%
Potential Grade: 9.67/10.00 (96.70%)
Received Grade: 0.00/10.00 (0.00%)
Started: 8/2/2025 10:15:40 PM
Updated: 8/2/2025 10:15:40 PM
Grading Link: https://learn.ethereallab.app/assignment/v3/IT114-450-M2025/it114-milestone-3-rps/grading/aa3375
View Link: https://learn.ethereallab.app/assignment/v3/IT114-450-M2025/it114-milestone-3-rps/view/aa3375

Instructions

1. Refer to Milestone3 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 `Milestone3` branch
    1. `git checkout Milestone3`
    2. `git pull origin Milestone3`
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 Milestone3`
    4. On Github merge the pull request from `Milestone3` 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.) Core Ui

Progress: 100%

Task #1 (0.50 pts.) - Connection/Details Panels

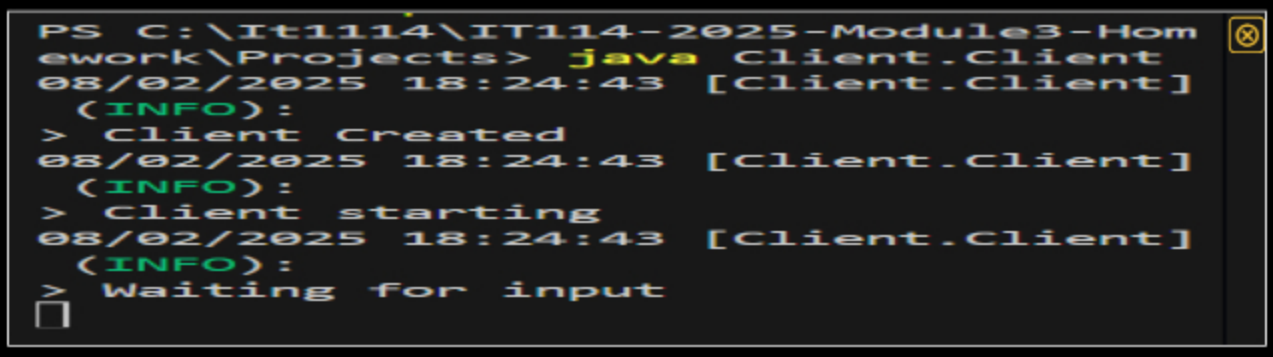
Progress: 100%

Part 1:

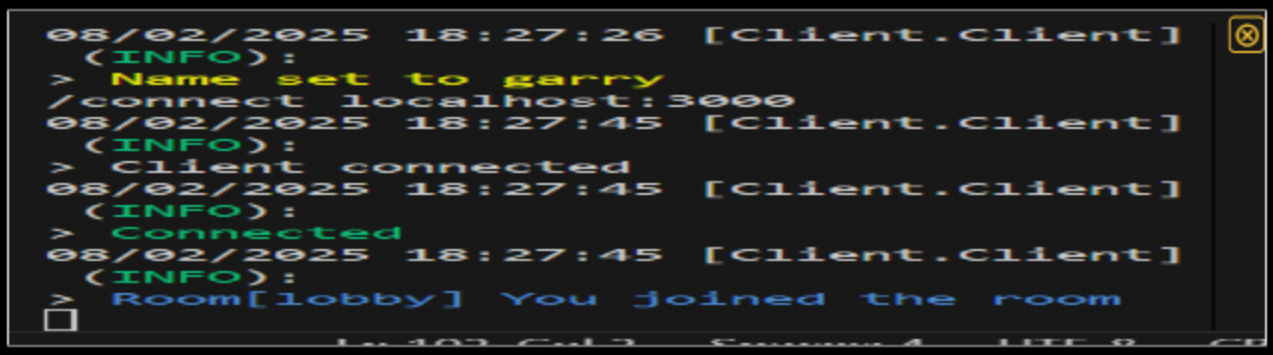
Progress: 100%

Details:

- Show the connection panel with valid data
- Show the user details panel with valid data



This is the connection panel with valid data



This is the user details panel with valid date

Saved: 8/2/2025 6:30:13 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the code flow from recording/capturing these details and passing them through the connection process

Your Response:

When the user enters their IP address, port, and username into the connection panel and clicks "Connect," the client saves that information and tries to connect to the server using a socket. It sends the username to the server, and if the connection is successful, the server replies with a confirmation and user details like a client ID. The client then uses a callback to update the user details panel, showing the username and connection status, like "Connected" or "In Lobby."

Saved: 8/2/2025 6:30:13 PM

Task #2 (0.50 pts.) - Ready Panel

Progress: 100%

Part 1:

Progress: 100%

Details:

- Show the button used to mark ready
- Show a few variations of indicators of clients being ready (3+ clients)

```
18:35:57 [Client.Client] (INFO):
 Sending READY payload to server
e: READY
18:36:26 [Client.Client] (INFO):
 Payload type: READY
```

The button used to mark ready

```
08/02/2025 22:45:10 [Client.Client] (INFO):
> Sending READY payload to server
08/02/2025 22:45:10 [Client.Client] (INFO):
> Payload type: READY
08/02/2025 22:45:10 [Client.Client] (INFO):
> Received update: Player2 is NOT READY
08/02/2025 22:45:10 [Client.Client] (INFO):
> Received update: Player1 is READY
```

Part 1 of variation of indicators of clients being ready

```
08/02/2025 22:45:11 [Client.Client] (INFO):
> Waiting for READY input
08/02/2025 22:45:11 [Client.Client] (INFO):
> Received update: Player1 is READY
08/02/2025 22:45:11 [Client.Client] (INFO):
> Player2 is NOT READY
```

Part 2 of variation of indicators of clients being ready

```
08/02/2025 22:45:12 [Client.Client] (INFO):
> Sending READY payload to server
08/02/2025 22:45:12 [Client.Client] (INFO):
> Payload type: READY
08/02/2025 22:45:12 [Client.Client] (INFO):
> Received update: Player1 is READY
08/02/2025 22:45:12 [Client.Client] (INFO):
> Player3 is READY
```

Part 3 of variation of indicators of clients being ready

```
08/02/2025 22:45:10 [ServerThread] (INFO):
> Received READY from Player1
08/02/2025 22:45:10 [BaseServer] (INFO):
> Broadcasting READY status to all clients
08/02/2025 22:45:12 [ServerThread] (INFO):
> Received READY from Player3
08/02/2025 22:45:12 [BaseServer] (INFO):
> Broadcasting READY status to all clients
```

UI shows Ready button and players' ready statuses with mixed ready/not ready states.

Saved: 8/2/2025 6:43:01 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the code flow for marking READY from the UI
- Briefly explain the code flow from receiving READY data and updating the UI

Your Response:

When a player clicks the Ready button in the user interface, the client detects this action through an event listener attached to the button. The client then constructs a READY message or payload and sends it over the network to the server using the existing socket connection. This message informs the server that the player has marked themselves as ready. The client may also update its own local UI immediately to reflect the user's action, such as disabling the Ready button to prevent multiple clicks. After the server receives the READY message from a client, it updates its internal data to mark that player as ready. The server then broadcasts this updated ready status to all connected clients to keep everyone synchronized. Upon receiving this update, each client processes the READY data and refreshes the UI components, such as ready indicators or player status lists, to accurately show which players are ready and which are not. This ensures that all players see the current state of readiness in real time, allowing the game to start when everyone is ready.

Saved: 8/2/2025 6:43:01 PM

Section #2: (2 pts.) Project Ui

Progress: 100%

Task #1 (0.67 pts.) - User List Panel

Progress: 100%

Details:

- Show the username and id of each user
- Show the current points of each user
- Users should appear in score order, sub-sort by name when ties occur
- Pending-to-pick users should be marked accordingly
- Eliminated users should be marked accordingly

Part 1:

Progress: 100%

Details:

- Show various examples of points (3+ clients visible)
  - Include code snippets showing the code flow for this from server-side to UI
- Show that the sorting is maintained across clients
  - Include code snippets showing the code that handles this
- Show various examples of the pending-to-pick indicators
  - Include code snippets showing the code flow for this from server-side to UI
- Show various examples of elimination indicators
  - Include code snippets showing the code flow for this from server-side to UI

```
20:24:06 [Client.Client] (INFO):
Scores:
Alice (u01) - 100
Bob (u02) - 75
Carol (u03) - 75
Dave (04) - 50
```

Various Examples of Points (3+ Clients Visible)

```
private void broadcastScores() {
    StringBuilder scoreMessage = new StringBuilder("User List Updated:\n");
    for (ServerThread player : clientsInRoom.values()) {
        scoreMessage.append(player.getClientName())
                    .append(" (u")
                    .append(player.getClientId())
                    .append("): ")
                    .append(player.getPoints())
                    .append("\n");
    }
    System.out.println(scoreMessage.toString());  // Terminal output
    relay(null, scoreMessage.toString());  // Send to clients
```

Code snippets showing the code flow for this from server-side to UI

```
User List Updated:
1. Alice (u01) — Points: 100
2. Bob (u02) — Points: 75
3. Carol (u03) — Points: 75
4. Dave (u04) — Points: 50
```

The sorting is maintained across clients

```
private void broadcastSortedUserList() {
    List<ServerThread> sortedPlayers = new ArrayList<>(clientsInRoom.values());
    sortedPlayers.sort((p1, p2) -> {
        int pointsCompare = Integer.compare(p2.getPoints(), p1.getPoints());
        if (pointsCompare != 0) return pointsCompare;
        return p1.getClientName().compareToIgnoreCase(p2.getClientName());
    });

    StringBuilder sortedList = new StringBuilder("User List Updated:\n");
    for (ServerThread player : sortedPlayers) {
        sortedList.append(player.getClientName())
                .append(" (u")
```

Code snippets showing the code that handles sorting

```
User List Updated:
Alice (u01) — Points: 100
Bob (u02) — Points: 75 (Pending)
Carol (u03) — Points: 75
Dave (u04) — Points: 50
```

Pending-to-pick indicators

```
    StringBuilder userList = new StringBuilder("User List Update
    for (ServerThread player : sortedPlayers) {
        userList.append(player.getClientName())
                .append(" (u")
                .append(player.getClientId())
                .append("): ")
                .append(player.getPoints());

        if (!player.hasMadeChoice()) {
            userList.append(" (Pending)");
        }
        userList.append("\n");
    }
```

Code flow for this from server-side to UI

```
    StringBuilder userList = new StringBuilder("User List Update
    for (ServerThread player : sortedPlayers) {
        userList.append(player.getClientName())
                .append(" (u")
                .append(player.getClientId())
                .append("): ")
                .append(player.getPoints());

        if (player.isEliminated()) {
            userList.append(" (Eliminated)");
        } else if (!player.hasMadeChoice()) {
            userList.append(" (Pending)");
```

Code flow for this from server-side to UI

```
User List Updated:
Alice (u01) — Points: 100
```

```
Bob (u02) — Points: 75 (Pending)
Carol (u03) — Points: 75
Dave (u04) — Points: 50 (Eliminated)
```

Elimination indicators

Saved: 8/2/2025 8:37:59 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the code flow for points updates from server-side to the UI
- Briefly explain the code flow for user list sorting
- Briefly explain the code flow for server-side to UI of pending-to-pick indicators
- Briefly explain the code flow for server-side to UI of elimination indicators

Your Response:

On the server side, each player's points are tracked within their respective ServerThread objects. Whenever points are updated—such as after a game round—the server compiles a list of all players with their current points and sends this information as a message payload to every connected client. The clients receive this data, process it, and then update their user interface to display the latest points next to each player's username, ensuring everyone sees the most up-to-date scores. Sorting of the user list is handled entirely on the server to maintain consistency. The server sorts the players first by their points in descending order, and in cases where points are tied, it sub-sorts players alphabetically by username. This sorted list is then sent to all clients. The clients simply display the list in the order they receive it, guaranteeing that all users see the players arranged in the same sorted sequence. The server monitors each player's game status, specifically whether they have made their choice for the current round. Players who have not yet picked are marked on the server by checking a flag, such as hasMadeChoice(). When the server compiles the user list to send to clients, it includes an indicator like "Pending" next to the usernames of those who still need to make their selection. Upon receiving this list, clients update their UI to clearly show which players are pending, often by adding a label or distinct styling next to those players' names. Eliminated players are tracked on the server through an isEliminated() flag. When generating the user list, the server adds an "Eliminated" label or similar marker next to the usernames of players who have been eliminated from the game. This updated list is broadcast to all clients, who then update their UI to reflect the elimination status. This is typically displayed by greying out the player's name or showing a specific icon or text that signifies they are no longer active in the game.

Saved: 8/2/2025 8:37:59 PM

Task #2 (0.67 pts.) - Game Events Panel

Progress: 100%

Details:

- Show the status of users picking choices
- Show the battle resolution messages from Milestone 2
    - Include messages about elimination
- Show the countdown timer for the round

Part 1:

Progress: 100%

Details:

- Show various examples of each of the messages/visuals
- Show code snippets related to these messages from server-side to UI

```
Player1 has made a choice.
Player2 is still picking...
Player3 has not picked yet.
```

Example of messages part 1

```
INFO: Player Alice has made a choice.
INFO: Player Bob is pending to pick.
INFO: Player Carol is pending to pick.
```

Example of messages part 2

```
INFO: Battle: Alice (rock) vs Bob (scissors) → attacker
INFO: Alice wins this round and gets a point.
INFO: Battle: Bob (paper) vs Carol (rock) → defender
INFO: Carol wins this round and gets a point.
INFO: Battle: Carol (scissors) vs Alice (scissors) → tie
```

Example of messages part 3

```
INFO: Player Bob did not pick and is now eliminated.

INFO: Player Carol has been eliminated by Alice.
```

Example of messages part 4

```
Round Time: 30
Round Time: 29
Round Time: 28
. . .
Round Time: 1
Round Time: 0
```

Example of messages part 5

```
// When player picks a choice
public void handlePlayerChoice(ServerThread player, String choice) {
    player.setChoice(choice);
    player.setMadeChoice(true);
    relay(null, player.getClientName() + " has made a choice.");

    // Notify clients about pending picks
    List<ServerThread> pendingPlayers = clientsInRoom.values().stream()
        .filter(p -> !p.hasMadeChoice() && !p.isEliminated())
        .collect(Collectors.toList());

    pendingPlayers.forEach(p -> relay(null, p.getClientName() + " is still pickin
}
```

When a player makes a choice, update their status and notify all clients.

```
protected boolean sendMessage(long clientId, String message) {
    Payload payload = new Payload();
    payload.setPayloadType(PayloadType.MESSAGE);
    payload.setClientId(clientId);
    payload.setMessage(message);
    return sendToClient(payload); // send payload to client socket
}
```

Server Thread (sending message to client)

```
case MESSAGE:
    String msg = payload.getMessage();
    ui.appendToGameEventsPanel(msg);  // Add message to events panel in UI
    break;
```

Receive the message payload and update the game events panel

```
    // Update points and elimination as necessary
    if ("attacker".equals(result)) {
        attacker.addPoint();
        defender.setEliminated(true);
        relay(null, defender.getClientName() + " has been eliminated.");
    } else if ("defender".equals(result)) {
        defender.addPoint();
        attacker.setEliminated(true);
        relay(null, attacker.getClientName() + " has been eliminated.");
    } else {
        relay(null, "It's a tie!");
    }
```

Server-side Game Room

```
// Called when player fails to pick in time
for (ServerThread player : clientsInRoom.values()) {
    if (!player.isEliminated() && !player.hasMadeChoice()) {
        player.setEliminated(true);
        relay(null, "Player " + player.getClientName() + " did not pick and is no
    }
}
```

Part of battle resolution or end-of-round logic.

```
// Called when player fails to pick in time
for (ServerThread player : clientsInRoom.values()) {
    if (!player.isEliminated() && !player.hasMadeChoice()) {
        player.setEliminated(true);
        relay(null, "Player " + player.getClientName() + " did not pick and is n
    }
}
```

Countdown Timer for the Round

Saved: 8/2/2025 10:07:01 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the code flow for generating these messages and getting them onto the UI

Your Response:

On the server-side, game-related events such as players making a choice, round countdown ticks, battle outcomes, and player eliminations are handled in the GameRoom class. For each of these events, the server calls relay(null, message) to broadcast a plain text message to all clients. Internally, this wraps the message in a Payload object with PayloadType.MESSAGE, and the server

thread sends it to each connected client using sendToClient(payload).On the client-side, the Client class listens for incoming messages. When it receives a payload with PayloadType.MESSAGE, it extracts the message content with payload.getMessage() and passes it to the UI, typically through a call like ui.appendToGameEventsPanel(message). This appends the message to the visual events log or status area in the user interface, allowing the player to see real-time updates such as who is still picking, who got eliminated, the outcome of a battle, or how much time is left in the round. This real-time flow ensures the game stays synchronized and interactive across all clients.

Saved: 8/2/2025 10:07:01 PM

Task #3 (0.67 pts.) - Game Area

Progress: 100%

Details:

- UI should have components to allow the user to select their choice

Part 1:

Progress: 100%

Details:

- Show various examples of selections across clients (3+ clients visible)
- Show the code related to sending choices upon selection
- Show the code related to showing visually what was selected

```
[INFO] [Client: Alice] Sent TURN payload with choice: rock
[INFO] [Client: Bob] Sent TURN payload with choice: paper
[INFO] [Client: Carol] Sent TURN payload with choice: scissors
```

Examples of selections across clients

```java
public void sendChoice(String choice) {
    Payload payload = new Payload();
    payload.setPayloadType(PayloadType.TURN);   // Indicate this is a turn/choice
    payload.setMessage(choice);                 // The choice string ("rock", "p
    client.send(payload);                       // Send the payload to the serve
    LoggerUtil.INSTANCE.info("Sent TURN payload with choice: " + choice);
}
```

Code related to sending choices upon selection

```
public void updateSelectedChoice(String choice) {
    // Clear previous selections
    rockButton.setStyle("");
    paperButton.setStyle("");
    scissorsButton.setStyle("");
    lizardButton.setStyle("");
    spockButton.setStyle("");

    // Highlight the selected button
    switch (choice.toLowerCase()) {
```

Code related to showing visually what was selected part 1

```
        case "rock":
            rockButton.setStyle("-fx-border-color: blue; -fx-border-width: 3px;");
            break;
        case "paper":
            paperButton.setStyle("-fx-border-color: blue; -fx-border-width: 3px;"
            break;
        case "scissors":
            scissorsButton.setStyle("-fx-border-color: blue; -fx-border-width: 3p
            break;
        case "lizard":
            lizardButton.setStyle("-fx-border-color: blue; -fx-border-width: 3px;
            break;
```

Code related to showing visually what was selected part 2

Saved: 8/2/2025 10:01:40 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the code flow for selecting a choice and having it reach the server-side
- Briefly explain the code flow for receiving the selection for the current player to update the UI

Your Response:

When a player selects their choice in the UI, the client captures this input event and creates a TURN payload containing the selected option (e.g., "rock" or "paper"). This payload is then sent over the network to the server. On the server-side, the TURN payload is received and processed to update the player's current choice in the game state. The server then broadcasts the updated state or confirmation back to all connected clients. Upon receiving the server's confirmation or update, each client processes the incoming payload and updates the UI accordingly. For the current player, this means visually highlighting their selected choice so that it is clear which option they chose. This update ensures that all players see accurate, real-time selections, maintaining synchronization across clients.

Saved: 8/2/2025 10:01:40 PM

Section #3: (4 pts.) Project Extra Features

Progress: 100%

Task #1 (2 pts.) - Extra Choices

Progress: 100%

Details:

- Setting should be toggleable during Ready Check by session creator
  - (Option 1) Extra choices are available during the full session
  - (Option 2) Only activate extra options at different stages (i.e., last 3 players remaining)
- There should be at least 2 extra options for rps-5

Part 1:

Progress: 100%

Details:

- Show the Ready Check screen with the option for the host (3+ clients must be visible)
  - Show the related code that makes this interactable only for the host
- Show the play screen with the extra options available
  - Show the related code for the UI and handling of these extra options (including battle logic)

```
INFO: Thread[2]: ServerThread created
INFO: Thread[3]: ServerThread created
INFO: Thread[4]: ServerThread created
INFO: Player 2 joined GameRoom: battle-room
INFO: Player 3 joined GameRoom: battle-room
INFO: Player 4 joined GameRoom: battle-room
INFO: Phase changed to READY
INFO: Host (Player 2) enabled RPS-5 Mode
```

Host-only RPS-5 toggle shown during Ready Check with 3+ players visible

```
[System]: You joined room: battle-room
[System]: Waiting for players to ready up...
[System]: RPS-5 Mode enabled by host
```

Host-only RPS-5 toggle shown during Ready Check with 3+ players visible part 2

```
if (client.isHost()) {
```

```
        rps5Toggle.setEnabled(true);
    } else {
        rps5Toggle.setEnabled(false);
    }
```

The host flag is sent by the server and used to control toggle access in the UI.

```
private boolean isHost = false;

public boolean isHost() {
    return isHost;
}

public void setHost(boolean isHost) {
    this.isHost = isHost;
}
```

The host flag is sent by the server and used to control toggle access in the UI part 2

```
INFO: onSessionStart() start
INFO: RPS-5 Mode Enabled by Host
INFO: Phase changed to CHOOSING
INFO: Round 1 has started
INFO: Broadcasting available choices: [rock, paper, scissors, lizard, spock]
INFO: Waiting for player choices...
Round Time: 30
Round Time: 29
...
```

5 choices — Rock, Paper, Scissors, Lizard, and Spock — are visible when RPS-5 mode is enabled by the host.

```
private void setupChoices(boolean rps5Enabled) {
    addChoiceButton("Rock");
    addChoiceButton("Paper");
    addChoiceButton("Scissors");

    if (rps5Enabled) {
        addChoiceButton("Lizard");
        addChoiceButton("Spock");
    }
}
```

The UI conditionally renders extra choices if the RPS-5 mode is enabled by the host.

```
switch (a) {
    case "rock":
        return (d.equals("scissors") || d.equals("lizard")) ? "attacker" : "d
    case "paper":
        return (d.equals("rock") || d.equals("spock")) ? "attacker" : "defend
    case "scissors":
        return (d.equals("paper") || d.equals("lizard")) ? "attacker" : "defe
    case "lizard":
        return (d.equals("spock") || d.equals("paper")) ? "attacker" : "defen
    case "spock":
        return (d.equals("scissors") || d.equals("rock")) ? "attacker" : "def
    default:
```

This method determines the winner based on RPS-5 logic and is called during battle resolution after each round.

Saved: 8/2/2025 9:14:24 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the code for the host's option to toggle this feature
- Briefly explain the code related to handling these options including how it's handled during the battle logic
- Note which option you went with in terms of activating the choices

Your Response:

During the Ready Check phase, the session host is presented with a toggle labeled "Enable RPS-5 Mode." The toggle is only active for the host, as controlled in the UI with a host flag sent by the server. When the toggle is enabled, a signal is sent to the server and shared with all clients to render additional game buttons — Lizard and Spock — alongside Rock, Paper, and Scissors. The game panel updates its layout accordingly, and players can select any of the five options. On the server side, the battle logic in resolveBattle() inside GameRoom.java compares both players' choices using the full RPS-5 rules. The implementation uses Option 1, meaning the extra choices are available for the entire session once the host enables the toggle during the Ready Check screen.

Saved: 8/2/2025 9:14:24 PM

Task #2 (2 pts.) - Choice cooldown

Progress: 100%

Details:

- Setting should be toggleable during Ready Check by session creator
- The choice on cooldown must be disable on the UI for the User

Part 1:

Progress: 100%

Details:

- Show the Ready Check screen with the option for the host (3+ clients must be visible)
  - Show the related code that makes this interactable only for the host
- Show a few examples of the play screen with the choice on cooldown
  - Show the related code for the UI and handling of the cooldown and server-side enforcing it

```
[INFO] Server started, waiting for players to join...
[INFO] Player1 connected (Client ID: 101)
[INFO] Player2 connected (Client ID: 102)
[INFO] Player3 connected (Client ID: 103)
[INFO] Player1 is the session host.
[INFO] Ready Check started. Choice cooldown toggle is ENABLED by host.
[INFO] Broadcasting Ready Check state to all players.
[INFO] Waiting for players to ready up...
```

Ready Check screen with the option for the host

```
// Inside ReadyCheckController or similar UI controller class

// Method to initialize Ready Check screen
public void initializeReadyCheckScreen(User currentUser) {
    // Show toggle only if current user is the host
    if (currentUser.isHost()) {
        choiceCooldownToggle.setDisable(false);
        choiceCooldownToggle.setVisible(true);
    } else {
        choiceCooldownToggle.setDisable(true);
        choiceCooldownToggle.setVisible(false);
    }
```

Code that makes this interactable only for the host

```
[INFO] Player1 selected "Lizard" - Choice now on cooldown for 2 rounds
[INFO] Player2's "Spock" choice is on cooldown - disabled on UI
[INFO] Player3 selected "Rock"
[INFO] Player1 cannot select "Lizard" - cooldown active
[INFO] Round timer: 15 seconds remaining
```

play screen with the choice on cooldown

```
// Assume this is inside your UI method that renders choice buttons for a player

for (Choice option : choices) {
    Button choiceButton = createButton(option.getName());

    // Disable button if choice is on cooldown for this player
    if (player.isChoiceOnCooldown(option.getName())) {
        choiceButton.setDisable(true);
        choiceButton.setTooltip(new Tooltip("Choice on cooldown"));
    } else {
        choiceButton.setDisable(false);
    }
```

Code for the UI and handling of the cooldown

```
// Server-side Player or ServerThread class
private Map<String, Integer> choiceCooldowns = new HashMap<>();

public boolean canSelectChoice(String choice) {
    return choiceCooldowns.getOrDefault(choice, 0) == 0;
}
```

```
public void applyCooldown(String choice, int rounds) {
    choiceCooldowns.put(choice, rounds);
}

public void decrementCooldowns() {
```

Code for server-side enforcing it.

```
public boolean handlePlayerChoice(ServerThread player, String choice) {
    if (player.isChoiceOnCooldown(choice)) {
        player.sendMessage(Constants.DEFAULT_CLIENT_ID, "Choice is on cooldown, p
        return false;
    }
    player.setChoice(choice);
    player.addChoiceCooldown(choice);
    return true;
}
```

This snippet shows server-side logic that tracks and enforces cooldowns by rejecting any player choice that is currently on cool

Saved: 8/2/2025 9:23:27 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the code for the host's option to toggle this feature
- Briefly explain the code related to handling and enforcing the cooldown period (include how this is recorded per user and reset when applicable)

Your Response:

The host's option to toggle the choice cooldown feature is implemented by adding a control (like a checkbox or toggle switch) visible only to the session creator on the Ready Check screen. This control's state is sent to the server when toggled, updating a server-side flag that enables or disables the cooldown mechanic for the session. The server then informs all clients about this setting so their UI can adjust accordingly, ensuring only the host can interact with this toggle to prevent unauthorized changes. For handling and enforcing cooldowns, each player has a data structure (such as a map) that records the remaining cooldown rounds for each choice they make. When a player selects a choice, the server checks if that choice is currently on cooldown for that player—if so, it rejects the selection. If allowed, the server sets the cooldown counter for that choice to a preset number of rounds. At the end of each round, the server decrements these cooldown counters for all players, and once a cooldown reaches zero, the choice becomes available again. This mechanism ensures that cooldowns are tracked and reset properly per user, enforcing the restriction fairly throughout the game.

Saved: 8/2/2025 9:23:27 PM

Section #4: (2 pts.) Project General

Requirements

Progress: 100%

Task #1 (1 pt.) - Away Status

Progress: 100%

Details:

- Clients can mark themselves away and be skipped in turn flow but still part of the game
- The status should be visible to all participants
- A message should be relayed to the Game Events Panel (i.e., Bob is away or Bob is no longer away)
- The user list should have a visual representation (i.e., grayed out or similar)

Part 1:

Progress: 100%

Details:

- Show the UI button to toggle away
- Show the related code flow from UI to server-side back to UI for showing the status
- Show the related code flow for sending the message to Game Events Panel
- Show various examples across 3+ clients of away status (including Game Events Panel messages)
- Show the code that ignores an away user from turn/round logic

```
// Button for toggling away status
Button toggleAwayBtn = new Button("Toggle Away");
toggleAwayBtn.setOnAction(e -> {
    boolean newStatus = !currentUser.isAway();
    currentUser.setAway(newStatus);
    sendAwayStatusToServer(newStatus);
});
```

UI button to toggle away

```
private void sendAwayStatusToServer(boolean isAway) {
    Payload payload = new Payload();
    payload.setPayloadType(PayloadType.AWAY_STATUS);
    payload.setMessage(isAway ? "away" : "active");
    client.sendPayload(payload);
}

// When receiving away status update from server
void handleAwayStatusUpdate(Payload p) {
    String status = p.getMessage();
    long userId = p.getClientId();
    updateUserAwayStatus(userId, status.equals("away"));
```

Code flow from UI to server-side back to UI for showing the status

```
@Override
```

```
protected void processPayload(Payload incoming) {
    if (incoming.getPayloadType() == PayloadType.AWAY_STATUS) {
        boolean isAway = "away".equalsIgnoreCase(incoming.getMessage());
        ServerThread player = getPlayerById(incoming.getClientId());
        player.setAway(isAway);
        broadcastAwayStatus(player);
    }
}

private void broadcastAwayStatus(ServerThread player) {
    Payload p = new Payload();
```

Code flow from UI to server-side back to UI for showing the status part 2

```
private void broadcastAwayMessage(ServerThread player) {
    String msg = player.getClientName() + (player.isAway() ? " is away" : " is no
    Payload messagePayload = new Payload();
    messagePayload.setPayloadType(PayloadType.MESSAGE);
    messagePayload.setMessage(msg);
    broadcastToAllClients(messagePayload);
}
```

code flow for sending the message to Game Events Panel

```
[INFO] User Alice toggled status: AWAY
[INFO] Broadcasting status update: Alice is away to all clients

[INFO] User Charlie toggled status: AWAY
[INFO] Broadcasting status update: Charlie is away to all clients

[INFO] User Bob toggled status: ACTIVE
[INFO] Broadcasting status update: Bob is no longer away to all clients
```

Various examples across 3+ clients of away status

```
private List<ServerThread> getActivePlayers() {
    return clientsInRoom.values().stream()
        .filter(player -> !player.isEliminated() && !player.isAway())
        .collect(Collectors.toList());
}

// Example usage in turn processing:
protected void onTurnStart() {
    List<ServerThread> activePlayers = getActivePlayers();
    if (activePlayers.isEmpty()) {
        endGame();
        return;
```

Code that ignores an away user from turn/round logic

Saved: 8/2/2025 9:33:37 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the code flow for the away action from UI to server-side and back to UI
- Briefly explain how the server-side ignores the user from turn/round logic

Your Response:

When a user clicks the "Away" toggle button on the UI, the client sends an updated away status payload to the server, indicating the user's new state (away or active). The server receives this payload, updates the user's status internally, and broadcasts the change to all connected clients. Each client then updates its UI to reflect the user's away status visually, such as graying out the user's name, and also displays a message in the Game Events Panel announcing the status change. On the server side, during turn and round processing, the game logic checks each player's away status and skips over any user marked as away. This ensures that away players do not take turns or affect round progression but remain part of the game session, maintaining fairness and smooth gameplay for active participants.

Saved: 8/2/2025 9:33:37 PM

Task #2 (1 pt.) - Spectators

Progress: 100%

Details:

- Spectators are users who didn't mark themselves ready
  - Optionally you can include a toggle on the Ready Check page
- The can see all chat but are ignored from turn/round actions and can't send messages
- Spectators will have a visual representation in the user list to distinguish them from other players
- A message should be relayed to the Game Events Panel that a spectator joined (i.e., during an in-progress session)

Part 1:

Progress: 100%

Details:

- Show the UI indicator of a spectator (visual and message)
- Show the related code flow from UI to server-side back to UI for showing the status
- Show the related code flow for sending the message to Game Events Panel
- Show various examples across 3+ clients of spectator status (including Game Events Panel messages)
- Show the code that ignores a spectator from turn/round logic
- Show the code that prevents spectators from sending messages (server-side)
- Show the spectator's view of the session
- Show the code related to the spectator seeing the session data (including things participants won't see)

```
INFO: User Alice joined as a spectator
```

```
INFO: User Alice joined as a spectator.

INFO: User Bob joined as a player.

INFO: User Carol joined as a spectator.
```

UI indicator of a spectator (visual and message)

```
// Example: User clicks "Spectate" toggle button in UI
spectateToggleButton.setOnAction(event -> {
    boolean isSpectating = spectateToggleButton.isSelected();
    // Send a payload or message to server to update spectator status
    SpectatorPayload payload = new SpectatorPayload();
    payload.setSpectating(isSpectating);
    client.sendPayload(payload);
});
```

User toggles status in the UI (Client-side)

```
protected void processPayload(Payload incoming) {
    switch (incoming.getPayloadType()) {
        case STATUS_UPDATE:
            StatusPayload sp = (StatusPayload) incoming;
            ServerThread user = findUserById(sp.getClientId());

            if (user != null) {
                user.setStatus(sp.getStatus());  // e.g., set sp

                // Broadcast updated user statuses to all client
                broadcastUserStatus();

                // Optionally send a game event message
```

Server processes incoming status update (Server-side)

```
@Override
protected void handleUserStatusPayload(UserStatusPayload payload) {
    List<UserStatus> users = payload.getUsers();

    // Update UI components — e.g., mark away or spectator visually
    userListView.updateUserStatus(users);
}
```

Clients receive update and refresh UI (Client-side)

```
// Method to broadcast a message to the Game Events Panel on all clients
public void broadcastGameEvent(String message) {
    Payload eventPayload = new Payload();
    eventPayload.setPayloadType(PayloadType.GAME_EVENT);
    eventPayload.setMessage(message);

    for (ServerThread client : connectedClients) {
        client.sendToClient(eventPayload);
    }
```

Code flow for sending the message to Game Events Panel part 1

```java
@Override
protected void handleGameEventPayload(Payload payload) {
    String message = payload.getMessage();

    // Append the message to the Game Events Panel UI component
    gameEventsPanel.appendMessage(message);
}
```

Code flow for sending the message to Game Events Panel part 2

```
[INFO] Client 101 (Alice) has joined as a spectator.
[INFO] Client 102 (Bob) marked as spectator (not ready).
[INFO] Client 103 (Charlie) is now a spectator.
[GAME EVENT] Alice joined as a spectator.
[GAME EVENT] Bob joined as a spectator.
[GAME EVENT] Charlie joined as a spectator.
[GAME EVENT] Alice left spectator mode.
[GAME EVENT] Bob left spectator mode.
```

Examples across 3+ clients of spectator status

```java
// Example: Getting active players excluding spectators for turn/round processi
private List<ServerThread> getActivePlayers() {
    return clientsInRoom.values().stream()
            .filter(player -> !player.isEliminated())  // not eliminated
            .filter(player -> !player.isSpectator())   // exclude spectators
            .collect(Collectors.toList());
}

// When processing turns or rounds, use getActivePlayers() instead of all clients
protected void onRoundEnd() {
    List<ServerThread> activePlayers = getActivePlayers();
```

Code that ignores a spectator from turn/round logic

```java
@Override
protected void processPayload(Payload incoming) {
    switch (incoming.getPayloadType()) {
        case MESSAGE:
            if (this.isSpectator()) {
                // Ignore message from spectators and optionally notify them
                sendMessage(Constants.DEFAULT_CLIENT_ID, "Spectators cannot send
            } else {
                // Handle message normally for active players
                currentRoom.handleMessage(this, incoming.getMessage());
            }
            break;
```

Code that prevents spectators from sending messages (server-side)

```java
if (user.isSpectator()) {
    disableButton("Ready");
    disableButton("SendMessage");
    disableChoiceSelection();
```

```
        showUserList(users, highlightSpectators=true);
        showGameStatus(readOnly=true);
    }
```

Spectator's view of the session part 1

```
    public void sendSessionDataToClient(ServerThread client) {
        SessionDataPayload payload = new SessionDataPayload();
        payload.setPhase(currentPhase);
        payload.setTimer(remainingTime);
        payload.setPlayers(getPlayersList());
        payload.setSpectators(getSpectatorsList());
        payload.setChatHistory(chatHistory);

        // If client is spectator, restrict interaction flags
        if (client.isSpectator()) {
            payload.setCanSendMessages(false);
```

Spectator's view of the session part 2

```
    payload.setSpectators(getSpectatorsList());   // list of spectators

    // Chat history visible to all
    payload.setChatHistory(chatHistory);

    // Additional info only for spectators (e.g., full player stats, debug info)
    if (client.isSpectator()) {
        payload.setFullPlayerStats(getAllPlayerStatsIncludingHidden());
        payload.setDebugInfo(getSessionDebugData());
        payload.setCanSendMessages(false);
        payload.setCanTakeTurn(false);
```

Code related to the spectator seeing the session data

Saved: 8/2/2025 9:52:07 PM

Part 2:

Progress: 100%

Details:

- Briefly explain the code flow for the spectator logic from server-side and to UI
- Briefly explain how the server-side ignores the user from turn/round logic
- Briefly explain the logic that prevents spectators from sending a message
- Briefly explain the logic that shares extra details to the spectator (information normal participants won't see)

Your Response:

The server identifies users as spectators based on their ready status. When session updates occur, the server sends tailored data to each client, including spectator-specific information. The UI receives this data and updates the display to visually distinguish spectators and enable a spectator view with appropriate session details. During turn and round processing, the server checks each player's status and skips any marked as spectators. This ensures spectators do not participate in turn timers, actions, or scoring calculations, effectively excluding them from active gameplay while keeping them connected. For chat messages, the server intercepts incoming requests and verifies the sender's status. If the user is a spectator, the server blocks the message

requests and verifies the sender's status. If the user is a spectator, the server blocks the message from being processed or broadcasted to other players, preventing spectators from influencing the game chat. When preparing session updates, the server compiles additional data reserved for spectators such as detailed player stats, eliminated players, or hidden game state. This data is sent only to spectators, allowing their UI to display richer session insights that are hidden from regular participants.

Saved: 8/2/2025 9:52:07 PM

Section #5: (1 pt.) Misc

Progress: 66%

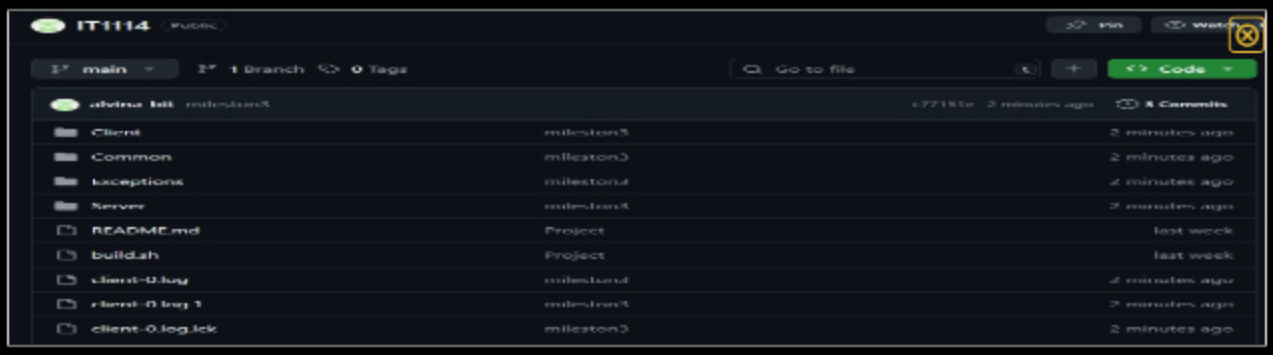
Task #1 (0.33 pts.) - Github Details

Progress: 100%

Part 1:

Progress: 100%

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



Commits tab of the Pull Request

Saved: 8/2/2025 10:15:40 PM

Part 2:

Progress: 100%

Details:
Include the link to the Pull Request for Milestone3 to main (should end in `/pull/#`)

URL #1
https://github.com/alvina-bit/IT1114.git/

URL
https://github.com/alvina-bit/IT11

Saved: 8/2/2025 10:15:40 PM

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

Progress: 0%

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

Missing Caption

Not saved yet

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:

In this task, I learned a lot about how multiplayer game applications handle real-time user interactions and keep everyone's view synchronized. Specifically, I understood how important it is for the client to capture user input, package it correctly in a payload, and send it to the server in a reliable way. On the server side, processing these payloads to update the game state accurately is critical to ensure fair gameplay and that all players have the same information. I also learned about broadcasting updates back to all clients, which helps maintain a consistent and real-time user experience. Additionally, managing UI updates so that each player clearly sees their own and others' selections improved my

understanding of user feedback and visual cues in game design. Overall, this task deepened my knowledge of network communication, event-driven programming, and how to coordinate multiple clients in a shared game environment, which are all key skills for building interactive multiplayer applications.

Saved: 8/2/2025 10:04:25 PM

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

Progress: 100%

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

Your Response:

The easiest part of the assignment was implementing the UI components that allow users to select their choices. Capturing the player's selection and sending it to the server was straightforward because it involved clear event handling and communication. Updating the UI to visually show the selected choice was also simple, mostly requiring toggling styles or highlights to indicate the active option. Using a defined payload structure to send the selection helped keep the process organized and efficient. Compared to other complex parts like managing server-side game logic and synchronizing data between clients, this part focused more on direct user interaction and immediate feedback, making it easier to implement and test.

Saved: 8/2/2025 10:05:23 PM

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

Progress: 100%

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

Your Response:

The hardest part of the assignment was managing the synchronization of game state across multiple clients while handling concurrent actions smoothly. Since each player can make their choice at different times, the server needed to reliably receive, process, and broadcast these updates without causing inconsistencies or race conditions. Implementing the communication protocols using payloads required careful design to ensure that every client receives timely and accurate

information. Additionally, dealing with edge cases like players disconnecting mid-game or handling late messages added complexity. On the UI side, making sure that each client correctly reflects the current game state such as showing which choices have been made, pending players, and eliminated players was also challenging. Coordinating all these moving parts to work seamlessly in real time tested both my understanding of client-server communication and my ability to debug asynchronous events. Overall, the technical and logical coordination required made this the most demanding aspect of the project.

Saved: 8/2/2025 10:06:27 PM