UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

REPORT

Subject: Network Progamming
Content: Networked Number Guessing Game

Group members:

Nguyen Dang Nhat Anh	**23BI14028**
Nguyen Duy Son	**23BI14387**
Le Thi Cam Giang	**22BA13108**
Hoang Van Bao	**22BA13044**
Nguyen Dinh Quang	**22BA13263**
Nguyen Thai Tu	**22BA13212**
Bùi Đức Minh	**23BI14305**

I, Description

The **Networked Multiplayer Number Guessing Game** is a Python-based application that allows multiple players to connect to a central server, join or create game rooms, and compete to guess a randomly generated number between 1 and 100. The game enhances player interaction with real-time chat functionality, implements a scoring system based on the number of attempts and speed, and provides a leaderboard to rank players after each game. The server efficiently manages multiple game rooms, each supporting 2 to 4 players, and ensures smooth handling of connections, disconnections, and game state transitions.

II, Requirements & Implementation

1. Server managing multiple game rooms with 2-4 players each

- The server, implemented in server.py, uses the game_room class to represent individual rooms, each with a host and up to 3 guests (total capacity of 4 players).
- Players can create new rooms or join existing ones via the room_list class, which maintains a list of active rooms, and the player_list class, which tracks player states and room memberships.
- Room capacity is enforced by the game_room class's capacity attribute, set to 4.

2. Random number generation (1-100) for each game

- At the start of each game, the server generates a random number using random.randint(1, 100) within the game_room_handle function in server.py.
- This number serves as the target for players to guess during the game session.

3. Turn-based gameplay with timeout mechanism

- Gameplay is managed by the game_progress class, which processes player guesses and provides feedback ("Guess a larger number" or "Guess a smaller number") via socket messages.
- A 10-second countdown timer, implemented in game_room_handle using time.time(), ensures games conclude within a set duration. If time runs out, the game ends, and results are announced.

4. Scoring system based on number of attempts and speed

- Scoring is handled by the player_progress class, which tracks each player's guess count (count) and time taken (time).
- The point_cal method calculates scores as follows:
    - Base score: 100 points.
    - Speed bonus: Up to 100 points, reduced by the time taken (e.g., max(0, int(100 - self.time))).
    - Guess penalty: 5 points deducted per attempt.
    - Final score: max(0, base + speed_bonus - guess_penalty), or 0 if the player didn't guess correctly in time.

5. Game chat functionality

- Players can send chat messages within their room, implemented in server.py under option "2" of the room menu.
- Messages are relayed to all room members except the sender using the socklist method of room_list, ensuring real-time communication.

Message handling:
The server waits for a message from the player for 5 seconds.
If there is a message:

- Retrieve the sender's room ID
- Retrieve the sender's name
- Send the message to all other players in the room

Error handling:

- If no response within 5 seconds
- If the message is empty
- If there is a connection error

6. Leaderboard tracking

- After a game ends, the game_progress class's announce_results method calculates scores, sorts players by score in descending order, and sends a leaderboard to all players via socket messages.
- The leaderboard includes each player's rank, socket address (as an identifier), and score.

7. Reconnection capability if a player disconnects

- The server handles disconnections gracefully in server.py. If a client socket receives no data or encounters a ConnectionResetError, it is removed from sockets_list and clients, ensuring the game state remains consistent.

III, Game Flow

1. Connection & Registration:

- Client (client.py) connects to the server (server.py).
- Server prompts for a username. Client sends username.
- Server registers the player using the player class (classes/player.py) and adds them to player_list (classes/player_list.py).

```
→  net-programming-final-main (2) python3 server.py
[Log] Server listening on 127.0.0.1:12344...
Accepted new connection from ('127.0.0.1', 49384)
Username 'giang' registered for ('127.0.0.1', 49384)
```

- Server sends a welcome message and the main menu (from classes/send_menu.py).

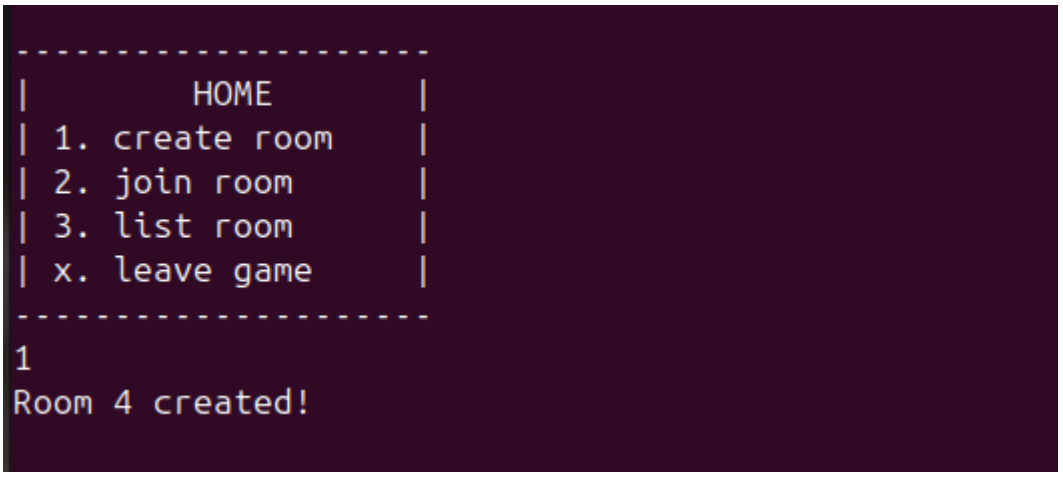
```
→  net-programming-final-main (2) python client.py

Please enter your username:
giang
Welcome giang!
```

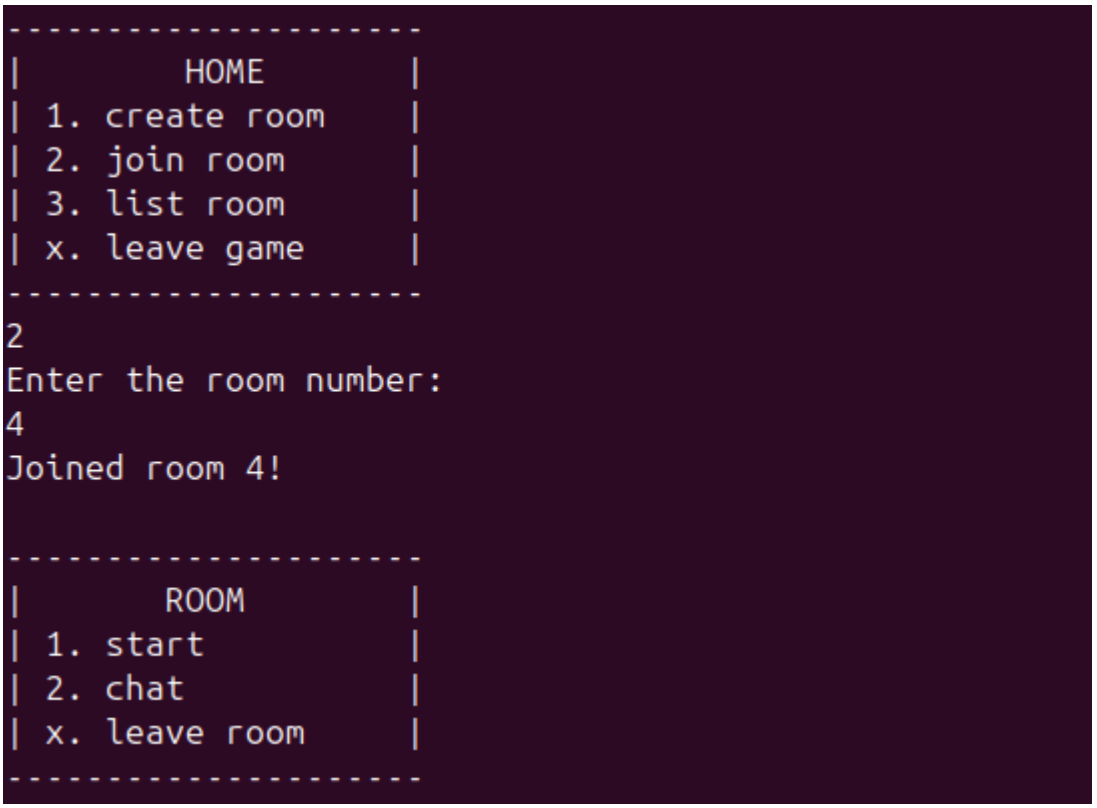
2. Menu:

- Player can create a room, join a room, list rooms, or leave the game:

  1. Create Room: Server creates a game_room with a unique ID (from classes/unique_random.py), adds it to room_list, and sets the player as host. Player enters the Room Menu.



  2. Join Room: Server prompts for Room ID. Player enters ID. If valid and game not in progress, player joins the room and enters Room Menu.



3. List Rooms: Server sends a list of available rooms with their IDs and player counts.

```
---------------------
|        HOME       |
| 1. create room    |
| 2. join room      |
| 3. list room      |
| x. leave game     |
---------------------
3
Available rooms:
Room ID: 3, Number of player: 1
Room ID: 4, Number of player: 2
```

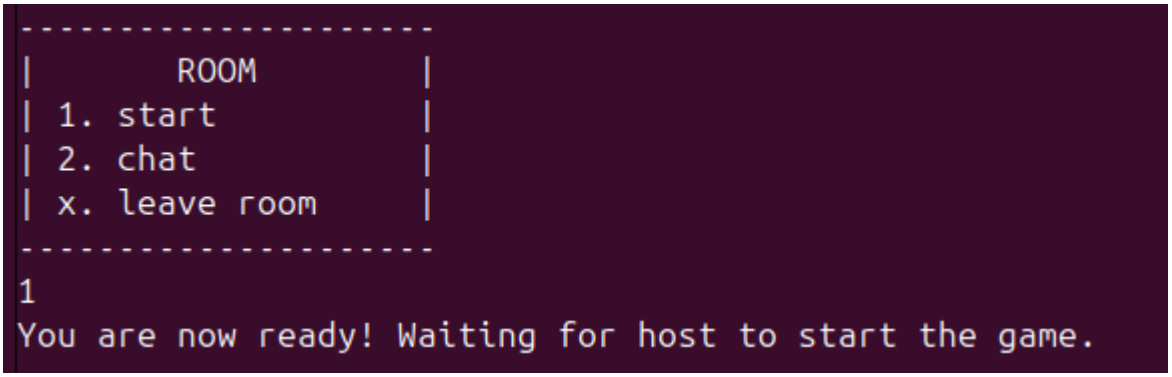
x. Leave Game: Player is prompted for confirmation. If 'y', connection is typically closed by the client or can be gracefully handled by the server.

```
---------------------
|        HOME       |
| 1. create room    |
| 2. join room      |
| 3. list room      |
| x. leave game     |
---------------------
x
Are you sure you want to leave the game? (y/n):
y
Goodbye.
```

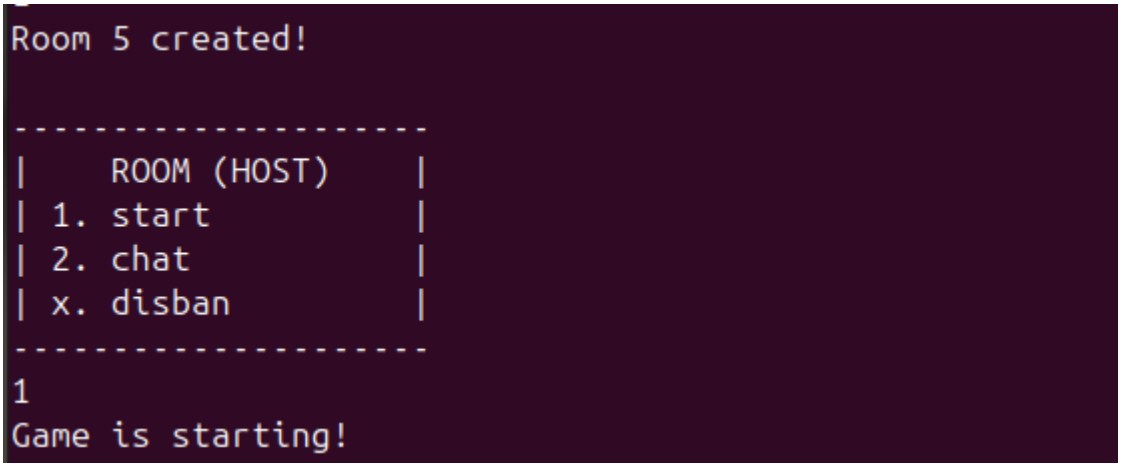
3. Room Menu & Pre-Game:

Players wait for others; the host can start the game when all are ready.

- Host Menu: "1. Start", "2. Chat", "x. Disband".
- Guest Menu: "1. Start", "2. Chat", "x. Leave room".
- Guests select "1" to toggle their ready status (player.in_ready).

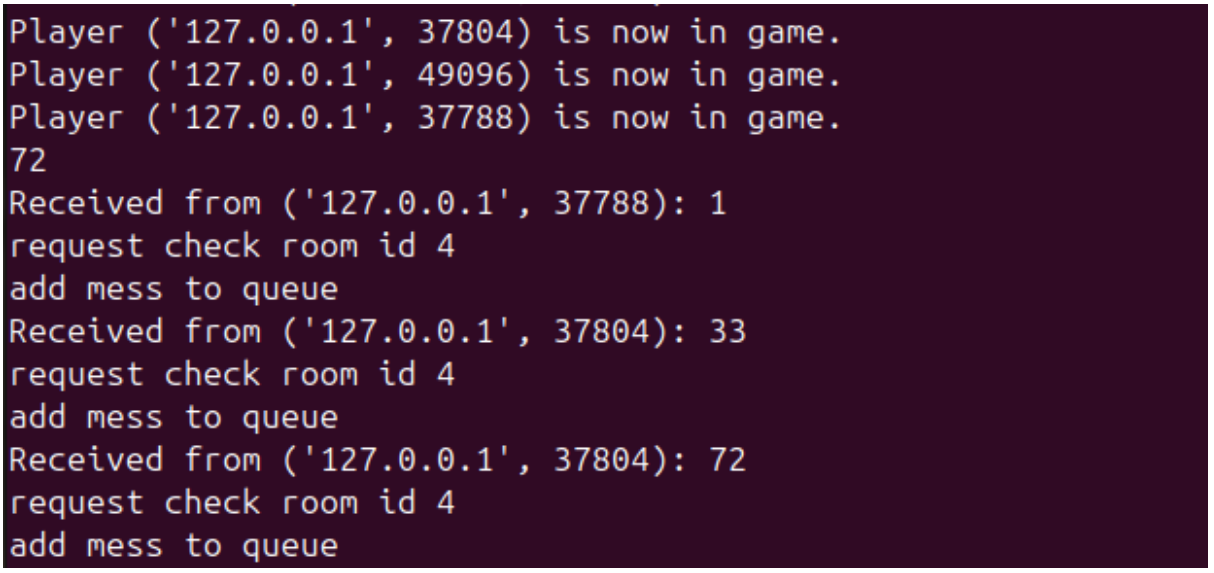


- Host selects "1" to start the game. Server checks if all non-host players are ready (player_list.check_all_ready). If so, the game begins.



4. Game:

- The server generates a random number (1-100).



- Players submit guesses via client.py, processed by game_room_handle in a separate thread. Feedback is provided immediately.

```
55
[Game] Guess a larger number.
Your guess is smaller than the final result.
```

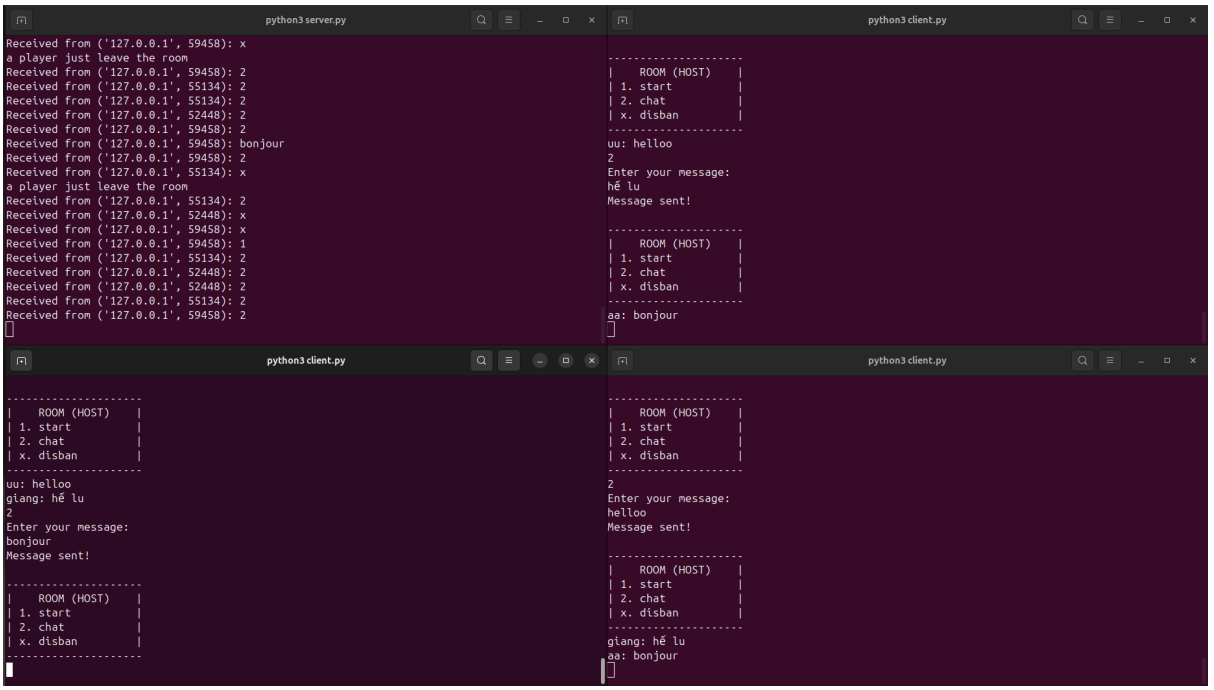
- The game ends when a player guesses correctly or the 30-second timer expires.Scores and leaderboard are shown.

```
59
[Game] Correct!!!!
Your guess is right!!!!
time up !!!
[Game] Game over! Your score: 186. Your rank: 1.

[Leaderboard]
Rank 1: Player ('127.0.0.1', 41718) - Score: 186
Rank 2: Player ('127.0.0.1', 47460) - Score: 0
Rank 3: Player ('127.0.0.1', 55128) - Score: 0
```

5. Chat:

- Players can chat in the room at any time.



IV, Code Structure

- **server.py:** Main server logic, handles connections, menus, game flow, and communication.
- **client.py:** Client-side logic for connecting, sending/receiving messages, and user interaction.
- **classes/player.py:** Player object definition.
- **classes/player_list.py:** Manages all players and their states.
- **classes/room_list.py:** Manages all rooms and room operations.
- **classes/game_progress.py:** Handles scoring and leaderboard.
- **classes/player_progress.py:** Tracks individual player progress in a game.
- **classes/game_message_queue.py:** Queues and manages game-related messages.
- **classes/send_menu.py:** Generates menu strings based on player state.

1.Connection Setup and Basic Configuration

Socket setup:

The game uses TCP/IP sockets on port 12344 in non-blocking mode to handle multiple simultaneous connections. The server socket is configured with SO_REUSEADDR to avoid "address already in use" errors when restarting the server.

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind((HOST, PORT))
server_socket.listen()
server_socket.setblocking(False)  # Non-blocking mode
```

Multiplexing with Select:

The select.select() function acts as the system's core mechanism to monitor multiple sockets concurrently without needing a separate thread per connection. This approach enables the server to:

- Detect new client connections
- Identify sockets ready for reading data
- Handle exceptions when sockets encounter errors

Client Connection Management:

Each client connection is managed through the following steps:

1. Register username upon connection
2. Track client status (lobby, in room, or in game)
3. Route message processing based on user context
4. Handle disconnections proactively or due to errors

Connection Error Handling:

The game handles various connection errors such as:

- Connection reset (ConnectionResetError)
- Timeout while reading data (using select.select() with timeout)
- Resource cleanup when clients disconnect unexpectedly

2. Multithreaded Architecture for Game Room Management

Handling Multiple Game Rooms Concurrently:

- The system utilizes room_list to store and manage individual game rooms separately.
- Each room is assigned a unique ID and maintains a list of player sockets.
- The select.select() function enables the server to listen to multiple client connections simultaneously without blocking.
- A message queue system is implemented to manage messages independently for each game room.

Using One Thread per Game Room:

```
threading.Thread(target=game_room_handle,args=(room_id msg_queue,room_list socklist(room_id))) start()
```

- The game_room_handle function runs an infinite loop to continuously process the game logic within each room.
- Running this function in the main thread would block it, hence it is executed in a separate thread.
- Each room maintains its own countdown timer and processes messages independently.
- The game logic (such as number guessing) must be handled concurrently across multiple rooms.

Multithreading:

- The main thread remains responsive, handling new connections and menu navigation.
- Each game thread processes its own game logic independently, avoiding interference with other threads.
- A dedicated thread, finished_game, handles the cleanup and processing of completed games.
- This architecture prevents blocking I/O operations when waiting for player input, ensuring smooth performance.

Event Handling:

- select.select() monitors sockets for available data to read.
- The event_finished_game event signals when a game has ended.
- Different handling logic is applied depending on the player's state (in-game or in the lobby).
- The message queue system (msg_queue) distributes and routes messages appropriately between rooms

Timeout Mechanism Ensures:

- The game terminates after a predetermined time limit
- The server is not blocked while waiting for client responses
- Resources are released promptly and efficiently
- Players are notified if they fail to respond

V, Conclusion

The project successfully implements a networked multiplayer number guessing game with all required features, including room management, random number generation, turn-based gameplay with a timer, scoring, chat, and leaderboard tracking. The codebase is modular and organized, supporting a robust and interactive multiplayer experience.