Assignment
Multi-threaded Rock-Paper-Scissors Server-Client Application

Objective

Create a multi-threaded server-client application where multiple clients can play a rock-paper-scissors game against the server. The server maintains a high score board, which clients can update if they win. The update process to the high score board must be thread safe.

Requirements

1. Server Requirements

- The server should listen for incoming client connections.
- The server should handle client connections using threads.
- The server should play rock-paper-scissors with the clients.
- The server should maintain a high score board that clients can update if they win.
- The update to the high score board should be thread-safe using mutexes.

2. Client Requirements

- Each client connects to the server and plays a rock-paper-scissors game.
- If the client wins, they can play again in the same session, so high scores can be achieved
- The game choice (rock, paper, or scissors) for both the client and the server should be random.

3. High score Board

- The high score board should store the names of clients, who have won against the server, and the high score as well. (How many times has the client won against the server during one game session.)
- Only one client should be able to update the high score board at a time.
- Maintain only the top 10 scores.

Detailed Specifications

1. Server Implementation

- Use `std::thread` for handling multiple client connections.
- Use `std::mutex` to protect updates to the high score board.
- Implement a mechanism for playing rock-paper-scissors and determining the winner.

- Maintain a list of winners in a thread-safe manner.
- Server should be able to handle a high number of clients (>100).

2. Client Implementation

- Each client should connect to the server, play the game, and if they win, update their name on the high score board.

3. Game Logic

- Rock beats scissors, scissors beat paper, and paper beats rock.
- Both the client and the server should randomly choose rock, paper, or scissors.

4. General

- Use OOP for Server and Client
- Use Clean Code
- Log the relevant information as the program run
- Create a desired number of arbitrary demo sets
- There is no user interaction allowed after starting the game
- The solution contains one executable
- If there is any question we are glad to answer them

5. Bonus

- Implement the solution's build process using CMake

Evaluation Criteria

- **Correctness**: Does the application meet the specified requirements?
- **Concurrency**: Is the multi-threading implemented correctly and efficiently?
- **Code Quality**: Is the code well-structured, commented, and easy to understand?
- **Robustness**: Can the server handle multiple clients without crashing?
- **Functionality**: Does the game work as expected, and does the high score board update correctly?