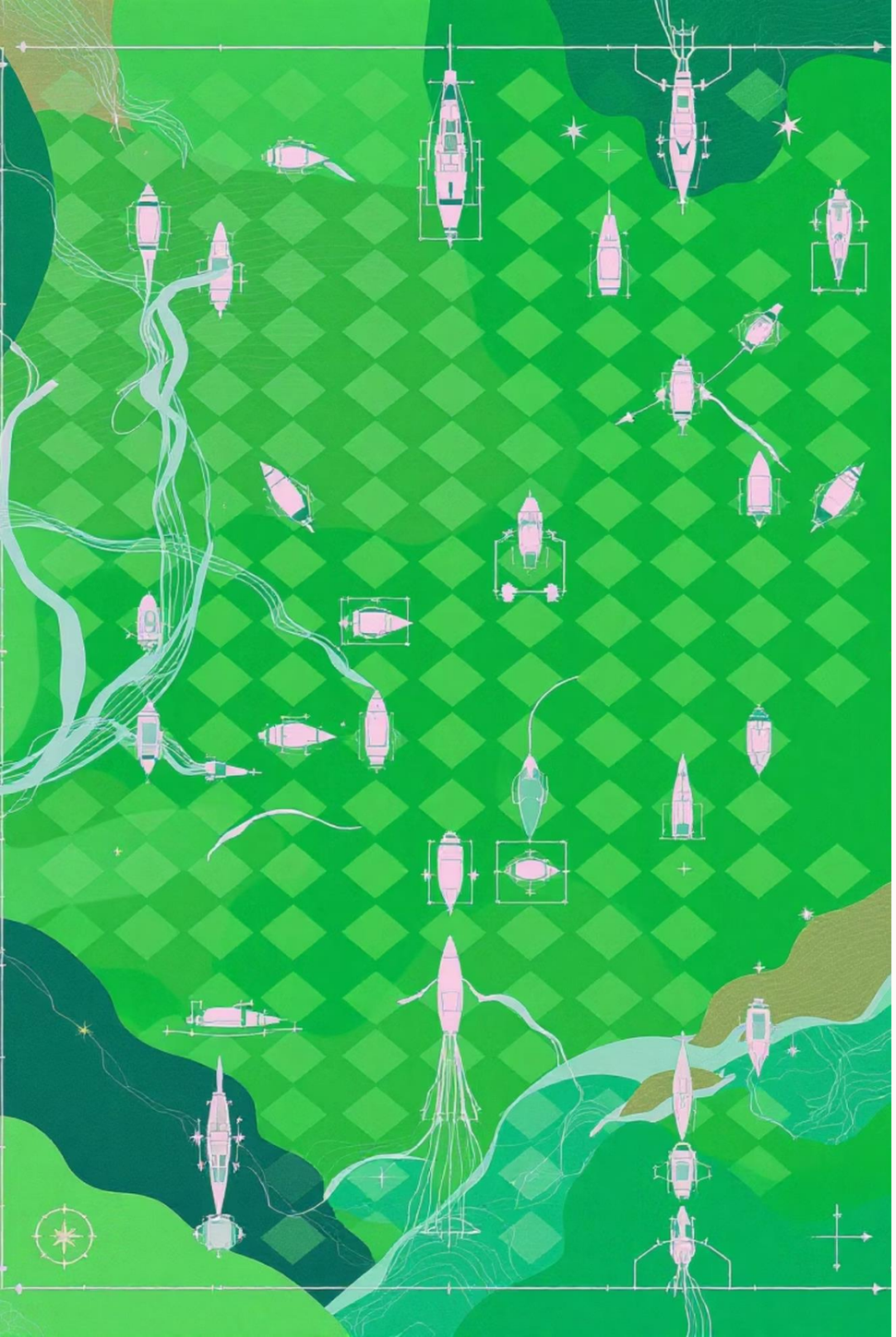


Battleship Game

A turn-based network game built in Python using UDP sockets

Real-time multiplayer game

Synchronization, validation, error handling

Client-server architecture using UDP

2 contributeurs

Project summary

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 between players, handling 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. The system has been designed for simplicity, clarity, and robustness.

Project Objectives & Requirements

Core Objectives

Implement a multiplayer Battleship game using UDP, manage turn-based firing with hit/miss verification, and design modular client-server code for maintainability.

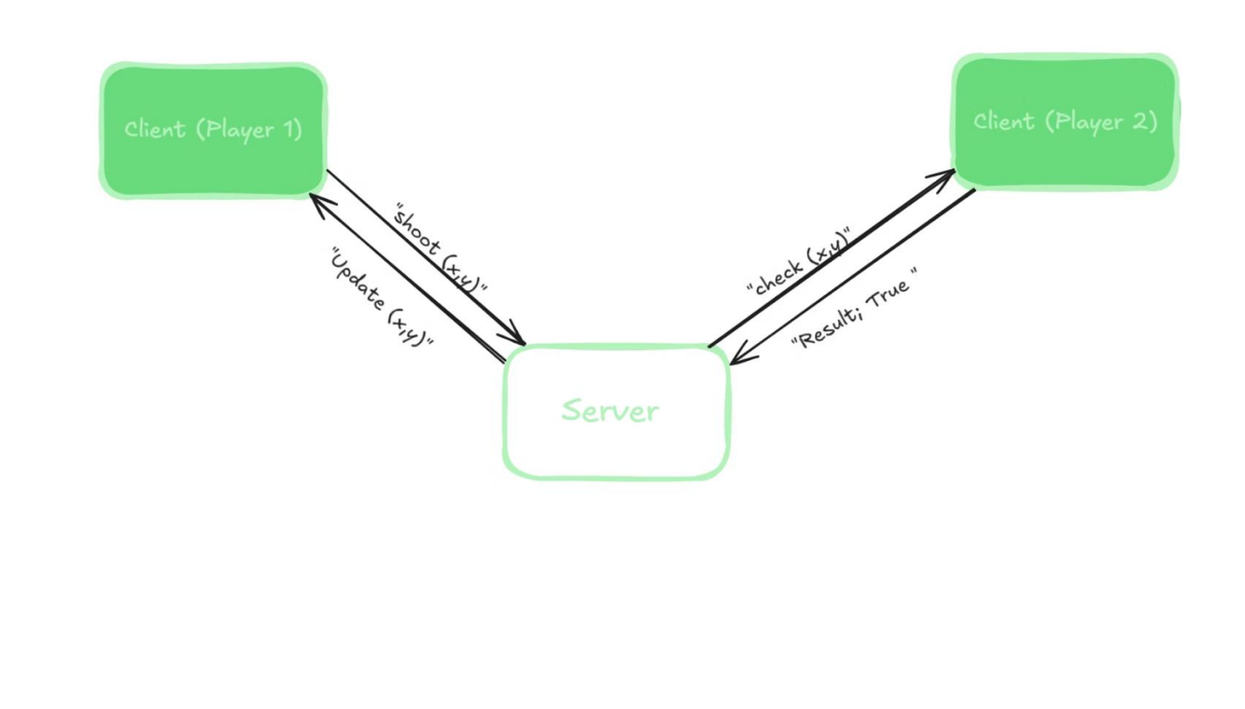
Functional Requirements

Allow two players to connect via UDP, place ships manually or using predefined layouts, alternate turns automatically, validate hits/misses, and end the game when one player either loses all ships, exceeds the time limit, or surrenders.

Non-Functional Requirements

Ensure low-latency communication, robust error handling, modular code structure, user-friendly console UI, and support for multiple concurrent game rooms.

System Architecture – Client-Server Model with UDP



- Follows a client-server model with 2 clients and 1 server acting as the game coordinator.
- The server handles:
  - Room creation and player assignment
  - Turn coordination and message relaying
  - Game termination detection
- Clients interact only through the server (no peer-to-peer communication).
- Communication uses UDP to reduce latency.
- The server maintains game state consistency and validation

Modular Code Structure

Client

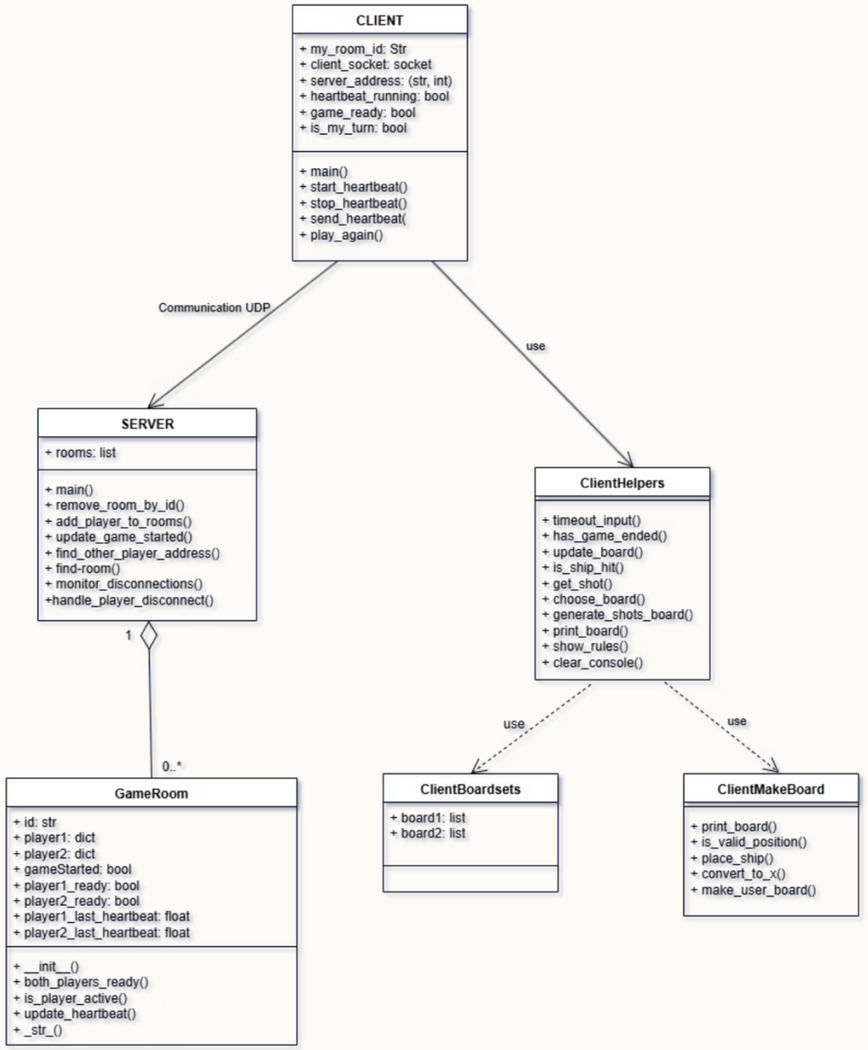
- **client.py**: wish manage main game logic
- **client_helpers.py**: User input, display, validation
- **client_make_board.py**, **client_boardsets.py**

Server

- **server.py**: coordinates communication handling messages and rooms
- **GameRoom.py**: game room model (ID, players, heartbeat)

The system follows a modular design where each component has a clearly defined role.On the **client side**, `client.py` manages the main game flow, `client_helpers.py` handles input, display, and validation, while `client_make_board.py` and `client_boardsets.py` are responsible for creating and managing the game board and ship layouts.On the **server side**, `server.py` coordinates communication and room management, and `GameRoom.py` encapsulates game session data, including player tracking and heartbeat monitoring.This modularity improves clarity, debugging, and long-term maintainability.

Class Diagram



Class Diagram

Client

- **Responsibilities**:
  - Connect to the server using UDP.
  - Manage game flow: setup, turn-taking, game end, replay.
  - Send/receive game data like shots and results.
  - Monitor timeouts and opponent disconnections.
  - Launch heartbeat for disconnection detection.

SERVER

- **Responsibilities**:
  - Accept new player connections via UDP.
  - Manage matchmaking by creating or filling rooms.
  - Relay messages between players (e.g., moves, results).
  - Monitor player heartbeats for disconnections.
  - Control game flow (start, turns, end).

GameRoom

- **Responsibilities**:
  - Store and manage state for a 1v1 match.
  - Track readiness and connection status of each player.
  - Identify disconnections based on heartbeat timeout.

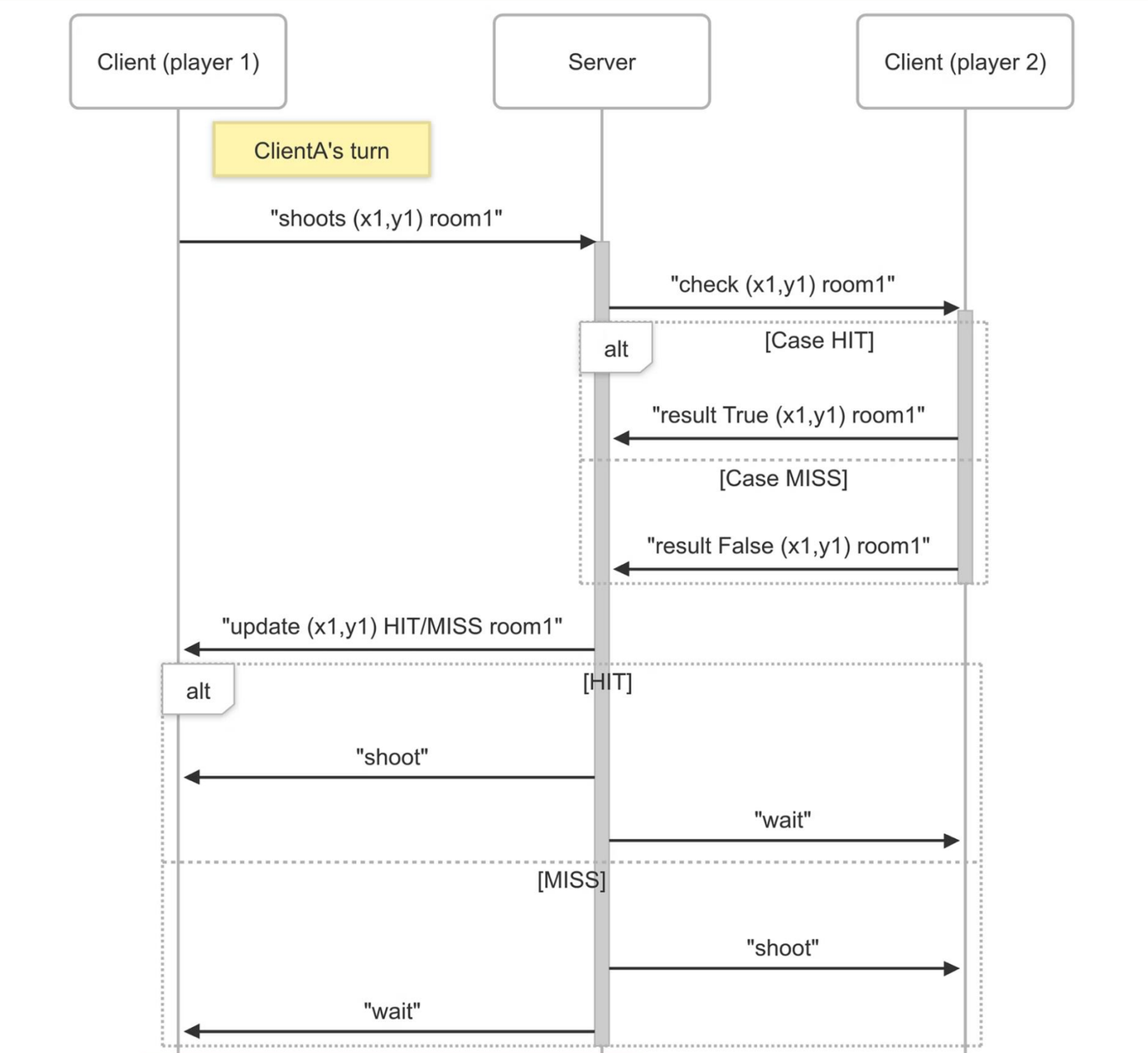
ClientHelpers

- **Responsibilities**:
  - Handle timed input from the user.
  - Update and print the game board.
  - Detect if the game is over (all ships hit).
  - Prompt for shooting coordinates and validate input.
  - Present game rules and handle display cleanup.

ClientBoardsets

- **Responsibilities**:
  - Offer ready-made board configurations to reduce setup time.
  - Support player choice during game preparation.

Sequence Diagram



- **Client (Player 1)**: Initiates the game by taking a turn and shooting at coordinates `(x1, y1)` in `room1`.
- **Server**: Receives the shot, checks if it's a **HIT** or **MISS**, and sends the result (`True` for hit, `False` for miss) back to Player 1.
- **Also server**: Updates the board with the result (HIT/MISS) and either continues shooting (if it's their turn) or waits for Player 2's turn.
- **Alternate Outcomes**:
  - **Player Surrenders**: Triggers an "end" message, declaring the winner and terminating the game.
  - **All Ships Sunk**: Results in an "end_you_won" message for the winning player, followed by a "winner" announcement and game termination.
- **Client (Player 2)**: Acts as the opponent, waiting for their turn or responding to game-ending conditions.

Implementation Details

- **Network Layer (UDP Protocol)**
  - UDP is used for low-latency, connectionless communication.
  - Messages are formatted as simple strings: `shoots`, `check`, `result`, etc
- **Turn-Based Control (Server-side Enforcement)**
  - The server strictly controls player turns by sending `shoot` and `wait`.
  - Only one player is allowed to act at any given time, avoiding race conditions.
- **Game State Management**
  - The server manages game rooms, player readiness, and game progression.
  - Clients maintain their own boards and update them only upon receiving server messages (e.g., `update`).
  - Victory is detected locally when all enemy ships are hit.
- **Timeout & Fault Tolerance**
  - Players have 30 seconds to make a move; otherwise, they lose automatically.
  - Clients send regular heartbeat signals every 10 seconds.
  - The server monitors these signals and removes inactive players after 30 seconds of silence.
- **Error & Disconnection Handling**
  - Invalid messages are ignored to prevent crashes.
  - If a player quits or disconnects, the opponent is notified and the game ends gracefully.
  - Game sessions are cleaned up automatically to free resources.

Validation & Testing

Manual Gameplay

Complete games played from board creation to win detection

Edge Case Handling

Handled invalid inputs, out-of-bounds shots, repeated coordinates, surrender

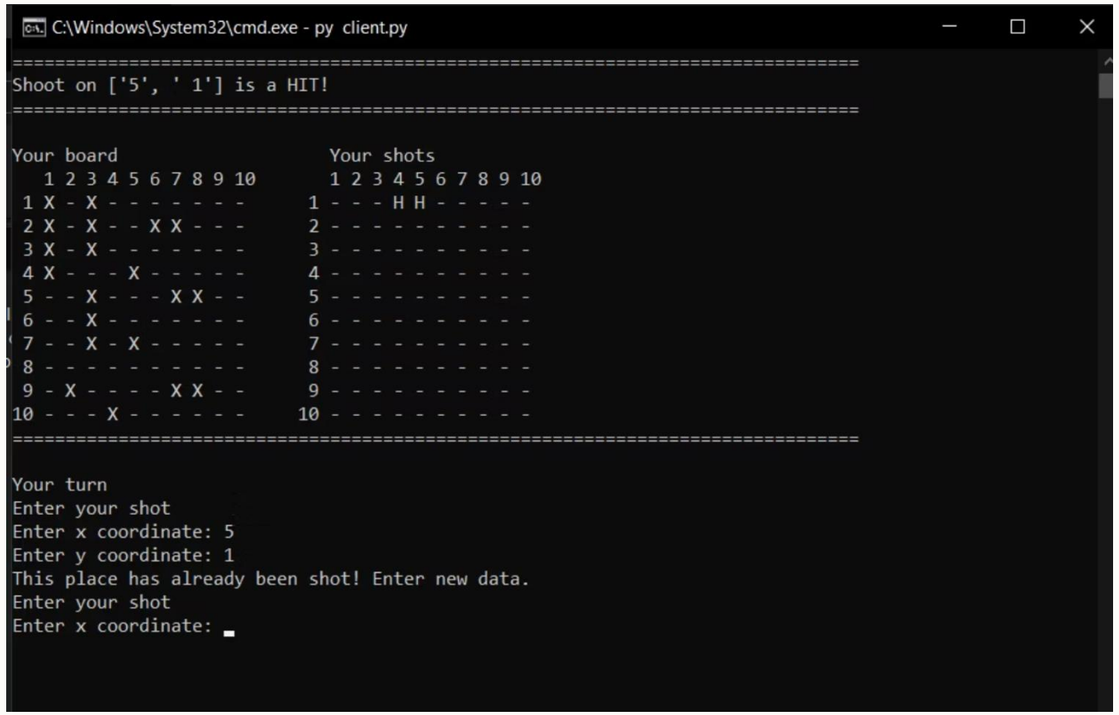
Interface Testing

Checked board rendering, error messages, and turn-based prompts

Communication

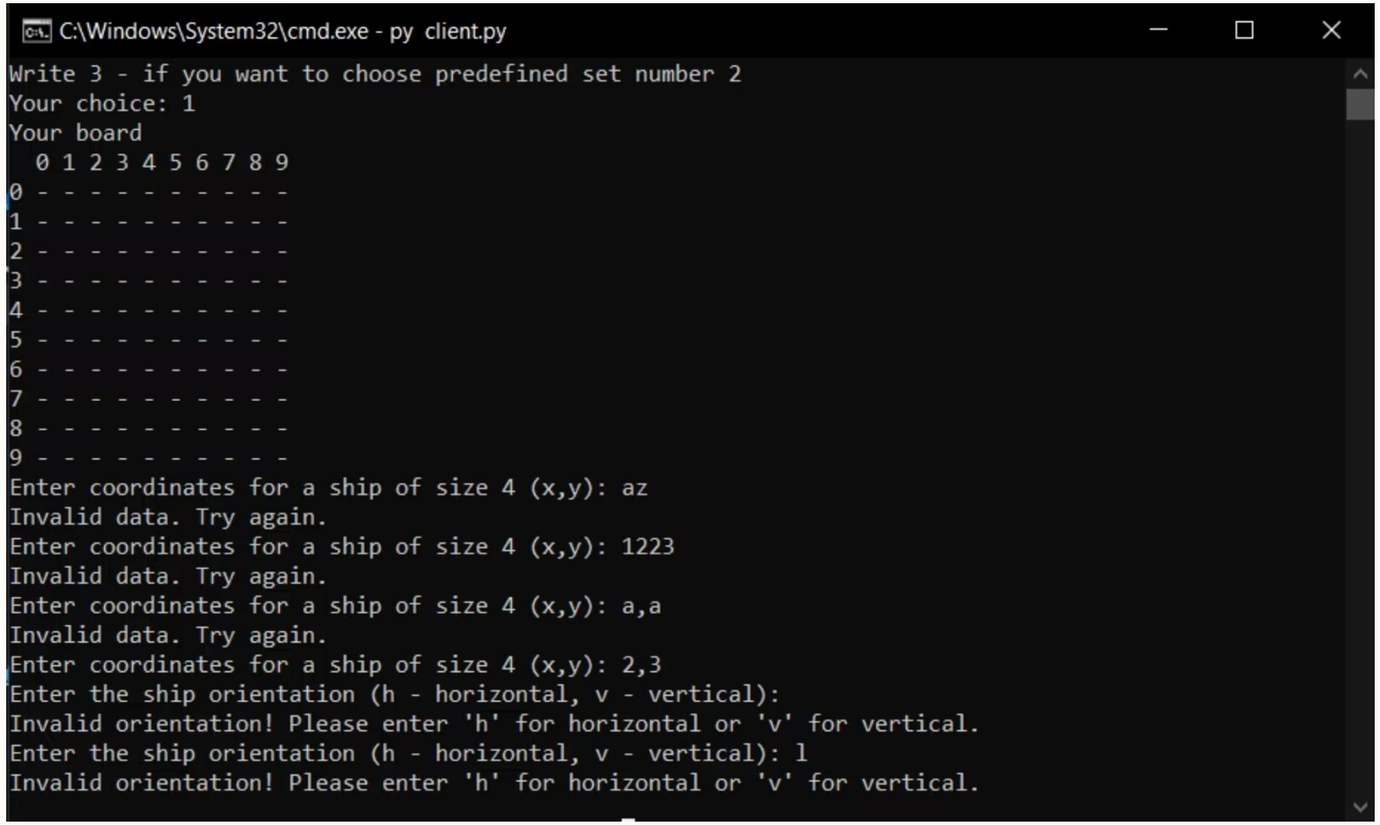
Verified UDP message exchange and server logging under real conditions

Validation & Testing



When a player attempts to shoot a location that has already been targeted, the system displays an appropriate message.

Validation & Testing



When invalid ship placement input is provided during custom board setup, the system displays appropriate error messages to guide the user:

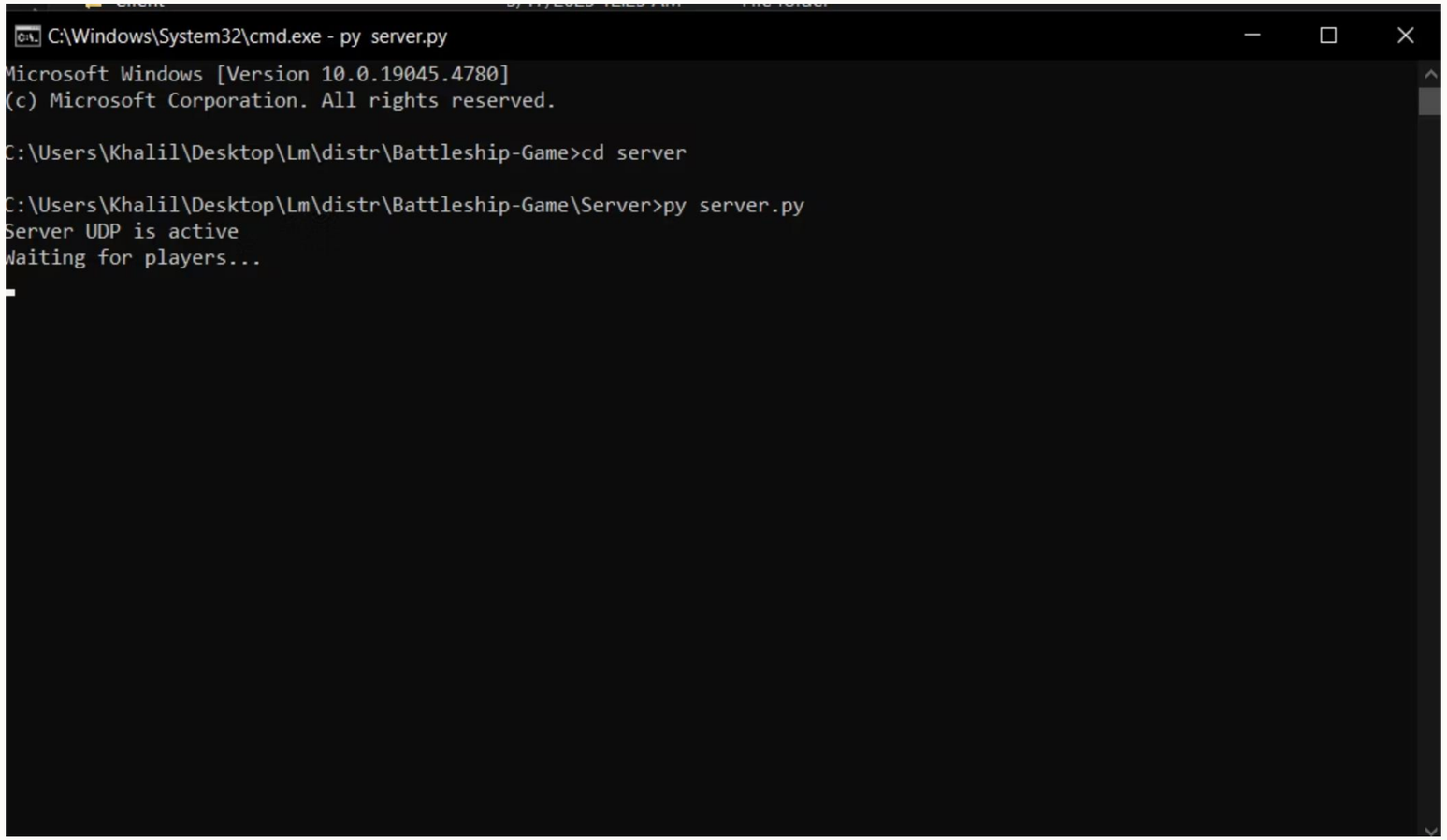
- Rejects non-numeric coordinates (e.g., `"az"`, `"a,a"`).
- Blocks coordinates outside the valid range (must be **1–10**).
- Ensures ship orientation is strictly `"h"` (horizontal) or `"v"` (vertical).

Usage Example

The Battleship game implementation showcases both client and server functionality in action. These screenshots demonstrate key interactions during actual gameplay.

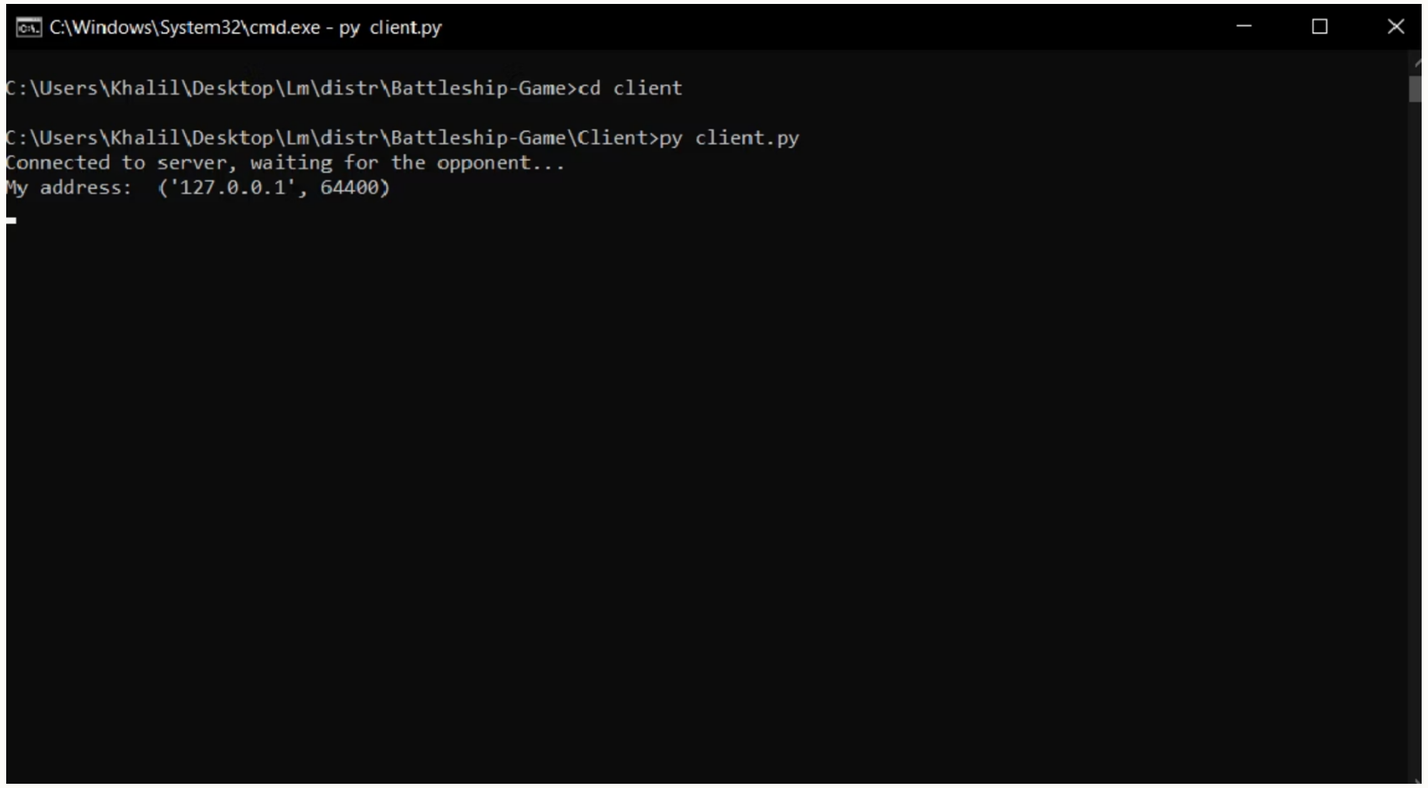
The terminal-based interface provides a clean, responsive experience while maintaining the UDP communication architecture described earlier.

Server Initialization



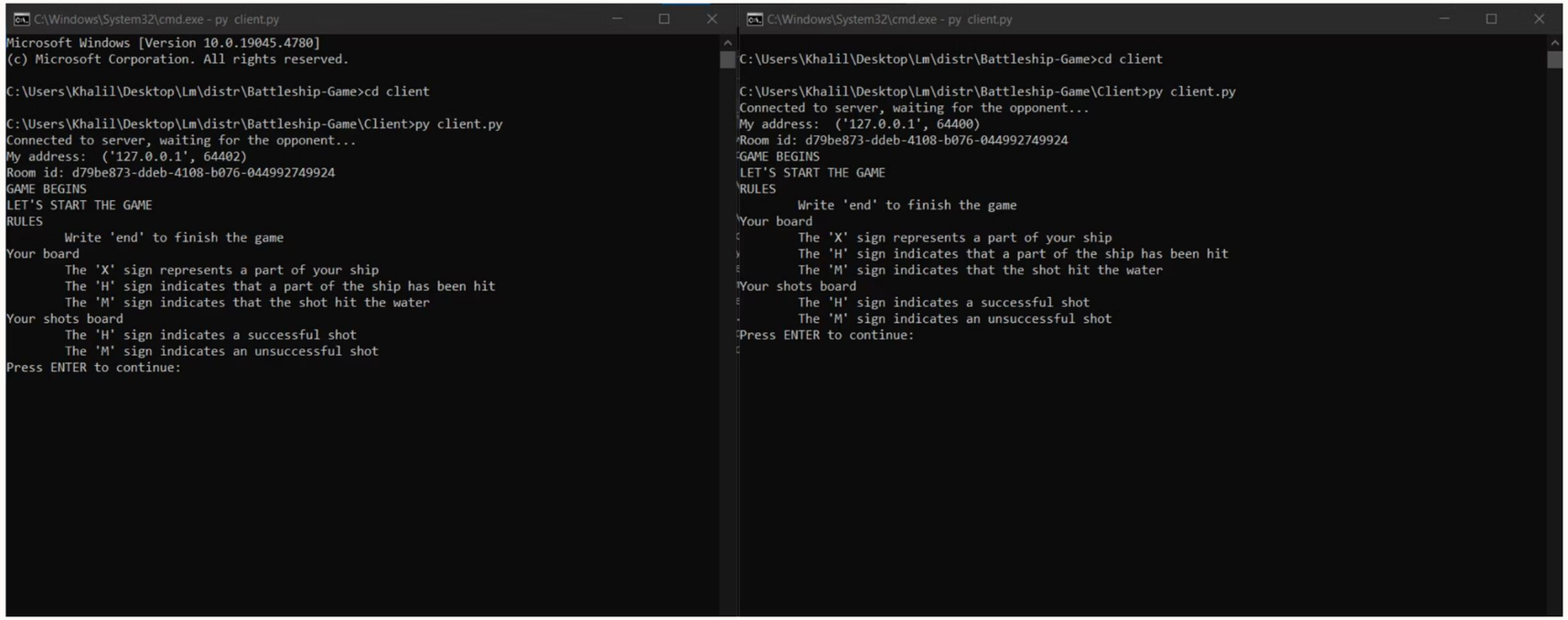
- Launching the game server using UDP protocol
- Server displays active status message
- Ready to accept player connections

Player Joining the Game



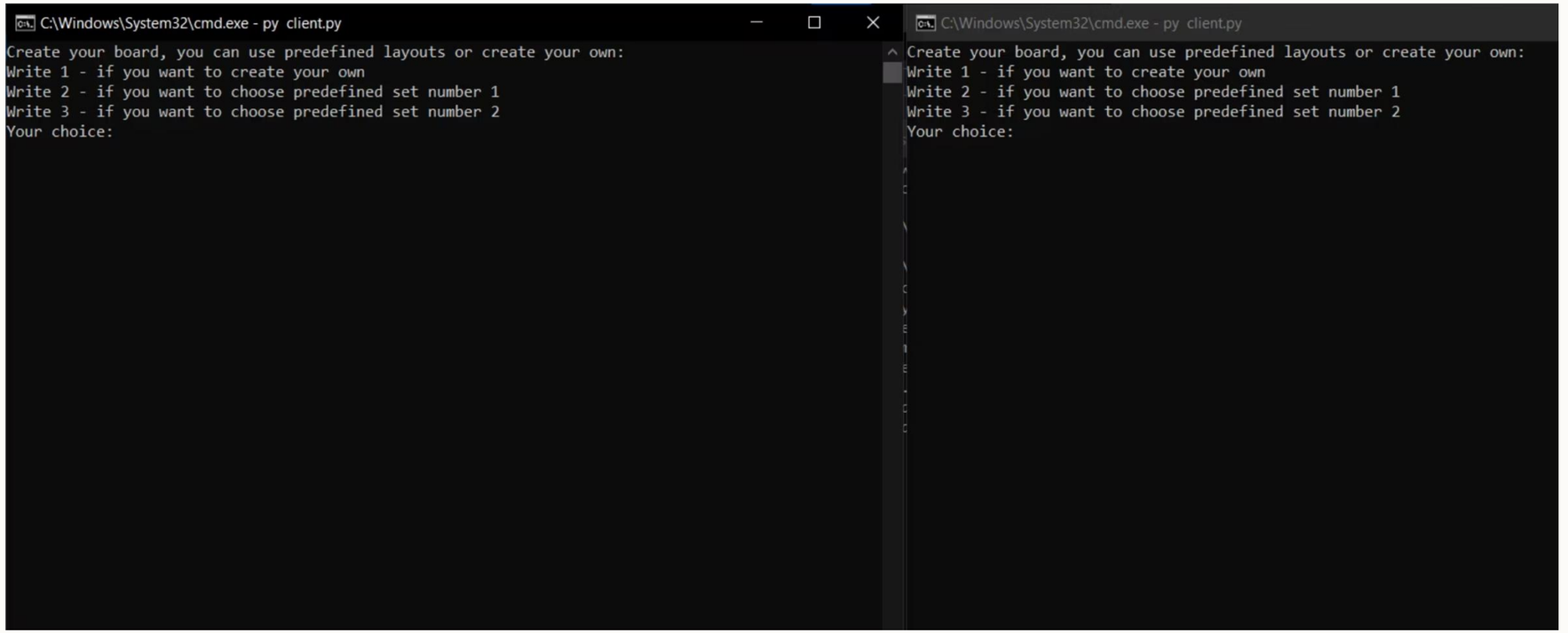
- Client successfully connects to server
- Displays local IP address and port
- Waiting state for opponent connection

Battle Instructions



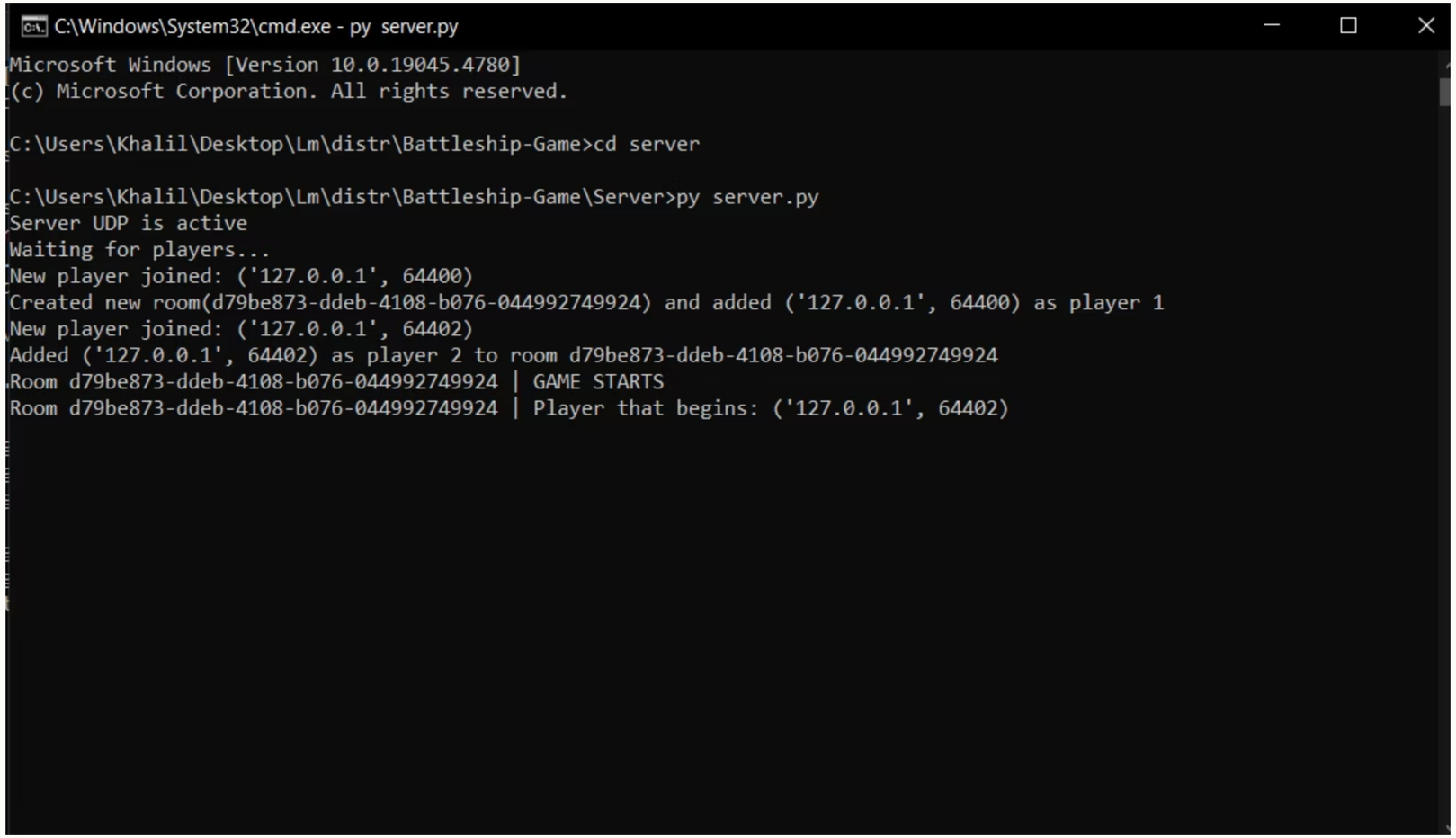
- Key game symbols explained:• `X` = Ship position• `H` = Successful hit• `M` = Missed shot
- End game command shown (`'end'`)
- Interactive prompt to begin gameplay

Board Setup Options



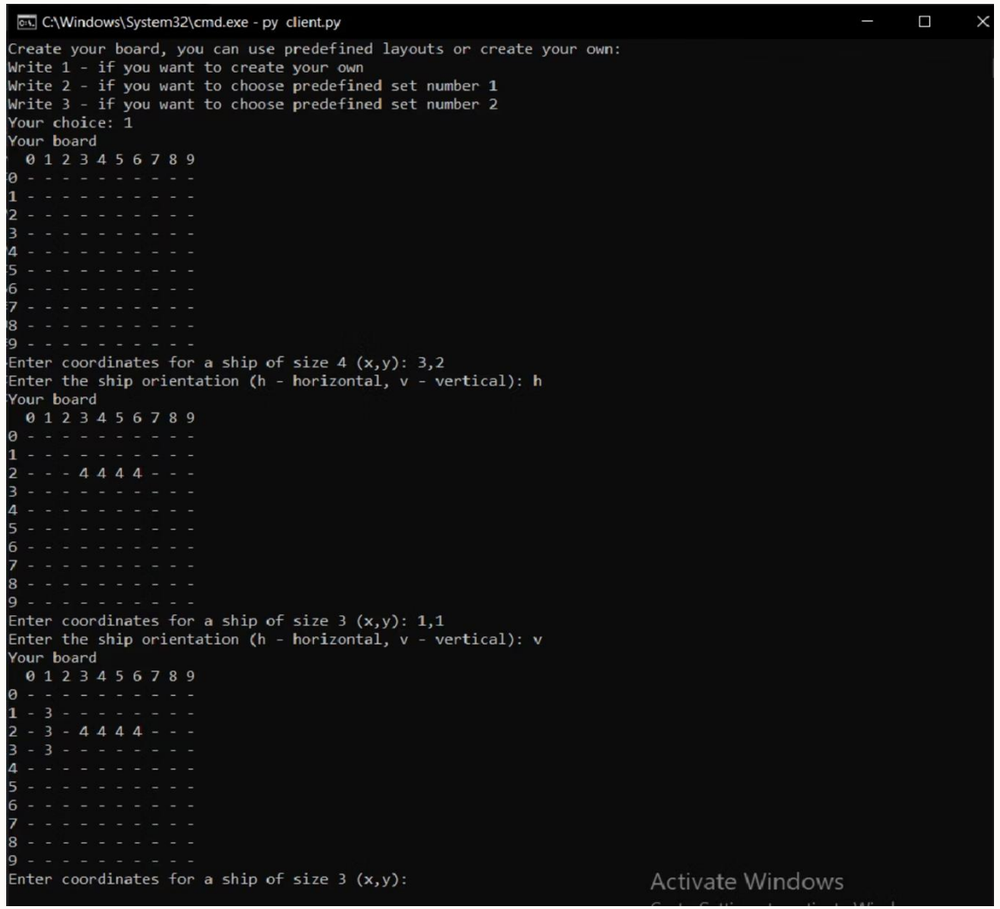
- Players choose their battle configuration:`1` - Create custom board (manual ship placement)`2&3` - Use predefined board layouts

Multiplayer Session Initialization



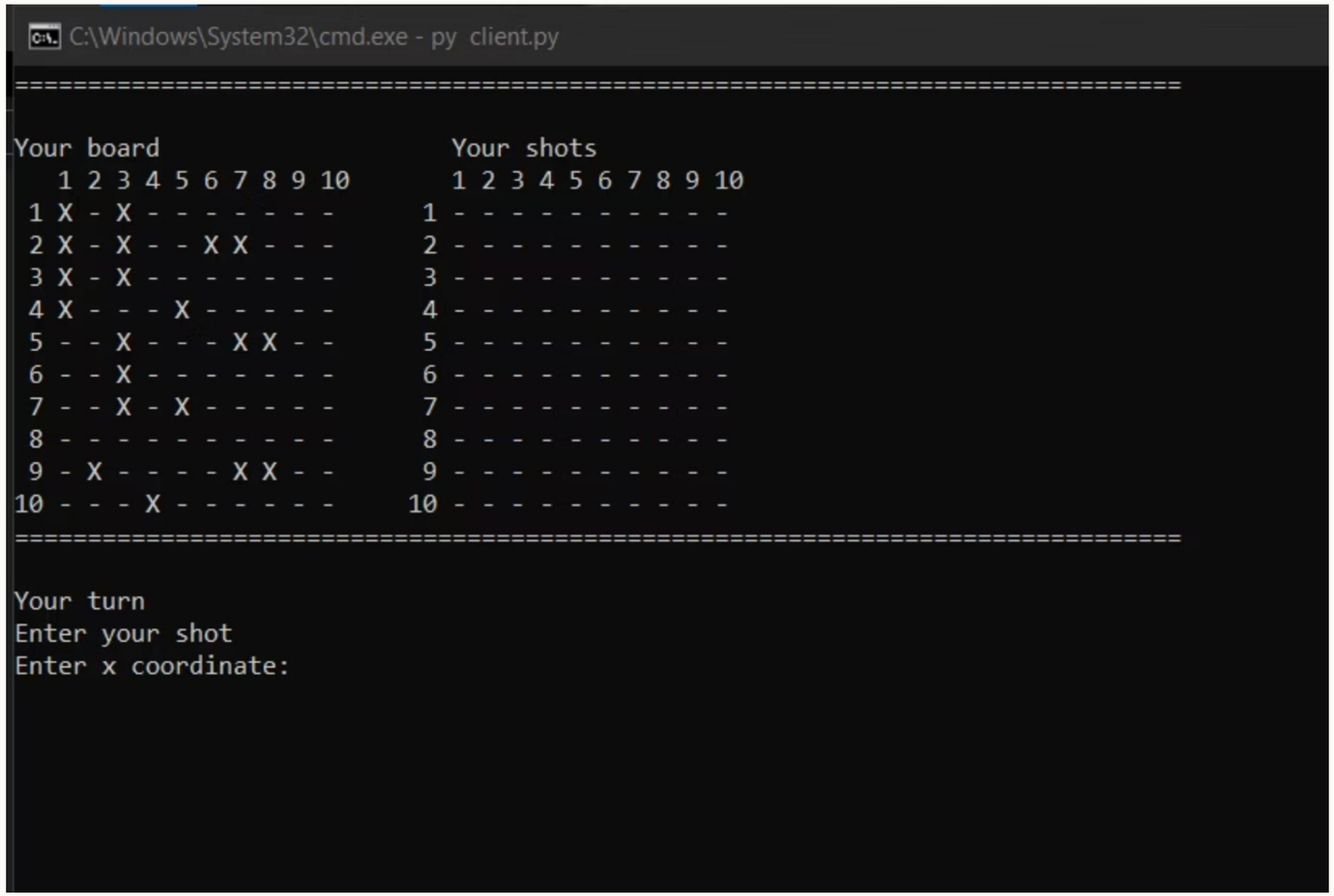
- Server automatically creates game room when 2 players connect
- Unique room ID generated for each session
- Assigns player numbers (1 & 2)
- Randomly selects starting player

Custom Board Creation



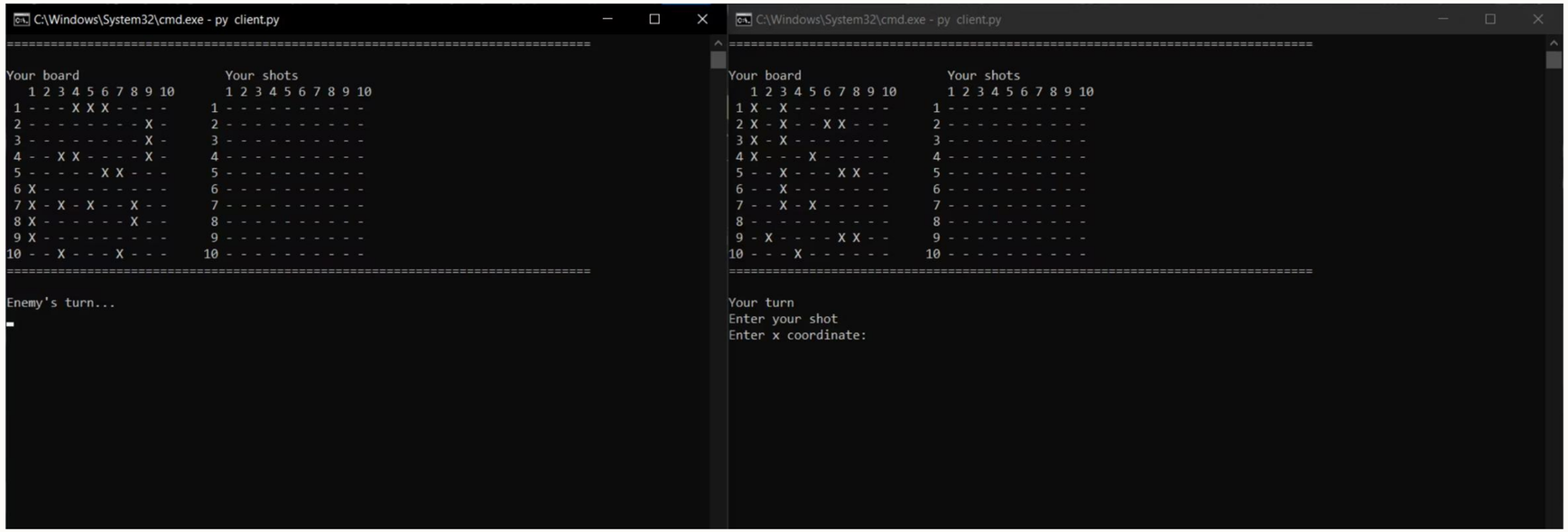
- Step-by-step ship placement:
  1. Enter coordinates (x,y)
  2. Choose orientation (H/V)
  3. Progress through ship sizes (4 → 3 → etc.)
- Visual feedback after each placement

Instant Battle Setup



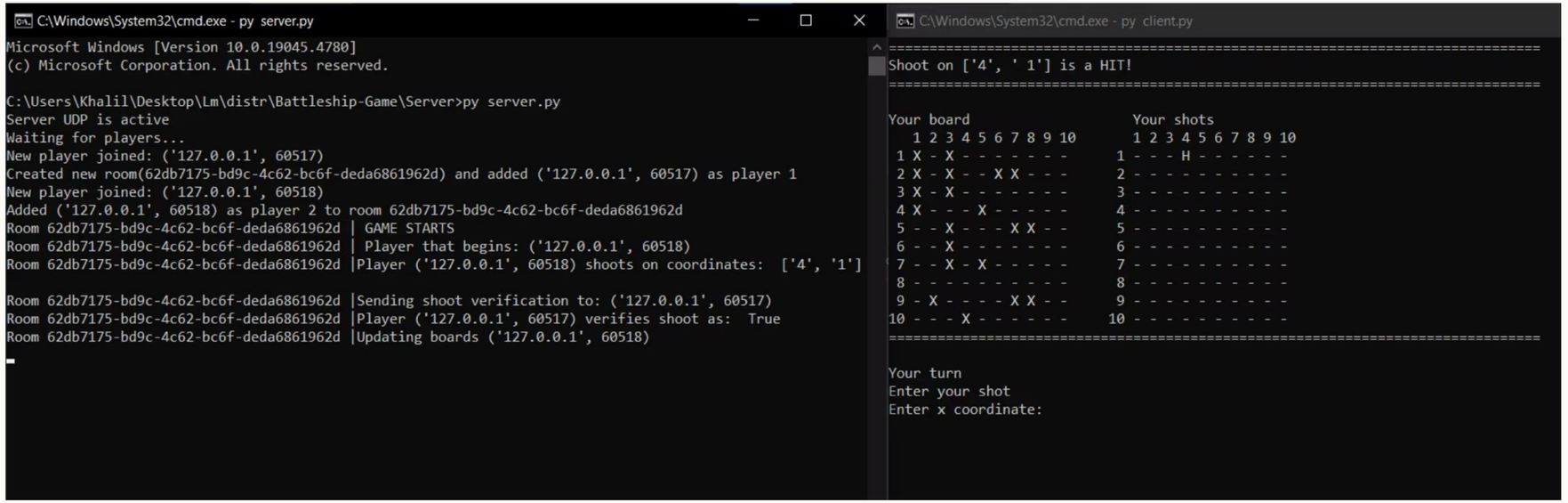
- Players selecting options 2 or 3 get:• Ready-to-play naval formations• Automatically generated ship positions• Immediate transition to gameplay

Player Boards & Turn-Based Gameplay



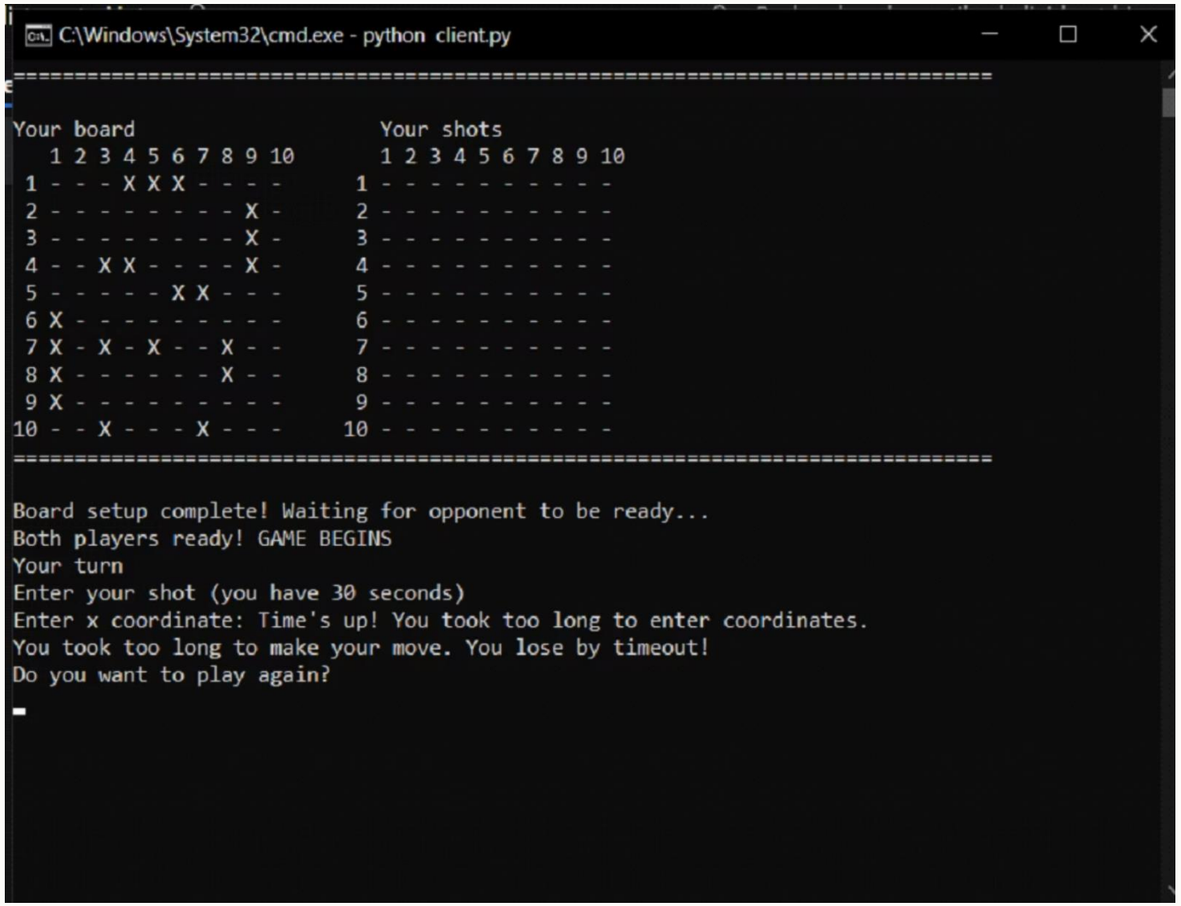
- Each player has their own board to place ships (`X`).
- Players take turns attacking coordinates.
- Hits are marked as `H`, misses as `M`.
- Turn continues until a miss occurs.
- First to sink all enemy ships wins

Turn-Based Combat & Hit Feedback



- Player hits opponent at `['4', '1']` (marked as `H`).
- Turn continues until a miss occurs.
- Server updates both players' boards in real time.
- Left grid: Player's ships (`X` = intact, `H` = hit).
- Right grid: Attack history (`H` = hit).
- Goal: Sink all enemy ships first to win.

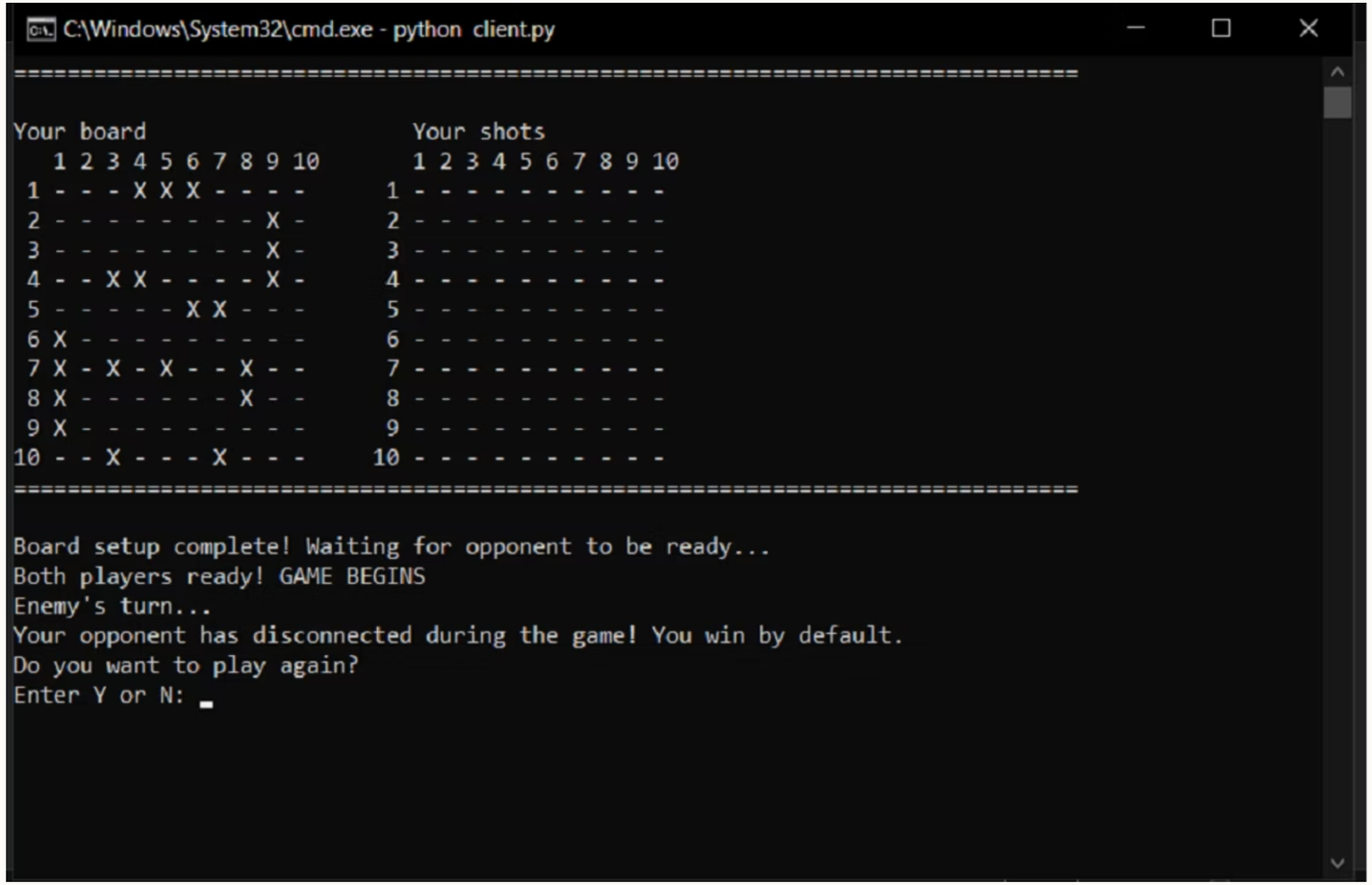
Timeout Penalty - Automatic Loss



- Player failed to input move within 30 seconds
- System message: "You took too long! You lose by timeout"
- Game offers replay option (Y/N)

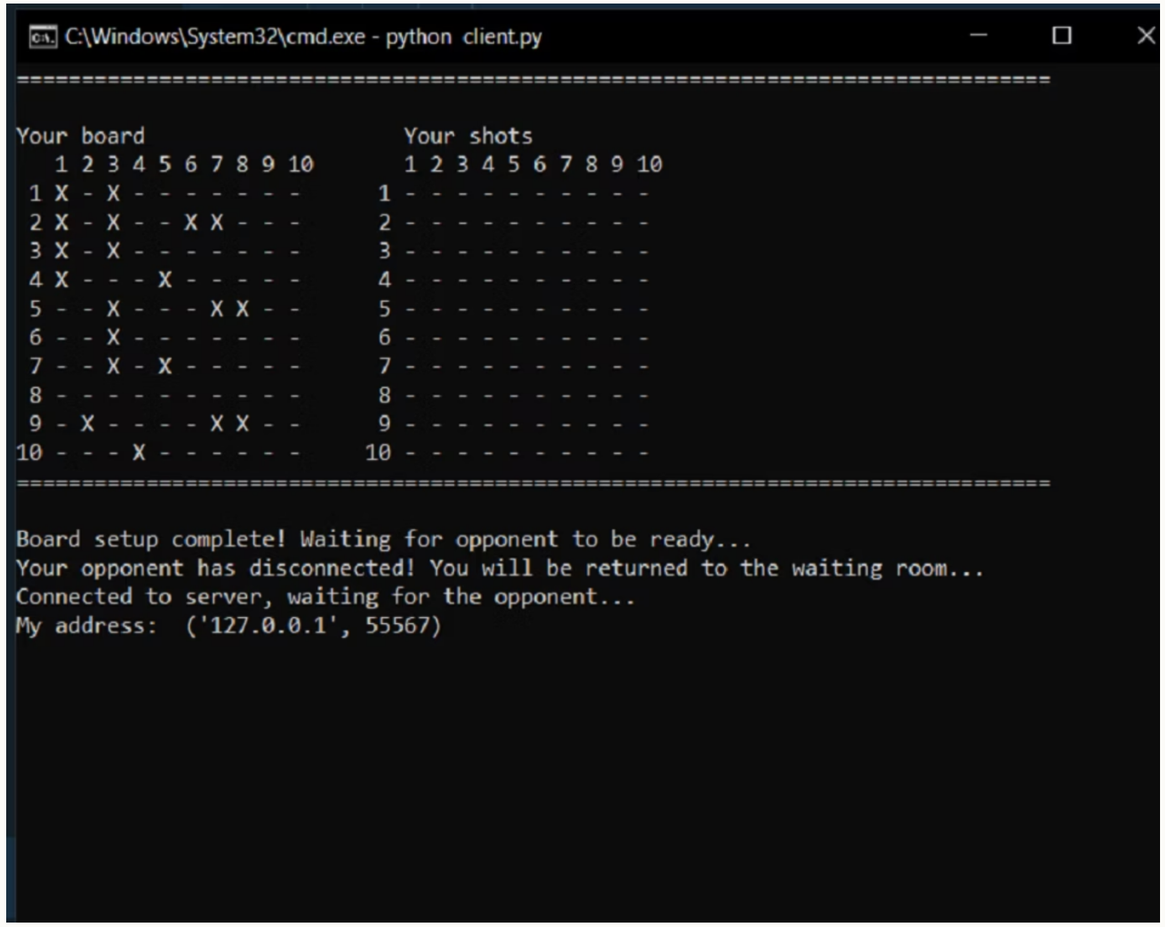
- Opponent failed to respond in time
- Victory message: "Opponent lost by time. You win!"
- Shows player's intact board and unused shots grid
- Prompts for rematch (Y/N)

Mid-Game Disconnection - Automatic Win



- Opponent disconnects during active gameplay
- Clear victory message: "Your opponent has disconnected! You win by default."
- Shows player's current board state and shot history
- Prompts for rematch (Y/N)
- Server detects disconnect within 5-15 seconds via heartbeat

Pre-Game Disconnection - Waiting Reset



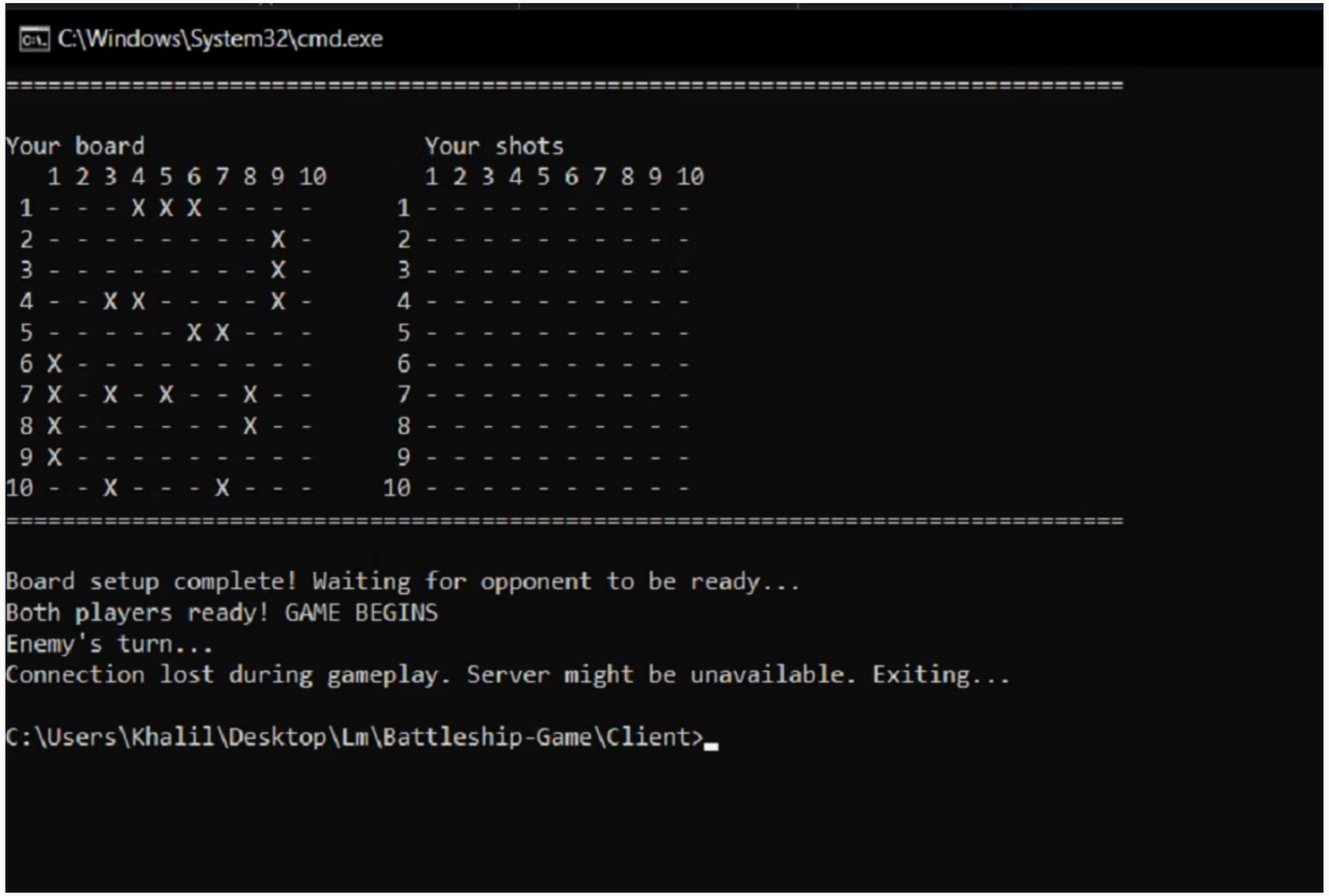
- Opponent disconnects during board setup phase
- Non-punitive message: "Opponent disconnected! Returning to waiting room"
- Preserves player's configured board
- Maintains connection to server for new opponent
- Shows seamless reconnection process

Server-Side Disconnection Handling

```
Created new room(b9d74fc9-f75f-4d5a-bbe3-fd50687a2d17) and added ('127.0.0.1', 63352) as player 1
New player joined: ('127.0.0.1', 55449)
Added ('127.0.0.1', 55449) as player 2 to room b9d74fc9-f75f-4d5a-bbe3-fd50687a2d17
Room b9d74fc9-f75f-4d5a-bbe3-fd50687a2d17 | Both players connected, preparing game
Room b9d74fc9-f75f-4d5a-bbe3-fd50687a2d17 | Player 2 is ready
Room b9d74fc9-f75f-4d5a-bbe3-fd50687a2d17 | Player 1 is ready
Room b9d74fc9-f75f-4d5a-bbe3-fd50687a2d17 | GAME STARTS
Room b9d74fc9-f75f-4d5a-bbe3-fd50687a2d17 | Player that begins: ('127.0.0.1', 55449)
Room b9d74fc9-f75f-4d5a-bbe3-fd50687a2d17 | Player 2 disconnected
Game in room $b9d74fc9-f75f-4d5a-bbe3-fd50687a2d17 ended
Room b9d74fc9-f75f-4d5a-bbe3-fd50687a2d17 removed due to player ('127.0.0.1', 55449) disconnection
```

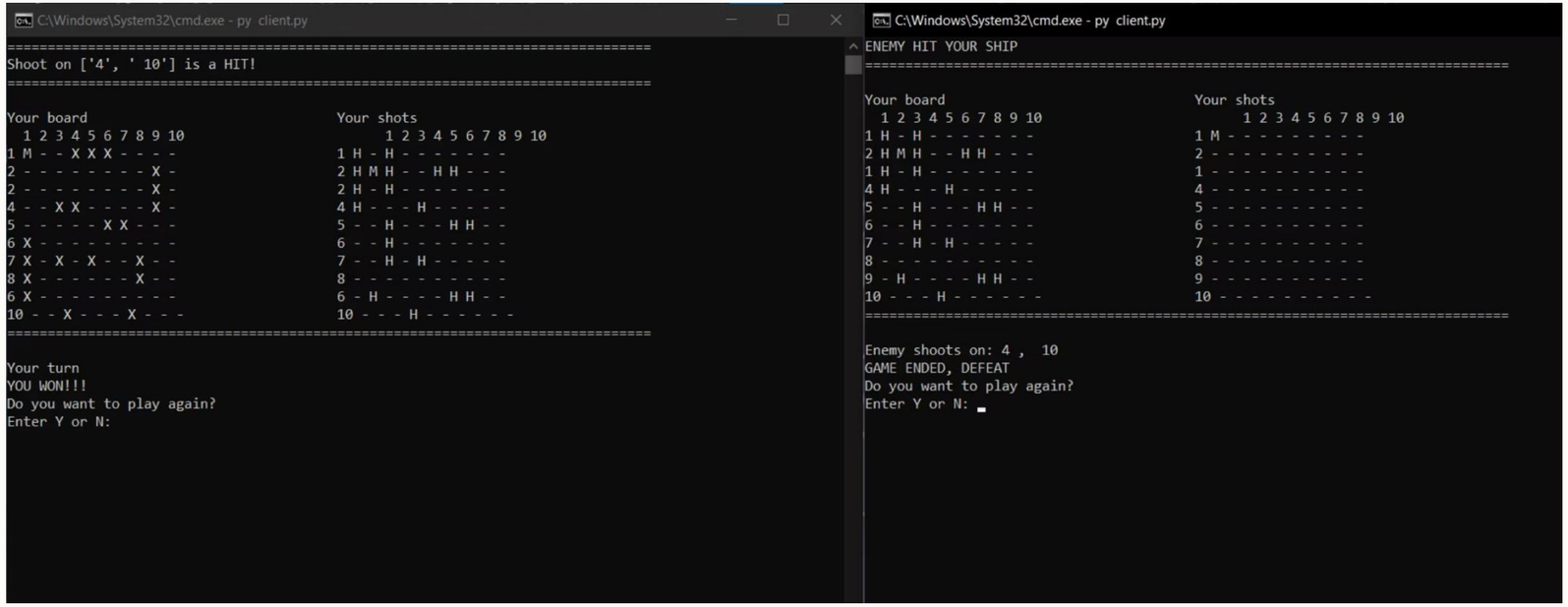
- Detects and processes disconnection in real-time
- Automatically cleans up game room

Server Shutdown - Connection Lost



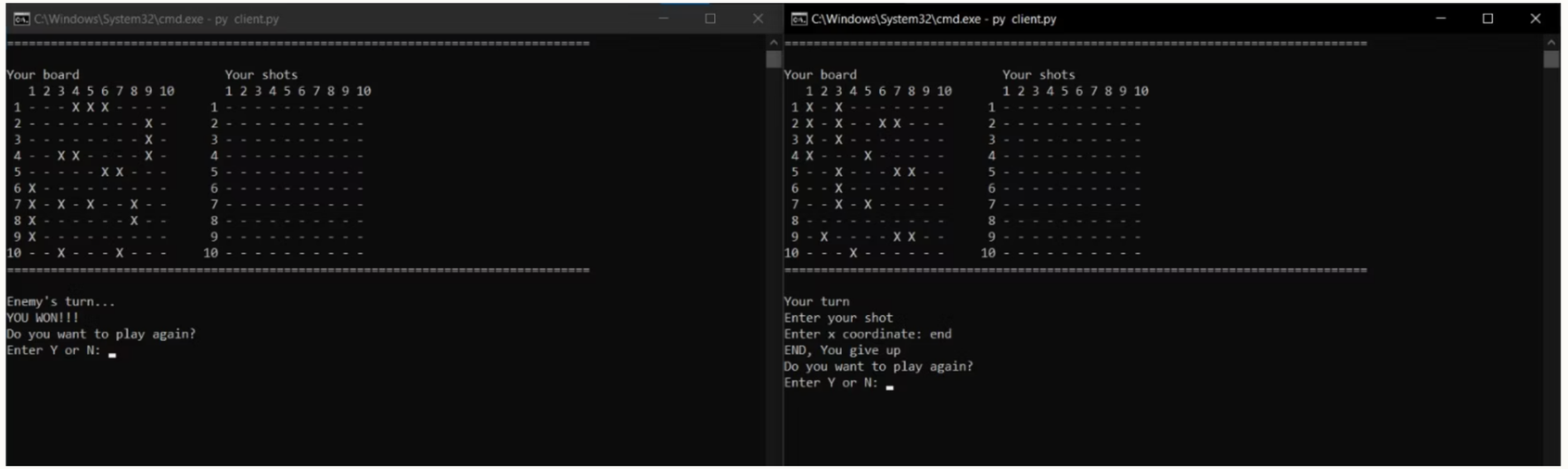
- **Server Crash/Shutdown:** Shows player's perspective when server becomes unavailable mid-game.
- **Clean Exit:** Displays warning: "*Connection lost. Server might be unavailable. Exiting...*"
- **Automatic Closure:** Client safely terminates after detecting server disconnect.

Victory Screen & Game Conclusion



- Winning player sees: **"YOU WON!!"**
- Losing player sees: **"GAME ENDED DEFEAT"**
- Prompts players to replay: **"Enter Y/N to play again"**
- Cleanly ends or restarts the game based on choice

Surrender & Game End



- Player can surrender by typing "**END**".
- Message displayed: "**You give up!**" (for quitter).
- Opponent sees: "**You win!**" (automatic victory).
- Prompts both players: "**Play again? (Y/N)**"

Conclusion & Future Work

Achievements

Successfully built a stable, responsive multiplayer game using UDP

Educational Value

Practical application of networking concepts and distributed programming

Lessons Learned

Gained experience with distributed systems, real-time communication, and state management

Future Improvements

Graphical interface, persistent storage, multi-room support