Battleship Game – Final Report

Intisar Bel Meddeb Kalile Horri
intisar.belmeddeb@studio.unibo.it Kalilehorri@studio.unibo.it

A.Y. 2024/25

Abstract

Abstract. This project implements a networked, turn-based Battleship game in Python 3.13 using UDP sockets. Two players connect to a central server, place their fleets on a 10×10 grid, and fire at each other's ships. The server mediates all messages: connection, shot requests, hit/miss verification, turn control, and game termination. The architecture demonstrates low-latency UDP communication, error handling for invalid messages, and modular code design for client and server components.

Contents

1 Introduction **2**

2 Objectives **2**

3 Requirements **2**

3.1 Functional Requirements 2

3.2 Non-Functional Requirements 3

3.3 Implementation Requirements 4

4 Technology Choices **5**

5 Design **6**

5.1 Architecture 6

5.2 Infrastructure 6

5.3 Modelling 7

5.4 Interaction 10

6 Implementation Details **14**

7 Validation **17**

7.1 Manual Gameplay Testing 17

7.2 Corner Case Testing 18

8 Usage Scenarios **20**

9 Usage Example **22**

10 Conclusion and Future Work **26**

11 What We Learned **27**

1 Introduction

Building interactive, real-time applications over unreliable networks presents unique challenges in synchronization, error handling, and user experience. The **Battleship Game** project addresses these challenges by recreating the classic two-player naval combat in a distributed setting, where each participant maintains a private fleet layout and exchanges shot coordinates through a central coordinator.

Beyond mere gameplay, this implementation serves as a hands-on exploration of:

- **Concurrency Control**: Coordinating turn order and ensuring only one player acts at a time.
- **Network Reliability**: Managing message loss or duplication inherent to UDP, with retries and validation logic.
- **Modular Design**: Separating core functions—connection management, board handling, shot adjudication—into distinct components for clarity and extensibility.
- **User Interaction**: Delivering concise console prompts and clear visualizations of both personal ship placements and shot histories.

This report outlines the project's objectives and requirements (Section 3), followed by the main technology choices (Section 4) and the system's design, including architecture, infrastructure, and interaction patterns (Section 5). We then present the implementation strategy (Section 6) and validate the solution through gameplay and error handling tests (Section 7). Finally, practical usage scenarios are discussed (Sections 8–9), before concluding with a summary of the results, future development opportunities (Section 10), and reflections on the knowledge gained during the project (Section 11).

2 Objectives

- Implement a multiplayer Battleship game using UDP.
- Manage turn-based firing, hit/miss verification, and game state synchronization.
- Design modular client and server code for maintainability.

3 Requirements

3.1 Functional Requirements

1. **Core Gameplay**

1. The system shall allow two players to connect via UDP and join a game room.
2. Players shall place ships either manually or using predefined layouts.
3. The game shall alternate turns between players automatically.
4. The system shall validate hits/misses and update boards in real time.
5. The game shall end when all ships of one player are sunk (`has_game_ended()`).

2. **Communication Protocol**

    1. The server shall relay messages between clients using UDP with the following formats:
        - `"shoots (x,y);room_id"` (attack)
        - `"check(x,y);room_id"` (validation request)
        - `"result;True/False;room_id"` (validation response)
    2. The server shall enforce turn order and prevent out-of-turn actions.

3. **User Interface**

    1. The CLI shall display:
        - Player's ship board (with "X" for ships).
        - Attack board (with "H"/"M" for hits/misses).
        - Turn status ("Your turn" / "Enemy's turn").

4. **Game Management**

    1. The system shall generate unique room IDs for each game.
    2. Players may surrender by sending "`end`", triggering immediate game termination.

3.2 Non-Functional Requirements

1. **Low-Latency Communication**
    - Use UDP sockets to minimize round-trip time for shot and result messages.

2. **Robust Error Handling**
    - Invalid or malformed messages must be caught by `try/except` blocks; the server replies with an error string and ignores the invalid request.

3. **Modular Code Structure**
    - Separate concerns into distinct modules (`GameRoom`, server logic, client helpers, board setup, board maker) to ease maintenance and future extension.

4. **User-Friendly Console UI**
    - Clear prompts, board displays, and messages ("HIT!", "MISS!", "Your turn", "Enemy's turn") to guide non-technical users.
5. **Scalability of Game Instances**
    - Support multiple concurrent game rooms in memory without interference.
6. **Configuration Flexibility**
    - Allow easy adjustment of board size, ship sizes, and buffer limits via constants.
7. **Language and Environment**
    - Implemented in Python 3.13; must run on any OS with Python interpreter without additional dependencies.
8. **Reliability under Packet Loss**
    - While UDP may drop packets, critical messages (connect, start, result) should be retried or re-sent by clients/servers if no timely response.
9. **Security Considerations**
    - Do not trust client addresses blindly; validate room IDs and message formats to prevent spoofing or injection attacks.
10. **Performance Logging**
    - Server console must log key events (connections, shots, results, errors) for debugging and performance monitoring.

3.3 Implementation Requirements

IR-1 Programming Language:
The entire application must be developed in `Python 3.13`. This version was chosen for its built-in socket support, readability, and wide compatibility across platforms.

IR-2 Communication Protocol:
All communication between clients and the server must occur via the UDP protocol. UDP was selected for its low-latency characteristics, which are suitable for fast, turn-based games.

IR-3 User Interface Mode:
The game interface shall be entirely text-based (CLI). Players interact with the game using the terminal. No graphical or web-based interface is required.

IR-4 File Organization and Modularity:
The system must follow a modular code structure. The project is divided into the following components:

- `client.py, server.py` – main execution files.
- `client_helpers.py` – gameplay logic and state updates.
- `client_make_board.py, client_boardsets.py` – board creation.
- `GameRoom.py` – room and player management.

IR-5 Version Control:
All code shall be version-controlled using Git. A repository is maintained throughout the development cycle to track progress, ensure reproducibility, and support collaboration.

IR-6 Cross-Platform Execution:
The application must run on any operating system (Windows, Linux, macOS) as long as Python 3.13 is installed. No platform-specific packages or dependencies are allowed.

4 Technology Choices

To implement the Battleship Game, we adopted a lightweight client-server architecture using Python 3.13 as the primary language for both client and server components. Python was chosen for its simplicity, readability, and strong support for socket programming through its standard library.

For network communication, we opted for UDP sockets (`socket.SOCK_DGRAM`) to reduce latency during turn-based gameplay. Despite UDP's unreliability, our protocol includes basic error-handling mechanisms to tolerate occasional packet loss.

The game runs entirely in the terminal, using simple text-based interactions for displaying the boards, managing input, and guiding the player. This console-based approach ensured cross-platform compatibility and removed the need for external libraries.

Code is structured modularly, separating the server logic (`server.py`, `GameRoom.py`) from client-side functions (`client.py`, `client_helpers.py`, etc.), which improves maintainability and readability.

5 Design

5.1 Architecture

The architecture of the Battleship Game follows a simple client-server model using UDP sockets for communication. The system consists of three primary actors: two clients (players) and one central server that acts as a message broker and game coordinator.

The server is responsible for creating and managing game rooms, relaying messages between clients, enforcing turn order, and handling game termination. It listens continuously for incoming messages such as connection requests, firing coordinates, and result confirmations. Each room is represented by an instance of the `GameRoom` class, which holds the addresses of both players and tracks whether the game has started.

On the client side, each player runs a separate instance of `client.py`, which handles user interaction, input validation, and board rendering. The client modules are organized as follows:

- `client_helpers.py`: logic for detecting hits, updating boards, validating shots, and detecting the end of the game.
- `client_boardsets.py`: provides predefined ship layouts.
- `client_make_board.py`: allows the user to manually place ships via terminal prompts.

All communication is handled via UDP to minimize latency. Clients never communicate directly with each other—instead, they send and receive all information through the server, ensuring centralized control and game state integrity.

This modular architecture separates concerns across files and roles, making the system easier to understand, test, and extend in future iterations.

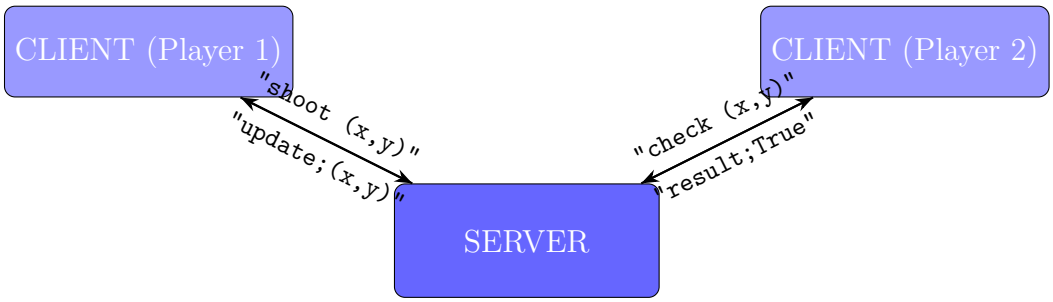

Figure 1: Architecture overview of the Battleship Game showing UDP message flow between clients through the central server

5.2 Infrastructure

All system components are distributed in a centralized topology. The UDP server and both clients are executed on the same physical machine during testing, communicating

over the local loopback interface (`127.0.0.1`).

Each client opens a UDP socket and sends messages to a known IP:port pair, with no dynamic discovery. There is no need for DNS, service registration, or balancing mechanisms due to the simplicity of the setup.

The server is single-threaded but manages multiple concurrent game rooms in memory, each encapsulated in a `GameRoom` instance. Since all data is in memory and sessions are isolated, the architecture supports multiple games in parallel without interference. The system avoids unnecessary complexity (firewalls, proxies, databases) to focus on its core goal: providing fast, real-time interaction between two players on a local network.

5.3 Modelling

Below is a detailed explanation of the system's class structure, based on the UML diagram. Each class encapsulates a specific concern in the overall client-server game flow and contributes to maintaining modularity and clarity in the architecture.

1. Client

The `Client` class represents a player's interface and serves as the main entry point for the user. It manages the game session from the player's perspective, including initializing the board, sending and receiving UDP messages, and handling turn logic. This class is also responsible for invoking helper functions to perform gameplay actions such as shooting or displaying boards.

Attributes:

- `my_room_id`: A unique identifier assigned by the server that associates the player with a specific game room.

Methods:

- `main()`: Starts the client loop, handles communication with the server, processes received messages (shoot, wait, update), and coordinates turn flow.
- `play_again()`: Allows the player to choose whether to replay after a game ends.

2. Server

The `Server` manages all client interactions, room assignments, and gameplay synchronization. It uses UDP to exchange messages with clients, controls turn order, validates actions, and broadcasts game results. All gameplay messages flow through the server, which guarantees correct sequencing and fair execution.

Attributes:

- `rooms`: A list of all active game rooms, each represented as a `GameRoom` instance.

Methods:

- `main()`: The main server loop; listens for incoming messages, interprets their intent (connect, shoot, result, end), and forwards them accordingly.
- `add_player_to_rooms()`: Allocates players to rooms dynamically upon connection. Automatically pairs players in available rooms.
- `update_game_started()`: Flags the beginning of a game when both players are ready.
- `find_room()`, `find_other_player_address()`: Internal search utilities to determine which room or player is associated with an incoming message.
- `remove_room_by_id()`: Removes a room once the game concludes or if a player quits.

3. GameRoom

The `GameRoom` class encapsulates the state of a single match. It keeps track of both players, the game's unique identifier, and whether the session is active. This class is lightweight and acts as a container for session metadata.

Attributes:

- `id`: A UUID string generated when the room is created to ensure uniqueness.
- `player1`, `player2`: Dictionaries containing the client socket addresses of the two players.
- `gameStarted`: Boolean flag indicating if the game has officially begun.

Methods:

- `__str__()`: Returns a formatted string displaying room ID and player information, useful for server logs and debugging.

4. ClientHelpers

This utility class provides the majority of gameplay logic on the client side. It is responsible for core functionalities such as hit detection, board updates, turn checking, and user interactions. It ensures that the main client class remains clean and focused on message routing.

Methods:

- `has_game_ended()`: Checks whether all ships have been destroyed.
- `is_ship_hit()`: Determines whether a shot hits a ship based on board state.
- `update_board()`, `generate_shots_board()`: Modify the visual board to reflect gameplay events.
- `choose_board()`, `get_shot()`: Handles input from the user to place ships and fire at opponents.
- `print_board()`, `show_rules()`, `clear_console()`: Provide readable output and terminal formatting to improve user experience.

5. ClientBoardsets

This module contains two hard-coded board layouts that players can select at the start of the game. These layouts ensure fair and validated ship positioning for players who choose not to manually place ships.

Attributes:

- `board1`, `board2`: Each is a 10x10 2D list representing valid Battleship ship placements.

6. ClientMakeBoard

The `ClientMakeBoard` class allows players to manually construct their ship layout. It validates ship sizes and positions to ensure no overlapping or out-of-bounds errors. It is called if the player declines to use a predefined board.

Methods:

- `print_board()`: Shows the current state of the player's board in the terminal.
- `place_ship()`, `is_valid_position()`: Logic for placing ships and validating input.
- `convert_to_X()`, `make_user_board()`: Finalizes and formats the board for use in gameplay.

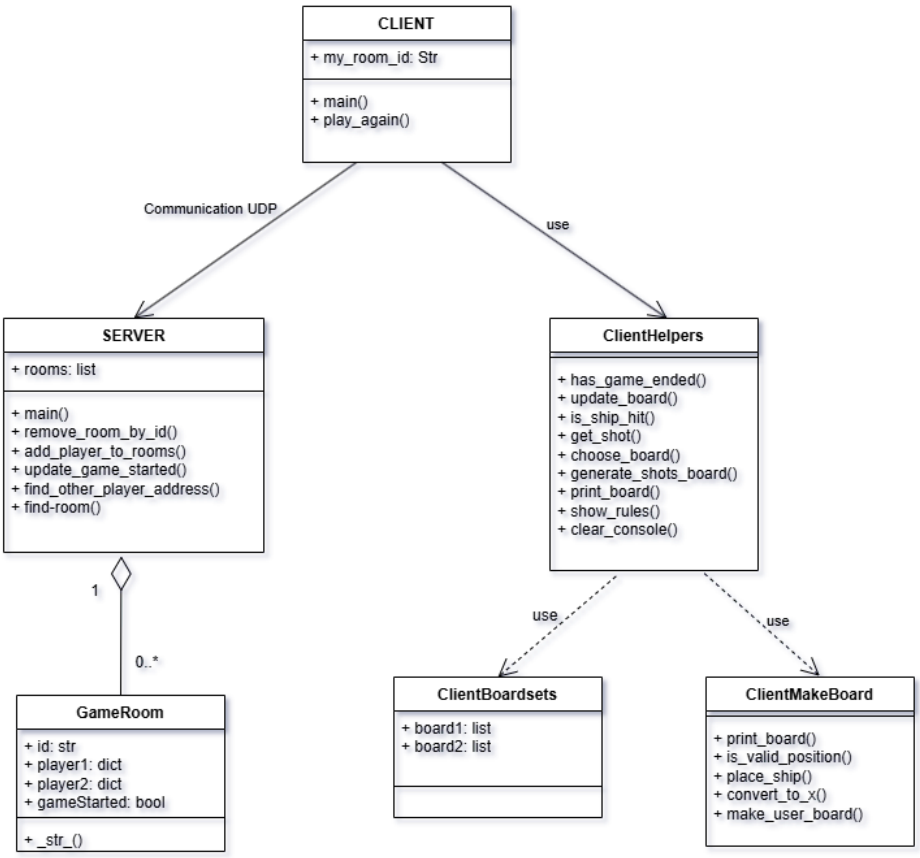

Figure 2: Class diagram representing the main entities of the Battleship game.

5.4 Interaction

The following sequence diagram illustrates the dynamic message exchange that occurs during a single gameplay turn in the Battleship Game. It captures how actions initiated

by a player are routed through the server, verified by the opponent, and result in a coordinated update for both parties.

In this scenario, Player 1 initiates a shot during their turn. The server receives the coordinates and relays a validation request to Player 2, who confirms whether the shot was a hit or miss. The server then updates the firing client and, depending on the result, determines whose turn is next.

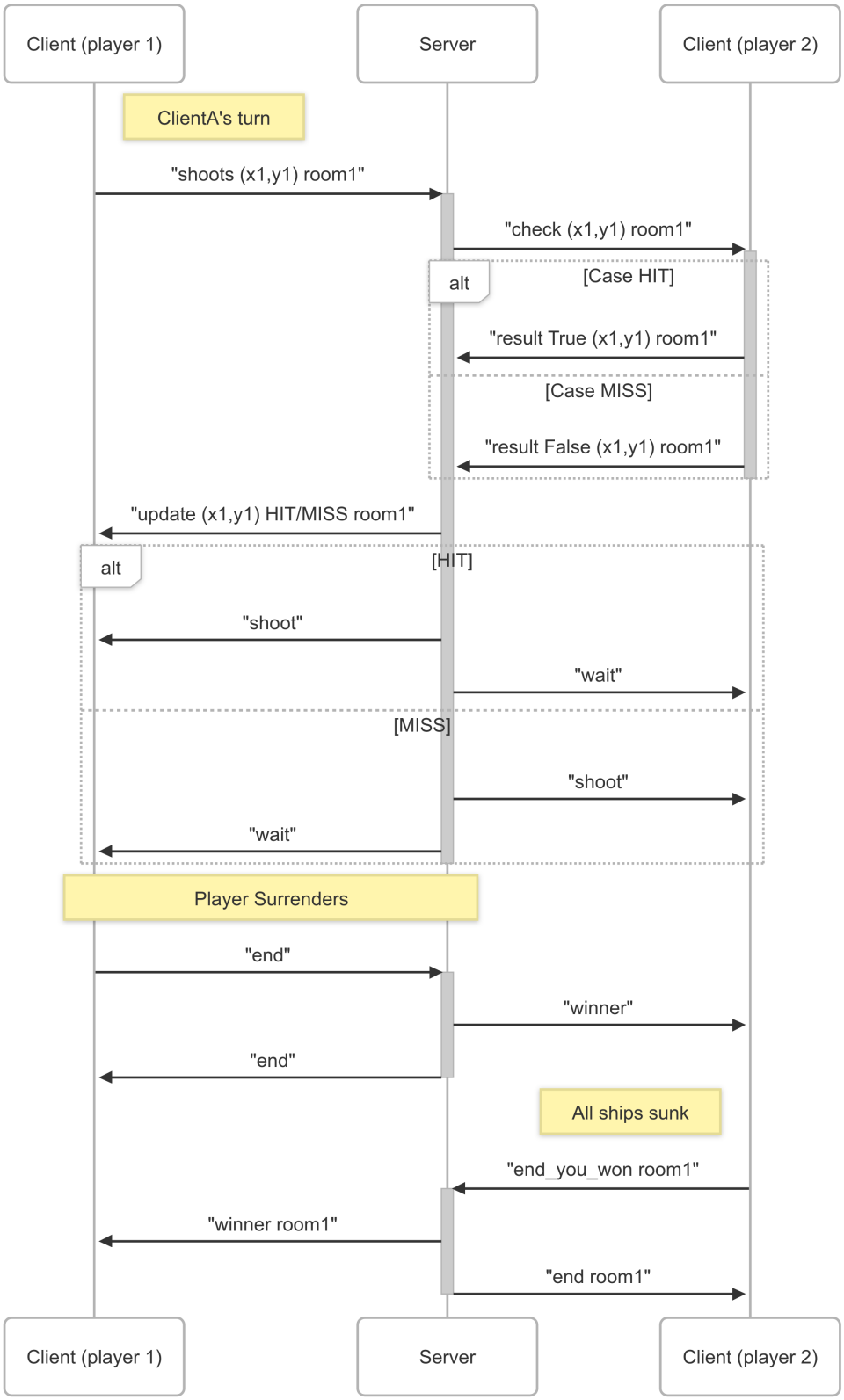

Figure 3: Sequence diagram of one turn in the Battleship game. Player 1 sends a shot, which is verified by Player 2 and managed by the server.

Gameplay Message Flow: The following outlines the precise flow of gameplay messages exchanged between the players and the server during a single shooting action. Each phase corresponds to actual behaviors implemented in the codebase.

1. **Phase 1: Shot Initiation:**
   - **Client (Player 1)** sends a `shoots (x,y), room_id` message containing:
     - Exact shot coordinates (1-10 grid system)
     - The unique `room_id`, to route the message to the correct game session.
   - **Server** performs validation based on:
     - Whether it is Player 1's turn privilege (as tracked by the `GameRoom` instance)
     - Validity of coordinates ($1 \leq x, y \leq 10$)
     - Ensuring that there are no duplicate shots (checks against stored attempts)
2. **Phase 2: Shot Validation:**
   - **Server** relays coordinates to **Client (Player 2)** as `check(x,y), room_id`
   - **Player 2** performs:
     - Local board lookup
     - Hit detection (value `"X"` = hit, `"M"` = miss)
     - Response:
       - `result, True, (x,y), room_id` (hit)
       - `result, False, (x,y), room_id` (miss)
3. **Phase 3: State Synchronization**
   - **Server** broadcasts unified game state:
     - `update, (x,y), HIT, room_id` (updates both players' shot boards)
     - `update, (x,y), MISS, room_id` (marks water on attacker's board)
   - Clients visually update:
     - Attacker: Appends `H/M` to shots board
     - Defender: Marks `H` on ship board if hit

   .
4. **Phase 4: Turn Transition:**
   - **IF HIT**:
     - Server maintains Player 1's turn (`shoot` privilege retained)

- Immediate new shot allowed (no wait period)

- **IF MISS**:
  - Turn is switched.
  - Server sends `shoot` to Player 2 and `wait` to Player 1.
  - The next player is now active.

5. **Phase 5: Termination Protocol:**

   - Victory is detected using `has_game_ended()` (20 hits = victory)
   - Server sends:
     - `end_you_won` to the winner.
     - `winner` to the opponent.
     - Followed by `end` to both clients.
   - The game room is then removed using `remove_room_by_id()`.

   Alternative: Player Surrender If a player voluntarily ends the game, the client sends an `end` message to the server. The server then:

   - Sends an `end` message to the opponent, indicating the game is over.
   - Deletes the associated game room using `remove_room_by_id()`.

   This ensures that the game concludes cleanly, even if one player exits before the end.

Key Observations:

- The server strictly controls the game flow, enforcing turn alternation and result consistency.
- Players never communicate directly — every action is mediated by the server.
- The design supports multiple simultaneous games by using unique room IDs in each message.
- Use of UDP requires that the client and server both implement validation logic for reliability, which is abstracted here for clarity.

6 Implementation Details

This section explains the key technical choices and runtime behaviors implemented in the Battleship project. We focus here on how core networking, data exchange, and control logic are handled.

Which network protocol is used?

The project uses the UDP (User Datagram Protocol) to enable low-latency, connectionless communication between clients and the central server. UDP was chosen over TCP due to its simplicity and performance, especially for turn-based games where rapid message delivery is preferable to guaranteed delivery.

The server binds to a local IP and port using Python's `socket` module:

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind((IP, PORT))
```

Listing 1: UDP server socket initialization

While UDP lacks built-in reliability, the project compensates by handling invalid or duplicate messages using logic encapsulated in `try/except` blocks and validation steps.

How is game data transmitted and interpreted?

All communication between clients and the server uses UTF-8 encoded strings following a lightweight, custom protocol.

Examples of transmitted messages include:

- `connect` — initial connection request
- `shoots(x,y);room_id` — player action to fire at a cell
- `check(x,y);room_id` — forwarded to opponent for verification
- `result;True/False;(x,y);room_id` — opponent's response
- `update;(x,y);HIT/MISS;room_id` — server propagates result
- `shoot / wait` — turn control messages
- `end / end_you_won / winner` — termination

This format is easy to parse using Python string methods and avoids external dependencies.

How is game state managed?

Game state is managed entirely in-memory. Each client maintains two local grids:

- **Ship Board** — where the player's fleet is located
- **Shots Board** — tracking hits and misses on the opponent

State updates are triggered by server messages. No persistent database is used, as the game is short-lived and session-based.

Victory detection is performed locally using the following logic:

```
def has_game_ended(board):
    h_counter = 0
    MAX_NUMBER_OF_X = 20
    for row in board:
        for place in row:
            if place == "H":
                h_counter += 1
    return h_counter == MAX_NUMBER_OF_X
```

Listing 2: Client-side victory check

How are turns and rules enforced?

Turn control is enforced server-side. When a room is full, the server randomly selects a starting player and begins sending `shoot` and `wait` instructions.

After each shot, the server determines whether to switch turns based on the result:

```
 if result == "False":
    server_socket.sendto("wait".encode('utf-8'), other_address)
    server_socket.sendto("shoot".encode('utf-8'), sender_address)
else:
    server_socket.sendto("wait".encode('utf-8'), sender_address)
    server_socket.sendto("shoot".encode('utf-8'), other_address)
```

Listing 3: Turn switching logic

Only one player is ever allowed to act at a time, avoiding race conditions.

How are games terminated?

There are two ways a game may end:

1. **Victory** — detected locally via `has_game_ended()`, the client sends `end_you_won`, and the server replies with `winner` and `end`.
2. **Surrender** — a player voluntarily sends `end`, and the opponent is notified immediately.

```
elif mess.lower() == "end":
    room, player_number = find_room(address)
    other_address = find_other_player_address(room, player_number)
    if other_address:
```

```
5           server_socket.sendto("end".encode('utf-8'), other_address)
6       print(f"Room", room.id, "|Game has ended by Player {address}.")
7       remove_room_by_id(room.id)
```

Listing 4: Surrender handling

Game rooms are deleted at the end of every session to free memory.

How are errors handled?

Although UDP does not guarantee delivery, the project implements minimal error-handling strategies:

- Invalid messages are caught using `try/except`.
- Duplicate coordinates are rejected client-side.
- Disconnects and timeouts do not crash the server; rooms are safely discarded.

More advanced features such as packet re-transmission or checksums could be considered in future improvements.

7 Validation

7.1 Manual Gameplay Testing

The validation process for the Battleship game was conducted manually by running both the server and two clients in terminal windows. The gameplay was tested in real conditions, simulating a complete game from board creation to win detection.

- **Test Setup:**
  - `server.py` is launched first to listen for incoming connections.
  - Two instances of `client.py` are started, each representing a player.
- **Tested Features:**
  - Manual and automatic ship placement (the player chooses to create the board manually or use a predefined layout).
  - Shot validation (rejecting coordinates outside the 1–10 range or invalid characters).
  - Detection of repeated shots (`This place has already been shot!` message).
  - Replay on successful HIT (the player gets an additional turn).

  - Game state update and synchronization across players.
  - Server-side logging and turn enforcement.

- **Sample Output:** Screenshots were taken during tests to show:
  - Error messages for invalid input
  - HIT/MISS results and board rendering
  - Server state updates and game flow coordination

7.2 Corner Case Testing

Several invalid inputs and edge cases were tested to ensure system robustness.

- Invalid characters (e.g., `AA` or symbols) in shot coordinates
- Out-of-bound coordinates (e.g., `22`, `-5`)
- Repeated shots on the same coordinate
- Leaving the game before finishing (surrender)
- Attempting to shoot during the opponent's turn

Why Manual Testing? The game is real-time and relies on UDP communication and user inputs in terminal, which makes automation non-trivial without mocks or simulation frameworks.

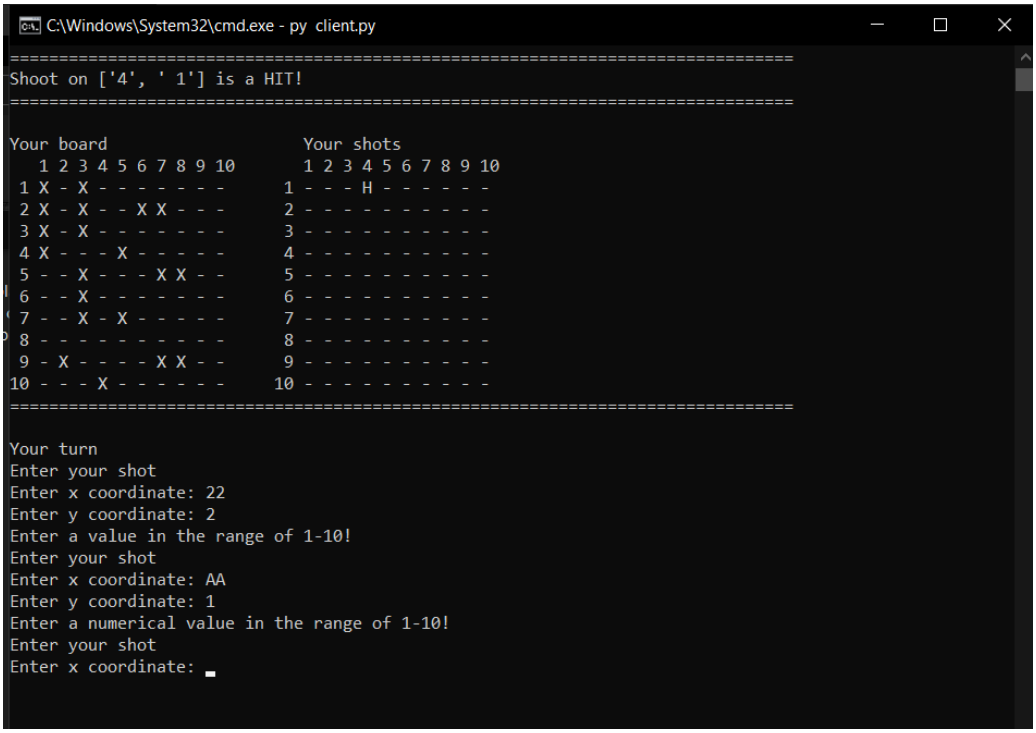

Figure 4: Invalid coordinates and input validation

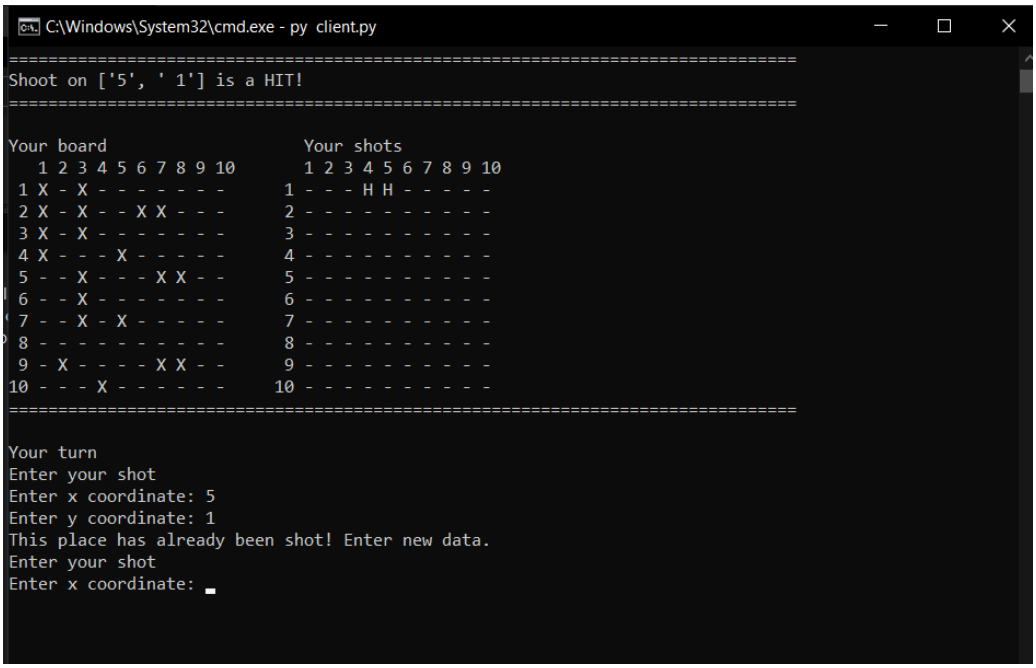

Figure 5: Repeat shot detection and error message

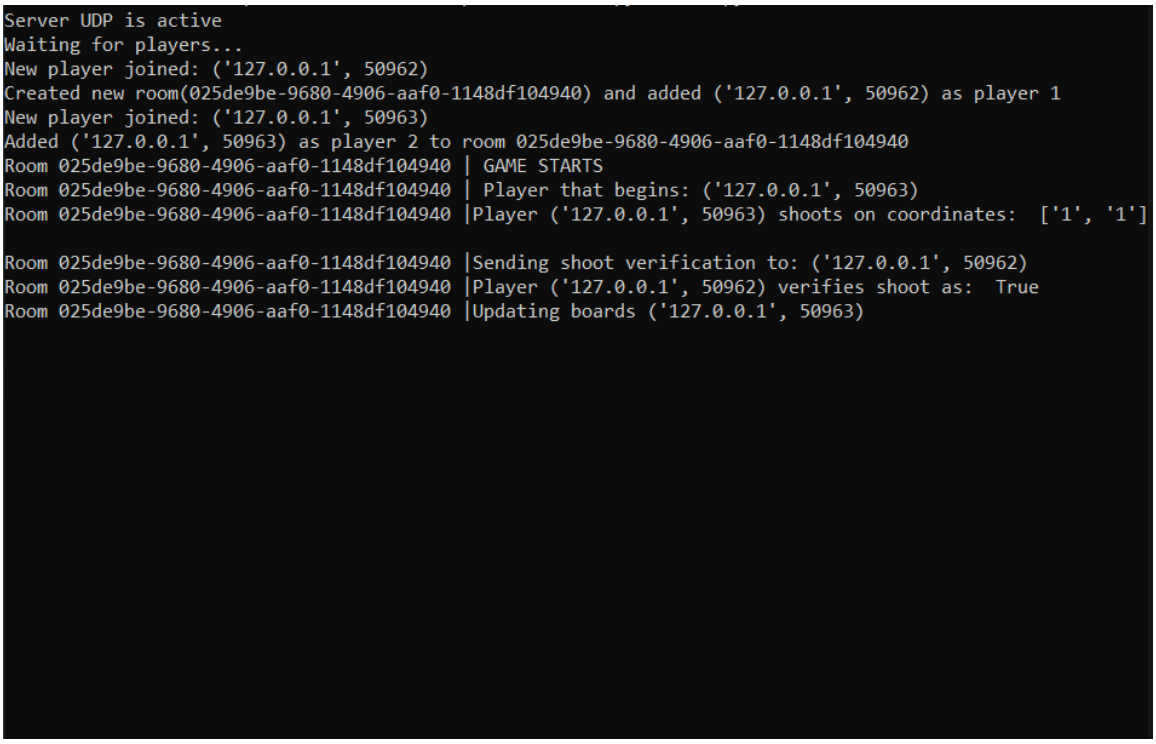

Figure 6: Server-side game flow during a HIT

8 Usage Scenarios

1. Connecting Two Players and Starting the Game

- **Scenario:** Two players launch the Battleship client to initiate a new game session via UDP. The server matches them and starts the game once both are connected.
- **Steps:**
  - Each player runs `client.py`.
  - The client sends a `connect` message via UDP to the server.
  - The server waits until two players are connected.
  - Upon pairing, the server creates a `GameRoom` instance and sends a `start; room_id; shoot/wait` message to each player to indicate whose turn it is.
- **Outcome:** A multiplayer session is successfully established, with players assigned to a room and the game ready to begin.

2. Ship Placement by Each Player

- **Scenario:** Once connected, each player must set up their board by placing ships manually or using a predefined layout.
- **Steps:**
  - The client displays the rules of the game using the `show_rules()` function.
  - The client displays a prompt to choose between manual or predefined placement.
  - If predefined is selected, a layout from `client_boardsets.py` is used.
  - If manual is selected, `client_make_board.py` guides the user to place ships one by one by entering coordinates.
  - Once complete, the board is saved locally, and the client waits for the opponent to finish setup.
- **Outcome:** Each player has a complete and valid 10×10 ship board, and the game is ready to proceed to the first move.

3. **Turn-Based Shooting Phase**

- **Scenario:** Players take turns shooting at each other's grid, following a turn-based structure controlled by the server.
- **Steps:**
  - Player with the turn sends `shoots(x,y);room_id` to the server.
  - Server relays `check(x,y);room_id` to the opponent.
  - Opponent checks local board and responds with `result;True/False;(x,y);room_id`.
  - Server updates both players with `update;(x,y);HIT/MISS;room_id`.
  - Server sends `shoot` or `wait` to determine next turn.
- **Outcome:** Each player alternates turns until one player hits all 20 ship segments, or the other surrenders.

4. **Game Termination by Victory or Surrender**

- **Scenario:** A game ends either when a player wins by hitting all ships or when the opponent surrenders manually.
- **Steps:**
  - If all ships are sunk, the losing client sends `end_you_won;room_id` to the server.
  - If a player quits, they send `end` to the server.
  - Server sends `winner;room_id` to the victor.
  - Server sends `end;room_id` to the loser or quitter.
  - Game room is cleared with `remove_room_by_id()`.
- **Outcome:** The game ends gracefully for both clients. The winner is notified, and players are prompted to play again.

9 Usage Example

This example describes a typical interaction between two players running the Battleship game using the terminal interface. It demonstrates how the system responds to player actions and manages the game flow in real-time using UDP communication.

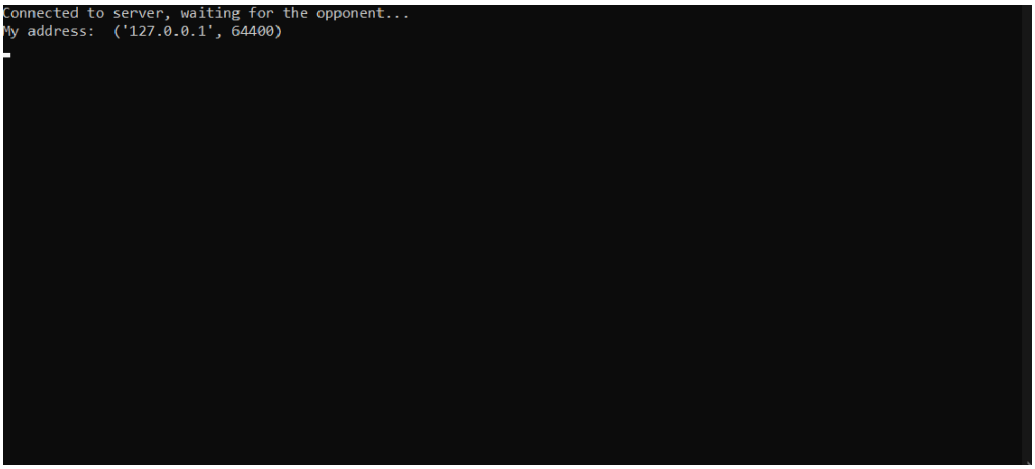

Figure 7: Connection phase Client.py

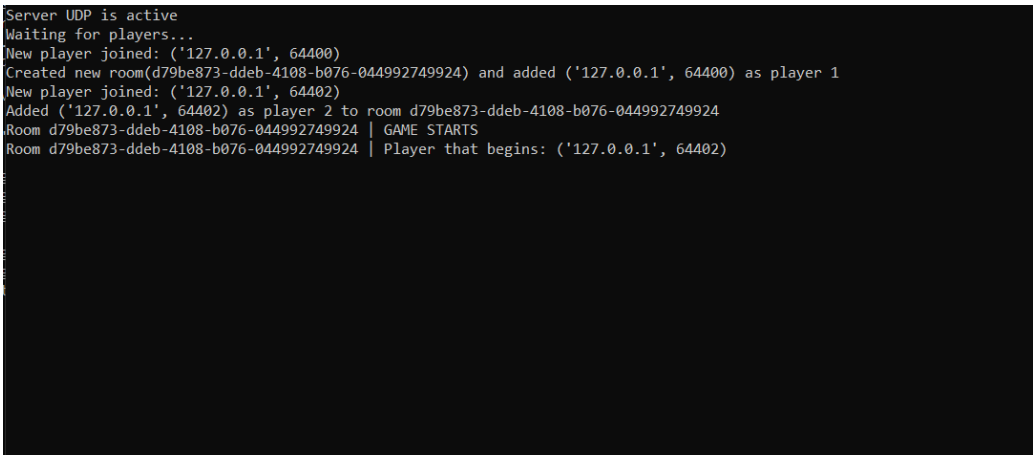

Figure 8: Room assignment Server.py

At the beginning of the game, each player launches the client script. The server receives their connection and creates a game room. Once both players are connected, a unique room ID is assigned, and the game begins.

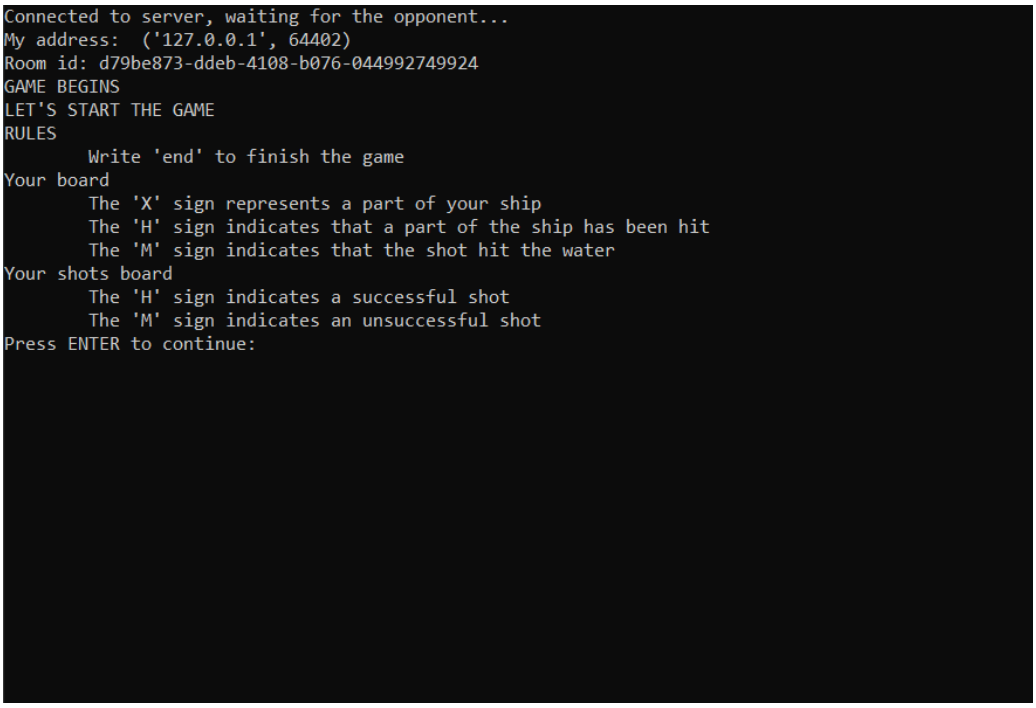

Figure 9: Rules shown before ship placement Client.py

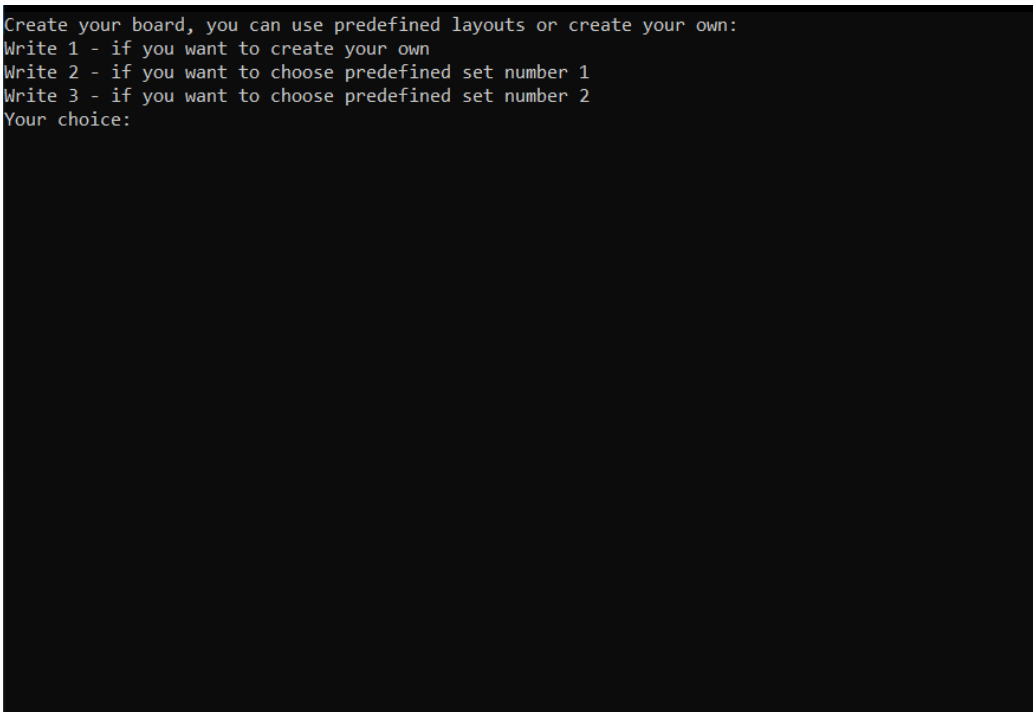

Figure 10: Prompt for ship placement: manual or predefined layout Client.py

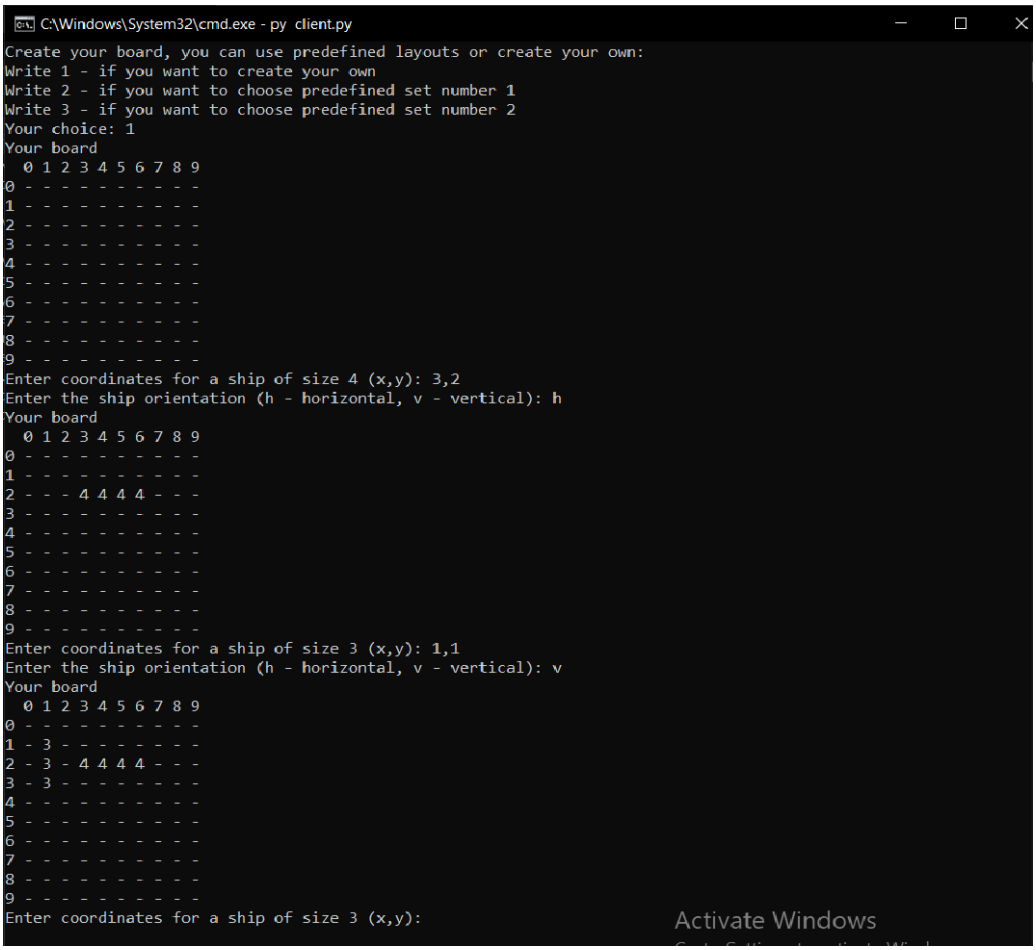

Figure 11: Manual ship placement initiated after selecting option 1

Before players place their ships, the system displays the game rules directly in the terminal. Players then choose whether to manually place ships or use a predefined layout. The board is rendered using ASCII characters, showing grid cells and ship positions.

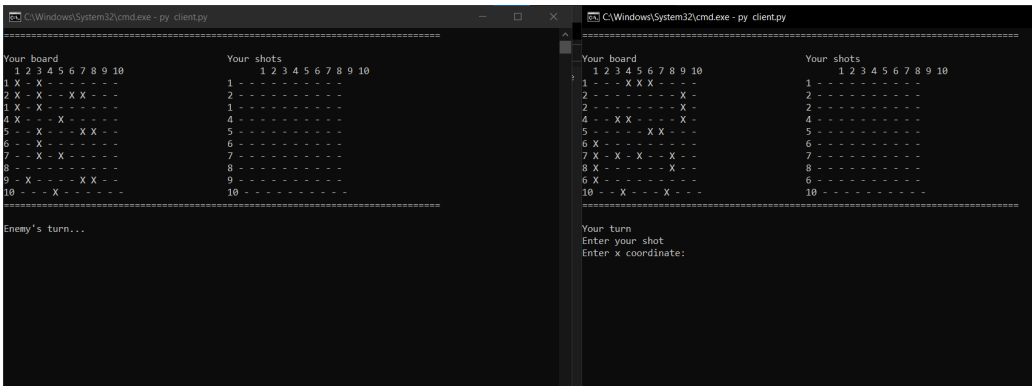

Figure 12: Gameplay phase with alternating turns

During the match, the server alternates turns between players. When one player fires at a coordinate, the system checks the opponent's board and responds with "HIT" or

"MISS". Players receive visual feedback on two boards: one showing their ships, the other showing their attempts.

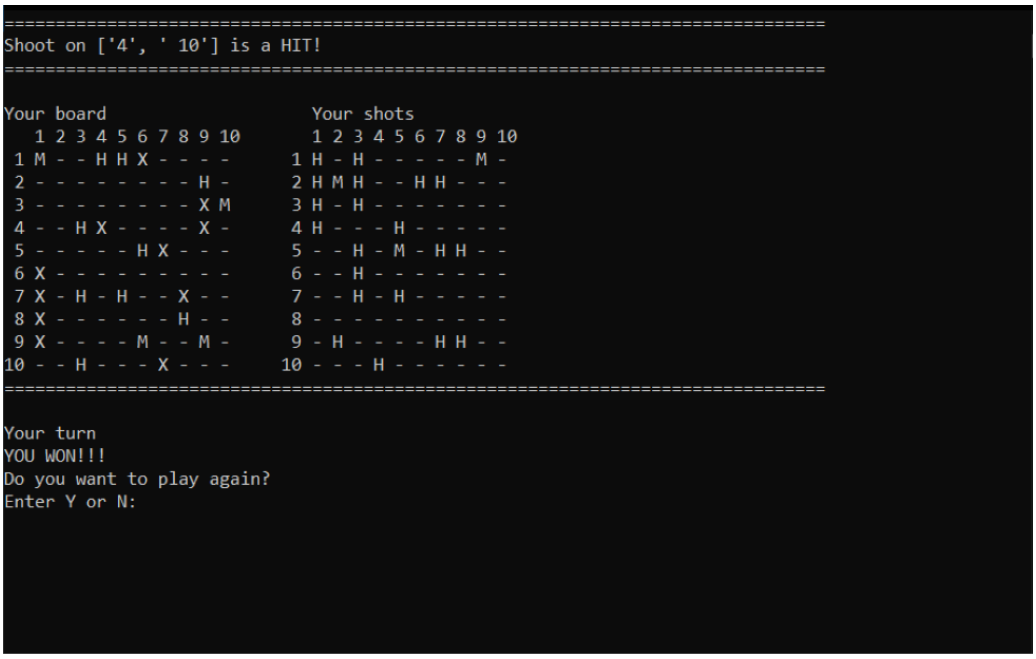

Figure 13: Victory message shown after winning

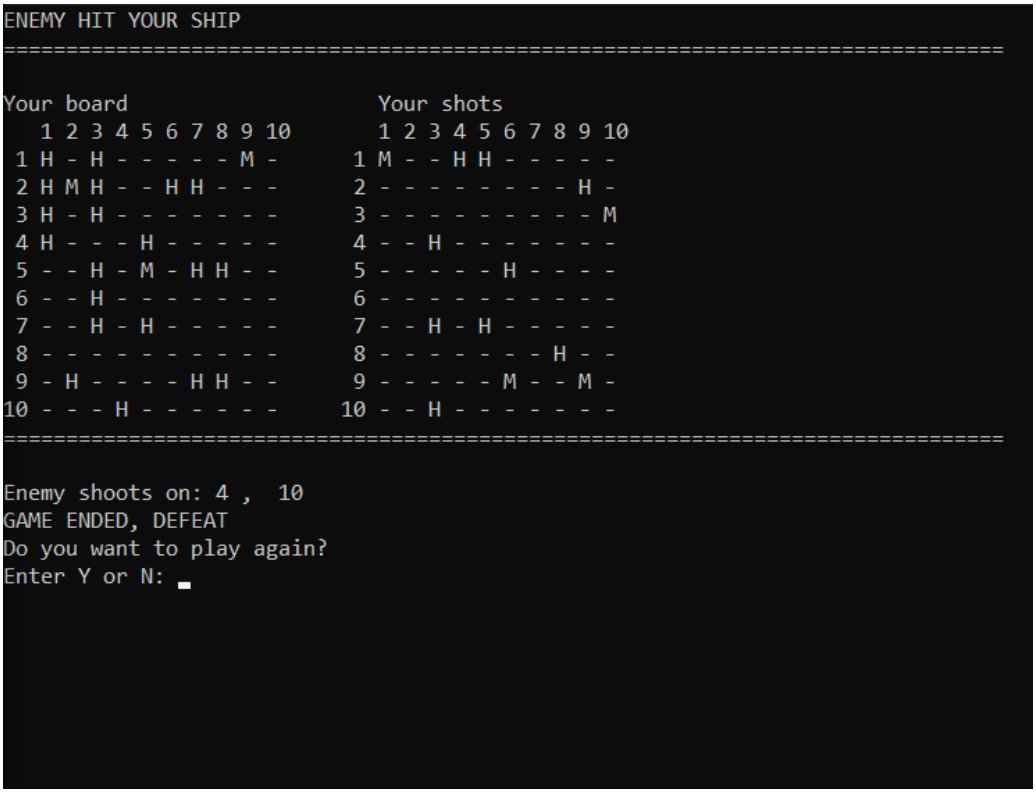

Figure 14: Loss message shown after losing

When all ships of one player are destroyed, the server sends an `end_you_won` message

to the winner and an `end` message to the loser. A final message appears in the terminal indicating the outcome, and players are prompted to play again or quit.

10 Conclusion and Future Work

This Battleship multiplayer game project successfully demonstrates the feasibility of building a real-time, event-driven application using a simple yet effective client-server architecture based on UDP sockets. Despite the connectionless and unreliable nature of the UDP protocol, we were able to design a stable and responsive system capable of handling synchronous interactions between two clients, validating gameplay logic, and maintaining a consistent shared game state. The implementation provided a valuable opportunity to put into practice fundamental principles of distributed systems—such as coordination, state management, turn handling, and fault tolerance.

Throughout the development process, a modular codebase was designed to clearly separate responsibilities between the server and clients. The server acts as a central authority that manages game rooms, assigns roles, controls the game loop, and forwards messages between players. On the client side, the application provides a text-based interface allowing players to either manually place ships or use a predefined layout. The client also performs input validation and visually renders the game state using terminal displays. Edge cases, such as duplicate shots, invalid inputs, and surrender events, are also properly managed through explicit feedback and server-side state tracking.

Looking ahead, several meaningful improvements could enhance the system. A graphical user interface (GUI) would significantly increase accessibility and user engagement. Additionally, introducing persistent storage—such as a database to store match history or player statistics—would extend the system's usefulness and support personalized experiences. Expanding support for multi-room setups and adopting more robust transport protocols (e.g., WebSockets or TCP) could further improve scalability and reliability for wider deployment in non-local networks.

From a distributed systems perspective, the project offered deep insight into managing real-time communication and ensuring consistency across nodes in a decentralized environment. We encountered challenges typical of distributed architectures, such as input synchronization, state updates, and user error recovery. By intentionally avoiding complex infrastructure (e.g., no firewalls, proxies, databases, or load balancers), we maintained focus on core distributed principles. Still, the architecture supports concurrent game sessions thanks to room-level isolation and in-memory state encapsulation—showcasing how even minimal designs can scale when built thoughtfully.

In conclusion, the Battleship game serves not only as a fun and functional multiplayer application but also as a concrete learning experience in designing distributed, interactive software. It lays a strong foundation for future iterations, and it reflects both the technical

and collaborative achievements of our development team.

11 What We Learned

Throughout the development of this Battleship Game project, we acquired substantial hands-on experience with the core concepts of distributed systems, real-time communication, and client-server synchronization. Designing and implementing a multiplayer game over a network using Python and UDP sockets required us to deeply understand the mechanisms of message exchange, connection coordination, and event-driven programming. Working with asynchronous communication helped us appreciate the challenges of maintaining consistency and turn-based logic between two remote players, especially when each interaction is reliant on the correct ordering and processing of messages.

One of the key lessons learned was the importance of managing state transitions robustly. We had to account for various edge cases such as duplicate shots, invalid inputs, player disconnections, and handling concurrent actions, all without compromising the game's flow or correctness. The modular breakdown of the client-side logic (board setup, shot validation, ship placement) gave us insight into how separation of concerns can make debugging and iteration easier. Similarly, the server's role as a non-authoritative but fully synchronized coordinator highlighted how centralization can simplify control without introducing performance bottlenecks in a low-scale game.

From a distributed systems perspective, we gained a clearer understanding of communication protocols, packet loss considerations, and error recovery strategies. Implementing UDP helped us recognize its trade-offs in reliability and speed, and forced us to manually handle validation and message ordering. This in turn deepened our appreciation for higher-level abstractions offered by tools like TCP or WebSockets. Additionally, the challenge of designing the game flow (e.g., allowing immediate re-shoot after a hit, preventing duplicate coordinates, or terminating the session after a win/surrender) served as a practical exercise in state machine design.

Overall, this project not only strengthened our technical skills in socket programming and real-time logic, but also taught us collaborative software design, iterative debugging, and the realities of network-based application development in distributed environments