Multiplayer Online Matching Tic-Tac-Toe

Yulin Fu
Computer Science
University of Auckland
Auckland, New Zealand
yxue732@aucklanduni.ac.nz

***Abstract*—This report focuses on the design and implementation of a server for a two-player board game. It discusses the creation of a server to handle multiple clients and facilitate multiplayer interactions in the context of a Tic-Tac-Toe game. The report explores alternative approaches, critically analyzes the choices made, and provides a comprehensive overview of the design and implementation process. An evaluation of the server's performance and functionality is included, along with an appendix summarizing the lessons learned and personal reflections on the assignment.**

Keywords—C#, Javascript, html, css, server, client.

I. Problem discussion

For this project, i have requested the design and implementation of a multi-threaded game server and further development of a playable two-player board game, such as Checkers, Chess, or Go. The server needs to be built using the C# programming language and synchronous server sockets, and it should communicate with the game client using HTTP REST as the foundation. The server needs to implement a series of GET endpoints, including /register, /pairme, /mymove, /theirmove, and /quit. Additionally, the server should handle error scenarios and incorporate appropriate concurrency control measures.

To accomplish this project, i have considered the following solutions:

Programming Language: I have chosen to use C# due to its robust object-oriented programming capabilities and extensive support from the .NET libraries.

Synchronous Server Sockets: I require building the server from scratch using synchronous server sockets, without utilizing higher-level APIs. This means you will handle the details of socket connections, requests, and responses manually.

Multi-threading for Client Connections: Since the server needs to handle multiple client connections, I plan to use a thread pool to process each client connection, which will improve efficiency.

Designing Appropriate Data Structures: I need suitable data structures to store player information, game records, etc. I can utilize data structures like dictionaries to manage and query this information.

Error Handling: It is crucial to implement proper error handling logic in the server. This includes responding correctly to invalid endpoints, parameters, etc., and returning appropriate HTTP status codes.

Next, I will divide the project into server side and client side respectively to explain.

II. Server Design and Critical Analysis

First, the main function.

```
static void Main(string[] args)
{
    Console.WriteLine("Server starting!");

    IPAddress ipAddr = IPAddress.Any;
    int port = 11000;

    TcpListener listener = new TcpListener(ipAddr, port);
    listener.Start();

    while (true)
    {
        TcpClient client = listener.AcceptTcpClient();
        ThreadPool.QueueUserWorkItem(HandleClient, client);
    }
}
```

This method is the entry point to the application, which starts the server and accepts client connections[1][3].

Code used in the multithreaded way of dealing with client connection, through the use of the thread pool ThreadPool. The QueueUserWorkItem method, can handle multiple concurrent connections. However, the way thread pools are used can lead to resource contention and performance issues, especially at high concurrency. An alternative is to use the Asynchronous Programming Model (APM) or the Async/Await Pattern to handle client connections.

And then is the HandleClient function:

```
while (!isEndOfRequest && (bytesRead = stream.Read(buffer, 0, buffer.Length)) > 0)
{
    string data = Encoding.ASCII.GetString(buffer, 0, bytesRead);
    requestBuilder.Append(data);

    if (requestBuilder.ToString().Contains("\r\n"))
    {
        isEndOfRequest = true;
    }
}
```

This method is used to handle requests and responses for a single client connection[3].

When processing a request, the code reads the network stream to get the data sent by the client and converts the data into a string for processing. However, this approach may run into the problem of data fragmentation, where requests may be split into multiple network packets that require more complex processing. The alternative could be to use StreamReader to read the network stream and the ReadLine method to read the request line by line.

ProcessRequest function:

```
static string ProcessRequest(string request, IPEndPoint clientEndPoint)
{
    string[] tokens = request.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

    if (tokens.Length < 2 || tokens[0] != "GET")
    {
        return "HTTP/1.1 400 Bad Request\r\n\r\n";
    }
```

```
else if (url.StartsWith("/pairme?player="))
{
    string playerName = GetParameter(url, "player");
    string gameRecordId = FindWaitingGameRecord(playerName);

    if (gameRecordId == null)
    {
        Random random = new Random();
        gameRecordId = random.Next(1000, 9999).ToString();

        GameRecord gameRecord = new GameRecord(gameRecordId, playerName, "wait");
        lock (lockObj)
        {
            gameRecords.Add(gameRecordId, gameRecord);
        }
        return $"\r\n{gameRecord.ToString()}\r\n";
    }
```

This method handles different operations depending on the

URL requested by the client.

The current implementation uses string matching and segmentation to parse the request URL, which may not be flexible or reliable. A better alternative is to use ASP. Net Core or a similar Web framework to handle requests and routes, which makes it easier to define and process different routes and operations.

GetParameter function:

```
8 个引用
static string GetParameter(string url, string paramName)
{
    int index = url.IndexOf(paramName + "=");
    if (index == -1)
        return null;

    int startIndex = index + paramName.Length + 1;
    int endIndex = url.IndexOf("&", startIndex);
    if (endIndex == -1)
        endIndex = url.Length;

    return url.Substring(startIndex, endIndex - startIndex);
}
```

This method is used to get the value of the parameter from the URL.

Current implementations use string matching and interception to extract parameter values, which can run into problems with string parsing and boundary handling. A better alternative is to use an existing URL resolution library or regular expression to extract and parse the parameters.

FindWaitingGameRecord function:

```
static string FindWaitingGameRecord(string playerName)
{
    foreach (var gameRecord in gameRecords.Values)
    {
        if (gameRecord.State == "wait")
        {
            return gameRecord.GameId;
        }
    }
    return null;
}
```

This method is used to find game records in waiting.

Current implementations traverse the game record dictionary to find waiting game records, and the performance of this approach can be affected by the size of the dictionary. A better alternative is to use a more efficient data structure, such as a queue, to maintain game records while waiting for faster lookup and matching.

ToString function:

```
public override string ToString()
{
    return $"{{ \"gameId\": \"{GameId}\", \"state\": \"{State}\", \"firstPlayer\": " +
        $"\"{FirstPlayer}\", \"secondPlayer\": \"{SecondPlayer}\", \"lastMovePlayer1\": " +
        $"\"{LastMovePlayer1}\", \"lastMovePlayer2\": \"{LastMovePlayer2}\" }}";
}
```

This method is used to store all the details of a player's match.

The following is a demonstration of the concrete implementation results of the code (using telnet local connection):

After code runing:

```
C:\Users\27966\Desktop\井字棋-server\Server-c\bin\Debug\net7.0\Server-c.exe
Server starting!
```

The server is now waiting for a connection.

Next I'll open two windows command-line Windows to connect to the server.

```
命令提示符
Microsoft Windows [版本 10.0.19045.2965]
(c) Microsoft Corporation。保留所有权利。

C:\Users\27966>telnet localhost 11000
```

```
Server starting!
Connection established with client: 127.0.0.1:59182
Connection established with client: 127.0.0.1:59188
```

As you can see, there are already two command line Windows connected to the server.

Now let's register two players separately, using the required instruction GET /register.

```
HTTP/1.1 200 OK
Access-Control-A
Access-Control-A
Access-Control-A
Content-Length:

Player6042
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET
Access-Control-Allow-Headers: Content-Type
Content-Length: 14

Player6703
```

Two different players have signed up.

Next use GET /pairme? player=username matches.

```
GET /pairme?player=Player6703 HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET
Access-Control-Allow-Headers: Content-Type
Content-Length: 136

{ "gameId": "6100", "state": "wait", "firstPlayer": "Player6703", "secondPlayer": "", "lastMovePlayer1": "", "lastMovePl
ayer2": "" }
```

When one player enters the match state, we can see that the state is wait when no second player enters the match state. And a game id has been generated.

```
GET /pairme?player=Player6042 HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET
Access-Control-Allow-Headers: Content-Type
Content-Length: 150

{ "gameId": "6100", "state": "progress", "firstPlayer": "Player6703", "secondPlayer": "Player6042", "lastMovePlayer1":
 "", "lastMovePlayer2": "" }
```

Once a second player matches, the match state changes to progress.

Now you can use

GET/mymove?player={username}&id={gameId}&move={move},GET/theirmove?player={username}&id={gameId} to get the chess piece step.

```
GET /mymove?player=Player6703&id=6100&move=1
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET
Access-Control-Allow-Headers: Content-Type
Content-Length: 151
```

```
GET /theirmove?player=Player6042&id=6100 HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET
Access-Control-Allow-Headers: Content-Type
Content-Length: 5

1
```

Next, GET /quit? player=username&id=gameid can quit the game and delete the game.

```
GET /quit?player=Player6703&id=6100 HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET
Access-Control-Allow-Headers: Content-Type
Content-Length: 21

HTTP/1.1 200 OK
```

This is a separate result test of the server code with Telnet.

III. Client Design and Critical Analysis

On the final project completion, I implemented a two-player game of tic-tac-toe that was playable on the web.

Next, I'll analysis how the js code connects to the server.

```
function showMessage(message) {
    var p = document.createElement( tagName: "p");
    p.innerHTML = message;
    messageArea.appendChild(p);
    messageArea.scrollTop = messageArea.scrollHeight;
}
```

First, the showMessage function.

This function is used to display the message on the page. It creates a <p> element, sets the message content to the innerHTML of the element, and then adds it to the messageArea element. A possible alternative is to use a more advanced message display library or framework to simplify the code and provide more style and functionality.

```
function register() {
    fetch( input: 'http://localhost:11000/register', init: { method: 'GET', mode: 'cors' })
        .then(response => response.text()) Promise<string>
        .then(data => {
            username = data.trim();
            showMessage("Registered successfully, user name: " + username);
        });
}
```

register function: This function registers users with the server and displays the returned user name on the page. It gets the username by sending a request to the server and stores it in the username variable. A possible alternative is to use more secure user authentication methods, such as JSON Web Tokens (JWT) to manage user sessions.

```
function pair() {
    if (!username) {
        showMessage("Register for Account");
        return;
    }
    fetch( input: `http://localhost:11000/pairme?player=${username}`, init: { method: 'GET', mode: 'cors' }) Pr
        .then(response => response.text()) Promise<string>
        .then(data => {
            const gameData = JSON.parse(data);
            gameId = gameData.gameId;
            firstPlayer = gameData.firstPlayer;
            secondPlayer = gameData.secondPlayer;
            if (username == firstPlayer) {
                currentPlayer = 'O';
                theOtherPlayer = 'X';
            }
            if (username == secondPlayer) {
```

pair function: This function is used to match players and get the game ID and player character. It sends a request to the server for matching information and stores the game ID and player character in the appropriate variables. A possible alternative would be to use a more powerful matching algorithm or system to handle player matching and provide more game options (such as choosing different game modes or difficulty levels).

```
function theirmove() {
    if (!username || !gameId) {
        showMessage("Please register and match first");
        return;
    }
    fetch( input: `http://localhost:11000/theirmove?player=${username}&id=${gameId}`, init: { method: 'GET', mode: 'cors'
        .then(response => response.text()) Promise<string>
        .then(data => {
            theirmoveId = parseInt(data);
            showMessage("Opponent's move: " + theirmoveId);
            if (username == firstPlayer){
                theirturn(theirmoveId, theOtherPlayer, secondPlayer);
            }
            if (username == secondPlayer){
                theirturn(theirmoveId, theOtherPlayer, firstPlayer);
            }
        });
```

theirmove function: This function is used to get the opponent's position and update the game state. It sends a request to the server to get the opponent's location and displays it on the page. A possible alternative would be to use a real-time communication protocol such as WebSocket to handle real-time updates of the opponent's game tiles and display dynamic animations of the tiles on the game board.

```
function quit() {
    if (!username || !gameId) {
        showMessage("Please register and match first");
        return;
    }
    fetch( input: `http://localhost:11000/quit?player=${username}&id=${gameId}`, init: { method: 'GET', mode: 'cors' })
        .then(response => response.text()) Promise<string>
        .then(data => {
            showMessage(data);
            username = null;
            gameId = null;
        });
}
```

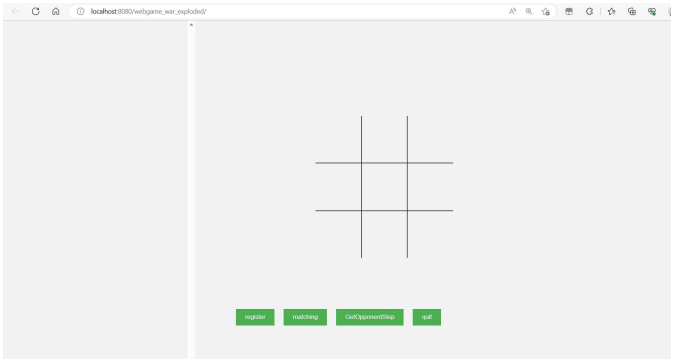
quit function: This function is used to exit the game and reset the relevant variables. It sends a request to the server to process the exit logic and displays the message returned by the server on the page. A possible alternative is to use a more elegant exit method from the game, such as by calling an exit function or method provided in the game engine or library.

```
function markMove(cellId) {
    if (!username || !gameId) {
        showMessage("Please register and match first");
        return;
    }
    const move = cellId;
    fetch( input: `http://localhost:11000/mymove?player=${username}&id=${gameId}&move=${move}`, init: { method: 'GET', mode
        .then(response => response.text()) Promise<string>
        .then(data => {
            showMessage(data);
        });
}
```

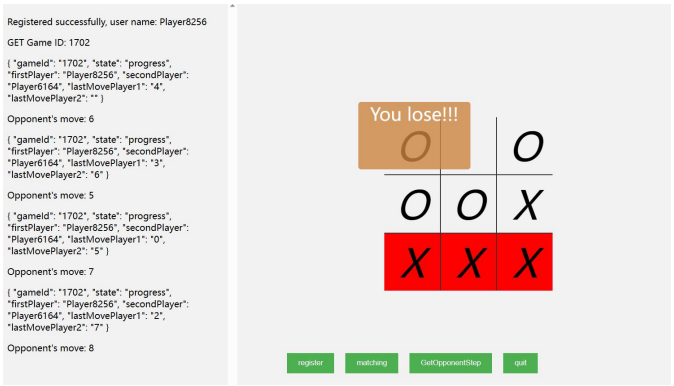
This function handles the logic of the player's moves. It updates the state of the game by sending a request to the server and displays the message returned by the server. A possible alternative is to use a more advanced game engine or library to handle game logic and status updates, such as React, Vue, or Phaser.

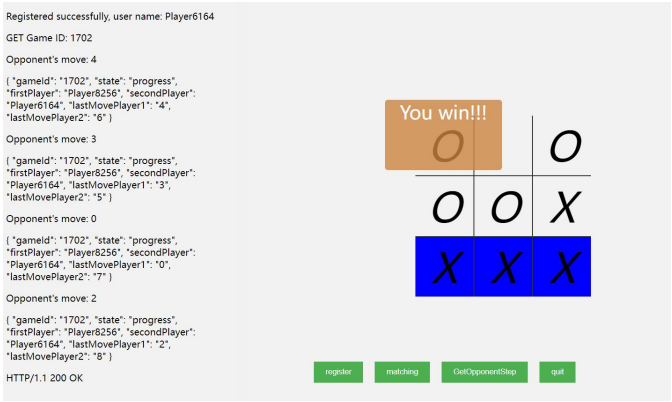
For tic-tac-toe game to determine victory and other js code, because the server is not much to do here. The following shows the concrete results of the code implementation.

After I run the web code in the server and IDEA.

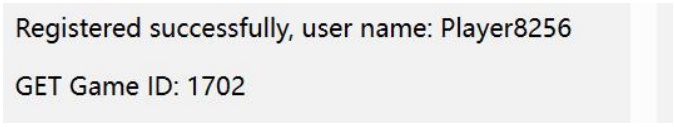


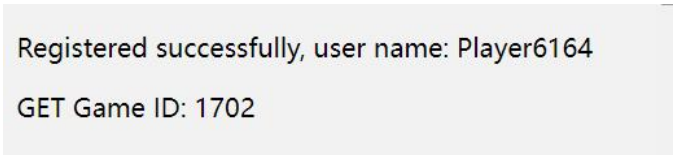
We can see the game interface as shown in the picture. On the left side of the overall page is a showMessage window, which displays the data transmitted by the server in real time. Here we open the edge and google browser pages to register and match.



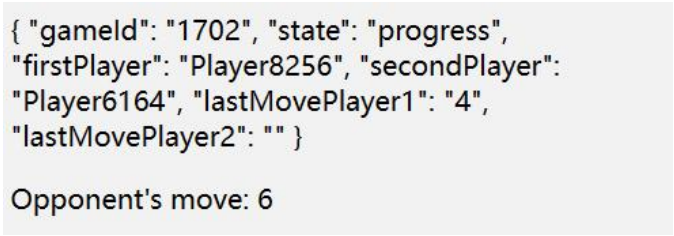


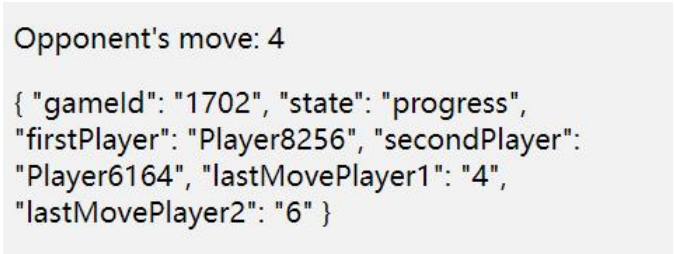
After going through a series of matchboard events, we can see that all of the matchboard data is displayed on the left, and I'll walk you through how each code data is generated in turn.





By clicking register and matching, you can see that both players have the same gameid and username, which means it's the same game.





Place a piece at a selected position on the board and click GetOpponentStep to get the position where your opponent is playing. One of the drawbacks of my code is that I can't transfer the data of my opponent's chess in real time because my connection to the server is a click event response.

After a complete match, we can see whether the chess player wins or loses. Click quit to quit the game and delete the gameid of the game.

Declaration, I refer to the code for the server side [1] and the code for the Tic-tac-toe implementation part [2].

IV. Conslusion

In the end result, I implemented a multithreaded game server suitable for two-player games such as backgammon, checkers, chess, Go, etc. Servers help match players and coordinate the exchange of game actions. In the program implementation, it still has some shortcomings, the server in a certain time after the automatic shutdown, unable to real-time access to the opponent's data, not beautiful css style and so on. But overall, I completed almost all of the project requirements.

REFERENCES

[1] Microsoft. "Create a Socket Server." Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/sockets/socket-services#create-a-socket-server. [Accessed: May 22, 2023].

[2] Gulaoyeya. "js implementation of tic-tac-toe game" CSDN. [Online]. Available: https://blog.csdn.net/pfourfire/article/details/125064111. [Accessed: May 22, 2023].

[3] "Tour of C# - C# Guide | Microsoft Docs," Microsoft Docs, [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/. [Accessed: May 22, 2023].

[4] Wikipedia, "Telnet," in Wikipedia, The Free Encyclopedia. Available: https://en.wikipedia.org/wiki/Telnet. [Accessed: May 22, 2023].

[5] allway2. "Play Tic-tac-toe with JavaScript: Create a board class" CSDN. [Online]. Available: https://blog.csdn.net/allway2/article/details/125170136.[Accessed: May 22, 2023].

[6] javaLuohuayu. "idea Building web Projects - ultra detailed tutorial" CSDN. [Online]. Available: https://blog.csdn.net/stepleavesprint/article/details/127776102?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216847499521680022282539 2%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=168474995216800222825392&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-127776102-null-null.142^v87^insert_down1,239^v2^insert_chatgpt&utm_term=idea%E6%90%AD%E5%BB%BAweb&spm=1018.2226.3001.4187.[Accessed: May 22, 2023].

Appendix

In this project, I learned how to build a server using C# and a corresponding JavaScript client to implement a simple tic-tac-toe game. The server utilizes TCP sockets to listen for client connections and processes each client's requests using a thread pool. The client communicates with the server through HTTP requests, enabling functionalities such as user registration, opponent matching, making moves, and quitting the game.

On the server side, I learned how to start a TCP listener on a specified IP address and port and handle client connections using a thread pool. I gained knowledge in receiving client request data through network streams and performing different operations based on the requested URL, such as user registration, opponent matching, making moves, and quitting the game. To ensure thread-safe access to shared data, I also learned to use lock objects for synchronization.

On the client side, I learned how to send HTTP requests to the server for user registration and obtain assigned usernames, game IDs, and opponent information. I also learned how to send HTTP requests to the server to make moves and retrieve server responses. Additionally, I learned how to send HTTP requests to the server to retrieve opponent moves and update the game state.

Through this project, I have gained a deeper understanding of building a C# server and a JavaScript client. I believe this knowledge will play an important role in my future work. I will continue to optimize and enhance these skills to tackle more complex projects and challenges.