Report

Ruiyu Peng,423510135,rpen966

I. Introduction

This assignment involves designing and implementing a multi-threaded game server in C#, specifically for two-player games. The server facilitates player matchmaking and coordination of game actions, without specific game rules.

The server is developed using C# programming language, starting from scratch with a synchronous server socket, without utilizing higher-level APIs.

Communication between the server and game clients is based on the HTTP REST approach. The following (GET) endpoints are implemented:

- /register: Generates a random username for the player, registers it, and returns the registered name to the user. The generated username is used for player identification in subsequent interactions.
- /trygame: Checks if the current user's username has been registered, preventing access to the game if not.
- /pairme?player={username}: Attempts to match the player in the queue and creates a game record to store information. The server returns the game information, including a game record, and introduces the endpoint /gamestate for polling the current game state.
- /mymove?player={username}&id={gameId}&move={move}: Allows the player to submit their move to the server during the "in progress" phase of the game. The server updates the corresponding game record with the provided move.
- /theirmove?player={username}&id={gameId}: Retrieves the move of the other player from the server during the "in progress" phase of the game. The server provides the last move recorded in the game record associated with the given game ID.
- /quit?player={username}&id={gameId}: Informs the server about the player's intention to quit the game. The server removes the game record associated with the given game ID. Subsequent attempts by the player to access the game record (e.g., retrieving moves) will fail.

In conclusion, this report presents the design and implementation of a multi-threaded game server in C#, specifically tailored for two-player games. The server utilizes HTTP REST for communication and provides endpoints for player registration, matchmaking, move submission and retrieval, and game quitting. The detailed implementation principles can be found in the "Implementation" section, while the "Evaluation" section discusses how to test the server and highlights its limitations.

II. Problem

During this project, I confronted several challenges and found corresponding solutions as follows:

A. Thread Handling

According to the design requirements, each player's request should be processed in an independent thread. This is to trace thread and port information in the logs. To fulfil this requirement, I utilized the Task.Run() method, which automatically creates a new thread to handle player requests. The system processes different requests based on the commands received.

B. Backend Connection Establishment

As TCPListener was not an option, I had to manually parse HTTP requests and construct HTTP responses. Having no previous experience in this area, I relied on ChatGPT for learning and code building. During browser testing, I encountered cross-origin problems, indicating that I did not have a correct HTTP response header. To tackle this problem, I designed a MakeJsonResponse function.

C. Player Matching and Movement Rights Allocation

During the player matching phase, a problem occurred: even when two different usernames were provided, the system treated them as two separate games instead of matching them as opponents. To solve this, I changed the game matching logic and introduced a game state concept. Based on the game state, I could determine whether the game could begin, satisfying the player matching requirement. Simultaneously, I added an authorization field in the game for managing the movement rights, initially set to player1, and swapped after every move.

D. Synchronization of Two Game Windows

In the initial design, the player entering later would immediately see the "match successful" prompt, but the earlier player would not. To solve this, I designed a polling mechanism for game state. When the game state is found to be "in progress" in the polling, it stops polling and enters the game.

E. Chess Piece Movement Interaction Processing

I implemented the chessboard using an HTML table and used images to represent chess pieces. The initial challenge was how to convert image movement into position information for transmission. By consulting ChatGPT and learning how to handle this issue, I assigned a unique id to each chessboard cell and chess piece, storing these ids for recording and transmitting movement information.

F. Handling Send Request without Movement

If a player clicked 'send' without moving, it would lead to subsequent 'get move' operations being unable to receive information, thus disrupting the alternation of rights. To tackle this problem, I added a judgement on the frontend: if no move is made, a message would be prompted, and no further request would be sent.

G. Telnet Testing

The command line in telnet differs from the browser, which has a connect:keep-alive tag, keeping the backend server and the browser continuously connected. During telnet testing, I faced the challenge that telnet would disconnect after the server processed a command, preventing continuous command processing. To resolve this, I made extensive changes to the code. I created a variable called alive and set it to true, using it to keep telnet connected with the server.

This summary represents the primary issues I encountered during this project and the strategies I employed to overcome them.

III. Implementation

A. async Task Main & HandleRequest

The Main function is the entry point of the application, creating a Socket object and binding it to an available IP address and port 8000. It starts listening for connection requests using the Listen method. In an infinite loop, it accepts client connections using the Accept method and sends a confirmation message to the client. Every request is handled in a new task asynchronously using Task.Run, ensuring concurrent processing of multiple client connections to enhance server responsiveness.

B. MakeJsonResponse

MakeJsonResponse is a function that generates a JSON response. It accepts JSON content as input and returns a complete response string that includes the response headers and JSON content. This function serves to generate a correctly formatted JSON response with the necessary response headers, ensuring accurate transmission and parsing of JSON data during network communication.

C. HandleRegisterRequest

In the initialization part, a name pool is designed. When a user requests registration, it is redirected to the HandleRegisterRequest function. In the handling process, the function checks whether the available username list is empty. If available usernames exist, the function randomly selects one and removes it, then adds this username to the set of registered users, and finally generates a successful response containing the username and sends it back to the client.

D. HandleTryGameRequest

This function first checks whether the username has been registered. If the username is unregistered, the function generates a JSON response with an error message. If the username is registered, the function generates a JSON response with a success message, indicating that the user is registered and can start the game.

E. HandlePairMeRequest

This function checks whether there is a game waiting for players to join. If there is, it further checks whether the request is made by the same player. If it's the same player, the function generates a JSON response indicating that the player is already in the queue. If it's a different player, the function sets the Player2 property of the current game object and updates the game status to "in progress," indicating the game has begun.

F. HandleGameStateRequest

This function is used to handle requests for game state queries. If a game is in progress, the function further checks whether the requested username is one of the players in the game. If it is, the function generates a JSON response containing the current game information and sends it to the client. If it isn't, the function generates a JSON response indicating that the user is not in the game.

G. HandleMyMoveRequest

This function handles requests from players to submit their moves. It first decodes the encoded move string into a list of Move objects. Then, it checks whether a game is currently in progress and if the game state is "in progress." If these conditions are met, the function further checks whether the current user is the player for the current round.

H. HandleTheirMoveRequest

This function handles requests from players to retrieve the moves of the other player. It first checks whether a game is in progress and if the game state is "in progress." If these conditions are met, the function further checks whether the requested username is one of the players in the game and whether it's their turn.

I. HandleQuitRequest

This function handles requests from users to quit the game. It first checks whether a game is in progress and whether the requested username is one of the players in the game. If these conditions are met, it sets the game state to "ended" and generates a JSON response indicating the game has ended, returning it to the client.

IV. Evaluation

A. Browser test

1) To test the application on the browser, you need to open two different browsers such as Google Chrome and Microsoft Edge. This is important to avoid thread merging issues when playing the game within the same browser.
2) Before starting the game, make sure you have completed the registration process. Without registration and player matching, you won't be able to enter the game or interact with the game board. Enter your username in the input box and click "Trygame" to verify the player. Once the verification is successful, you will see other action buttons.
3) Once inside the game, you will see four buttons: "pairme", "sendmove", "gettheirmove", and "quitgame". You can also interact with the game board. However, if there is no matched player, clicking the "sendmove" and "gettheirmove" buttons will display error messages. The main error messages displayed are "No move made" and "Player has not made a move yet". Nevertheless, these messages do not affect the progress of the game.
4) To begin the game, click the "pairme" button for player matching. I have implemented a polling function here to check the matching status. So, after clicking "pairme", you don't need to perform any other actions. After approximately 5 seconds, the system will automatically assign a player and set game permissions for you.
5) Once the player matching is completed, the first player to enter (Player1) will have the permission to make the first move (click the "sendmove" button), while the second player (Player2) will see the "gettheirmove" button. Although both players can interact with the game board, please avoid moving the pieces without permission. While the program can still record the move information, if Player2 moves a piece to position 4a and then Player1 also moves a piece to 4a and sends the move, when Player2 receives the move, both pieces will appear at position 4a simultaneously. As I have not implemented locking operations here, please refrain from moving the pieces when it is not your turn. Additionally, regarding capturing pieces, I have assumed that when one's own piece touches the opponent's piece, the own piece will be removed. This design may have some issues, but it does not affect the normal sending and receiving of move information. Also, if you move a piece, for example, changing its position or moving it multiple times but eventually returning it to its original position, it will still be considered as a move, and the move information will be sent and received normally. Therefore, if you want to see noticeable effects, it is recommended to move the pieces to empty spaces.
6) If the opponent has not made a move yet, continuously clicking "gettheirmove" will display the message "Your opponent has not made a move yet" at the bottom of the screen. If you click "sendmove" before making a move, you will also receive a prompt message indicating that you cannot send a move without making one.
7) When either player decides to quit the game, the system will display the message "Successfully quit the game". The other player can still click "send" and "get" buttons, but a prompt message will appear indicating that the opponent has quit the game. After quitting the game, the game board will be reset. As long as the server is still running, you can re-enter the game matching with the current username.
8) Browser server output logs: Firstly, during the registration, the program establishes the initial connection with the server and assigns two threads. Then, when clicking "pairme", the program establishes a second connection with the server and assigns two new threads. During the game interaction, each player will use their own thread and the respective port number for communication. Finally, when quitting the game, the program will close the relevant threads.

B. Telent test

After starting the server, you can connect to it by entering telnet localhost 8000 in the console. Each Telnet window will be assigned a single thread. Upon successful connection, you will see a new, cleared page where you can conduct testing. Here's an important tip: when entering commands, do not include extra spaces or carriage returns. Commands should follow one after the other, or the server connection may be disrupted. If a disconnect happens, just input telnet localhost 8000 to reconnect; the previous game test data will still be preserved, and you can continue with your testing. To emphasize, after each command is sent and server response is received, do not enter a space or a carriage return, but continue entering commands at the current cursor position, or the server may disconnect.

Next, we will proceed with the command test sequence:

1) Register in two windows:

   GET /register HTTP/1.1
   Host: localhost

2) Pair players in two windows:

   GET /pairme?player=name HTTP/1.1
   Host: localhost

3) After pairing, manually synchronize the game data:

   GET /gamestate?player=name HTTP/1.1
   Host: localhost

4) Then, you can start sending game data. Note that only the player with "Authenticate" can

send data, otherwise, "It's not your turn" will be returned. And, piece movement should be sent in encoded format:[{%22piece%22:%22img-7d%22, %22from%22:%22a7%22,%22to%22:%22a5%22}]

```
GET /mymove?player=name&id=Id&move=move HTTP/1.1
Host: localhost
```

5) After sending the data, the opponent can receive data:

```
GET /theirmove?player=name&id=Id HTTP/1.1
Host: localhost
```

6) Subsequently, data interchange can take place: one side sends first, then the other side receives and sends, in this alternating manner.
7) Lastly, to quit the game:

```
GET /quit?player=name&id=GameId HTTP/1.1
Host: localhost
```

This testing procedure covers all aspects of the game process: registration, pairing, data interchange, and finally, exiting. Each step can be done through the corresponding commands in the Telnet window. During this process, be sure not to enter a space or a carriage return between sending commands and receiving responses, to prevent disconnection from the server.

References

[1] Microsoft, "A tour of C# - Overview," learn.microsoft.com, Sep. 29, 2022. https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

[2] IEvangelist, "Use Sockets to send and receive data over TCP - .NET," learn.microsoft.com, Dec. 01, 2022. https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/sockets/socket-services#create-a-socket-server

[3] ChatGPT,https://chat.openai.com/

Appendix

During the course of this assignment, I encountered many concepts and knowledge that I did not understand. To help me understand and address these issues, I sought assistance from ChatGPT to gain insights and improve the game logic. I have analyzed and discussed some of the challenges in the problem section. Overall, this assignment has been a learning challenge for me, as I explored topics such as thread handling, implementing Telnet commands in the server, and dealing with locks. However, there are still many unresolved issues, such as potential thread conflicts when multiple game sessions are in progress, and the lack of robust error handling for Telnet commands. While I have made progress through my efforts and interactions with ChatGPT, I have only been able to meet basic operational requirements. Despite the limitations, this assignment has provided me with valuable learning opportunities and allowed me to gain insights through my efforts and interactions with ChatGPT.