Abstract—**This report aims to design and implement a multithreaded game server suitable for two-player games. The server is developed using the C# programming language and utilizes synchronized server sockets to establish connections with external sites. The communication between the server and game clients is based on the HTTP REST protocol. A browser-based client is built for a selected two-player game to validate the server's correctness through testing methods. Telnet is used as a testing tool to establish a remote login connection to external sites. The report discusses the problems, alternative solutions, design and implementation aspects, and evaluates the performance of the developed game server.**

Keywords—Multithreaded game server, C# programming language, HTTP REST communication.

I. INTRODUCTION

This study focuses on the design and implementation of a multithreaded game server capable of pairing players and coordinating the exchange of actions in two-player games. The server is developed using the C# programming language[1] and establishes connections with external sites through synchronized server sockets[2]. To ensure effective communication with game clients, the server supports the use of HTTP REST as the underlying protocol[3].

When designing and implementing a multithreaded game server, various alternative solutions can be considered. One possible approach is to implement the server using a different programming language such as Java or Python. However, considering the requirement to use the C# programming language, we have opted to proceed with the development in C#. Another alternative is to utilize higher-level APIs[] such as HTTPListener for handling client communication. However, given the requirement to establish connections using synchronized server sockets, the use of higher-level APIs is not permitted.

(The most important run test section is in III)

II. DESIGN & IMPLEMENT

According to the requirements, we will design a multithreaded game server based on C#. The server establishes connections using synchronized server sockets and utilizes HTTP REST as the foundation for communication with game clients. The following are the key design aspects of the server:

The server is developed as a command-line application and should be capable of running on a standard .NET 7 installation. The server employs multithreading to handle player requests and maintains active connections to process player interactions within the same thread. The server can accept connection requests from clients and record the client's IP address, port number, and accessed URL[5].

The server provides the following HTTP REST endpoints:

/register: Generates a random username for the player and registers it.

/pairme?player={username}: Attempts to pair the given player with another player.

/mymove?player={username}&id={gameId}&move={move}: Allows the player to provide their move.

/theirmove?player={username}&id={gameId}: Retrieves the move made by the other player.

/quit?player={username}&id={gameId}: Records the player's intention to quit the game.

The server employs appropriate concurrency control to avoid potential contention issues and ensure secure information sharing among threads.

Based on the design, we have implemented a multithreaded game server using the C# programming language. The server establishes connections to external sites using synchronized server sockets and utilizes HTTP REST as the communication protocol with game clients. We followed the following steps during the development process:

Created a C# command-line application and set up the basic project structure[1].

Implemented the main logic of the server, including accepting client connection requests, handling HTTP requests, and parsing request parameters.

```
Socket client = (Socket)state;
try
{
    // Fetch request data
    byte[] buffer = new byte[1024];
    StringBuilder requestBuilder = new StringBuilder();

    while (true) // A loop has been added to maintain a connection to the client
    {
        int length = client.Receive(buffer);
        string part = Encoding.UTF8.GetString(buffer, 0, length);
        requestBuilder.Append(part);

        // Check whether the request ends (HTTP request ends with a blank line)
        if (part.EndsWith("\r\n"))
        {
            string request = requestBuilder.ToString();

            Console.WriteLine("Sent response to IP: " + ((IPEndPoint)client.RemoteEndPoint).Addres

            // Parse HTTP request
            HttpRequest httpRequest = HttpRequest.Parse(request);
            // Processing HTTP requests
            HttpResponse httpResponse = HandleHttpRequest(httpRequest);
            // Send the response data to the client
            byte[] responseBytes = Encoding.UTF8.GetBytes(httpResponse.ToString());
            client.Send(responseBytes, SocketFlags.None);
            // Empty the request constructor to prepare the next request
            requestBuilder.Clear();
        }
    }
}
catch (Exception ex)
```

Designed and implemented appropriate data structures to manage players and game records.

```
class Player
{
    public string Username { get; set; }

    public Player(string username)
    {
        Username = username;
    }
}

class Game
{
    public string Id { get; set; }
    public string State { get; set; }
    public Player Player1 { get; set; }
    public Player Player2 { get; set; }
    public string LastMove1 { get; set; }
    public string LastMove2 { get; set; }

    public Game(Player player1)
    {
        Id = new Random().Next(00, 99).ToString();
        Player1 = player1;
    }
}
```

Implemented HTTP REST endpoints for registration, pairing,

movement, and quitting functionalities.

/register:

```
//Register
private static HttpResponse HandleRegister(HttpRequest httpRequest)
{
    string username = "Player" + new Random().Next(100, 999);
    Player player = new Player(username);
    players.Add(username, player);
    return new HttpResponse(HttpStatusCode.OK, "Username: " + username);
}
```

/pairme?player={username}:

```
//Pair
private static HttpResponse HandlePairme(HttpRequest httpRequest)
{
    string username = httpRequest.Query["player"];
    if (players.ContainsKey(username))
    {
        Player player = players[username];
        Game game = games.Values.FirstOrDefault(g => g.Player2 == null);
        if (game == null)
        {
            game = new Game(player);
            games.Add(game.Id, game);
            game.State = "Waiting";
            return new HttpResponse(HttpStatusCode.OK, $"You are Player1. Waiting for another player. Game ID: {game.Id}");
        }
        else
        {
            game.Player2 = player;
            game.State = "Gaming";
            return new HttpResponse(HttpStatusCode.OK, $"You are Player2. Game started. Game ID: {game.Id}");
        }
    }
    else
    {
        return new HttpResponse(HttpStatusCode.BadRequest, "Invalid player");
    }
}
```

/mymove?player={username}&id={gameId}&move={move}:

```
//Mymove
private static HttpResponse HandleMymove(HttpRequest httpRequest)
{
    string username = httpRequest.Query["player"];
    string gameId = httpRequest.Query["id"];
    string move = httpRequest.Query["move"];
    if (players.ContainsKey(username) && games.ContainsKey(gameId))
    {
        Game game = games[gameId];
        if (game.Player1.Username == username)
        {
            game.LastMove1 = move;
            return new HttpResponse(HttpStatusCode.OK, "Player1 moved. Move accepted");
        }
        else if (game.Player2 != null && game.Player2.Username == username)
        {
            game.LastMove2 = move;
            return new HttpResponse(HttpStatusCode.OK, "Player2 moved. Move accepted");
        }
        else
        {
            return new HttpResponse(HttpStatusCode.BadRequest, "Invalid player");
        }
    }
    else
    {
        return new HttpResponse(HttpStatusCode.BadRequest, "Invalid player or game");
    }
}
```

/theirmove?player={username}&id={gameId}:

```
//Theirmove
private static HttpResponse HandleTheirmove(HttpRequest httpRequest)
{
    string username = httpRequest.Query["player"];
    string gameId = httpRequest.Query["id"];
    if (players.ContainsKey(username) && games.ContainsKey(gameId))
    {
        Game game = games[gameId];
        if (game.Player1.Username == username)
        {
            return new HttpResponse(HttpStatusCode.OK, "Player1 move: " + game.LastMove2);
        }
        else if (game.Player2 != null && game.Player2.Username == username)
        {
            return new HttpResponse(HttpStatusCode.OK, "Player2 move: " + game.LastMove1);
        }
        else
        {
            return new HttpResponse(HttpStatusCode.BadRequest, "Invalid player");
        }
    }
    else
    {
        return new HttpResponse(HttpStatusCode.BadRequest, "Invalid player or game");
    }
}
```

/quit?player={username}&id={gameId}:

```
//Quit
private static HttpResponse HandleQuit(HttpRequest httpRequest)
{
    string username = httpRequest.Query["player"];
    string gameId = httpRequest.Query["id"];
    if (players.ContainsKey(username) && games.ContainsKey(gameId))
    {
        Game game = games[gameId];
        if (game.Player1.Username == username || (game.Player2 != null && game.Player2.Username == username))
        {
            games.Remove(gameId);
            return new HttpResponse(HttpStatusCode.OK, $"Game quit. Game: {game.Id} has closed.");
        }
        else
        {
            return new HttpResponse(HttpStatusCode.BadRequest, "Invalid player");
        }
    }
    else
    {
        return new HttpResponse(HttpStatusCode.BadRequest, "Invalid player or game");
    }
}
```

Used appropriate concurrency control to ensure secure information sharing among threads.

Performed testing and debugging to ensure the server can communicate and interact correctly with game clients.

On the client side, I have designed a two-player battleship game. Typically, the two players use black and white pieces respectively, taking turns to place them on the board. The first player to form a line of five pieces either horizontally, vertically, or diagonally wins[4].

Here are its key features and functionalities:

User Interface: Create an intuitive and user-friendly interface using HTML, CSS, and JavaScript to display the game board and the game state[7]. The interface should include a start game button, grid cells for the board, and a game information area.

Game Logic: Implement the game rules and logic of the battleship game, including board initialization, move validation, and victory condition checking[8].

Remote Connection[5][6]: Enable two-player remote connection using the HTTP protocol. The client should be able to connect to the server and communicate with the opponent. JavaScript's HTTP API can be used to handle connections, sending, and receiving messages.

/register:

```
// Register button click event handler
document.getElementById('registerButton').addEventListener('click', function() {
    fetch('http://localhost:11000/register', { method: 'GET', mode: 'cors' })
        .then(response => response.text())
        .then(data => {
            username = data.split(": ")[1];
            console.log("Successful registration, Username: " + username);
        });
});
```

/pairme?player={username}:

```
// Pair button click event handler
document.getElementById('pairButton').onclick = function() {
    fetch(`http://localhost:11000/pairme?player=${username}`, { method: 'GET', mode: 'cors' })
        .then(response => response.text())
        .then(data => {
            gameId = data.split("Game ID: ")[1];
            console.log("Successful pairing, Game ID: " + gameId);
            // Assuming the first player paired is always black
            playerColor = data.includes("Player1") ? 0 : 1;
            console.log(playerColor);
        });
}
```

/mymove?player={username}&id={gameId}&move={move}:

```
this.onclick = null; // Click and cancel the click event
if (playerColor == 0 && first == 0){
    fetch( input: `http://localhost:11000/mymove?player=${username}&id=${gameId}&move=${move}` , init: { method: 'GET', mode: 'cors' }) Pro
        .then(response: Response => response.text()) Promise<string>
        .then(data: string => {
            console.log(data);
        });
    first = 9999;
}
else{
    // Send location information to the server
    fetch( input: `http://localhost:11000/mymove?player=${username}&id=${gameId}&move=${move}` , init: { method: 'GET', mode: 'cors' }) Pro
        .then(response: Response => response.text()) Promise<string>
        .then(data: string => {
            console.log(data);
        });
    first = 9999;
}
```

/theirmove?player={username}&id={gameId}:

```
// Move button click event handler
document.getElementById( elementId: 'moveButton').onclick = function() {
    fetch( input: `http://localhost:11000/theirmove?player=${username}&id=${gameId}` , init: { method: 'GET', mode: 'cors' }) P
        .then(response: Response => response.text()) Promise<string>
        .then(data: string => {
            console.log("Opponent's move: " + data);
            var opponentPosition = [...] = convertMoveToPosition(data);
            updateBoard(opponentPosition, isBlack: playerColor === 0);
        });
}
```

/quit?player={username}&id={gameId}:

```
// Quit button click event handler
document.getElementById( elementId: 'quitButton').onclick = function() {
    fetch( input: `http://localhost:11000/quit?player=${username}&id=${gameId}` , init: { method: 'GET', mode: 'cors' }) Pr
        .then(response: Response => response.text()) Promise<string>
        .then(data: string => {
            console.log(data);
            username = null;
            gameId = null;
        });
}
```

Game State Synchronization: Ensure that the game state is synchronized between the two clients. Whenever a player makes a move, send the move to the server and broadcast it to the other player. The client should be able to receive the opponent's moves and update the game state accordingly.

```
function updateBoard(position, isBlack) {
    var piece: Element = document.querySelector( selectors: `.piece-${position[0]}-${position[1]}` );
    if (piece) {
        if (isBlack) {
            piece.style.backgroundColor = "#fff"; // white
            white.push(piece.className.match(reg));
            victory(white, c: 1); //Judge whether White wins or not

        } else {
            piece.style.backgroundColor = "#000"; // black
            black.push(piece.className.match(reg));
            victory(black, c: 0); //Judge whether Black wins or not
        }
    } else {
        console.error('No piece found at position');
    }
}
```

Exception Handling: Handle exceptions such as connection interruptions, timeouts, and error messages, providing appropriate prompts and error handling to the users.

Interface Beautification: Enhance the client interface with CSS styles[7] and graphic design to make it visually appealing and engaging.

```
style.css
.piece {
    border-spacing: 0px;
    border: 1px solid;
}

#buttonContainer {
    position: fixed;
    bottom: 0;
    left: 50%;
    transform: translateX(-50%);
}

button {
    display: block;
    margin: 20px auto;
    padding: 10px 20px;
    font-size: 18px;
    color: white;
    background-color: #007BFF;
    border: none;
    border-radius: 5px;
}

button:hover {
    background-color: #0056b3;
}
```

I have implemented all the server functions and most of the gobang rules.

III. ASSESSMENT & TEST

After the design and implementation process, our multi-threaded game server has been successfully completed and meets the requirements. The server is capable of pairing players and coordinating the exchange of actions in a two-player game. By using the C# programming language and synchronous server sockets for connection, we were able to establish links with external sites. With the use of HTTP REST as the communication protocol, the server efficiently handles requests from game clients and provides appropriate responses.

During the implementation, we took concurrency control and thread safety into full consideration and implemented appropriate measures to ensure the safety of information sharing between threads. The server has been tested and debugged to handle multiple concurrent connections and maintain active connections to serve clients.

Below are the Telnet tests[3]: **Drag server-kcui996 into Visual Studio to open it After running the server, open two command prompt windows** and enter "**telnet localhost 11000**" in each window. Perform the following HTTP command tests in the two windows:

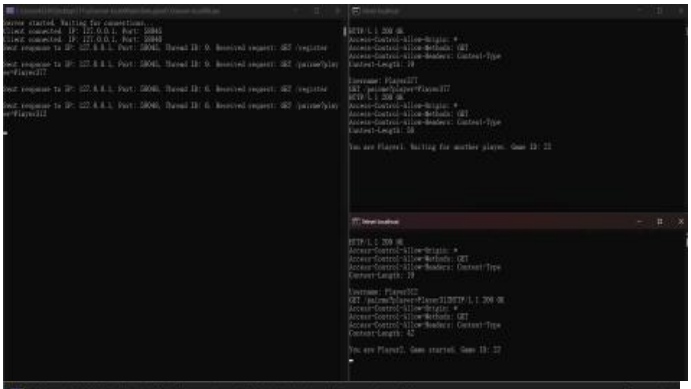
GET /register HTTP/1.1

GET /pairme?player={username} HTTP/1.1

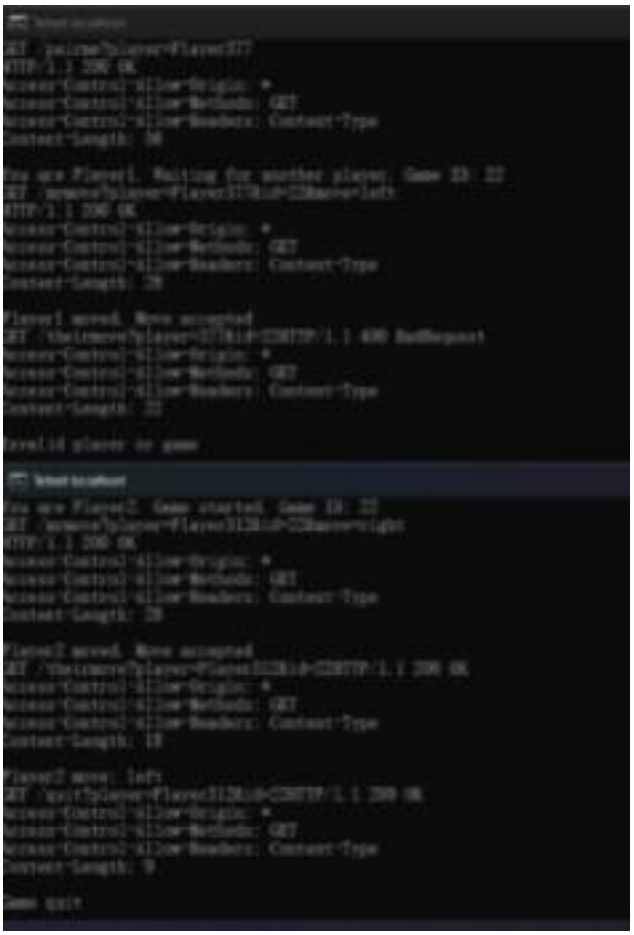
GET /mymove?player={username}&id={gameId}&move={move} HTTP/1.1

GET /theirmove?player={username}&id={gameId} HTTP/1.1

GET /quit?player={username}&id={gameId} HTTP/1.1

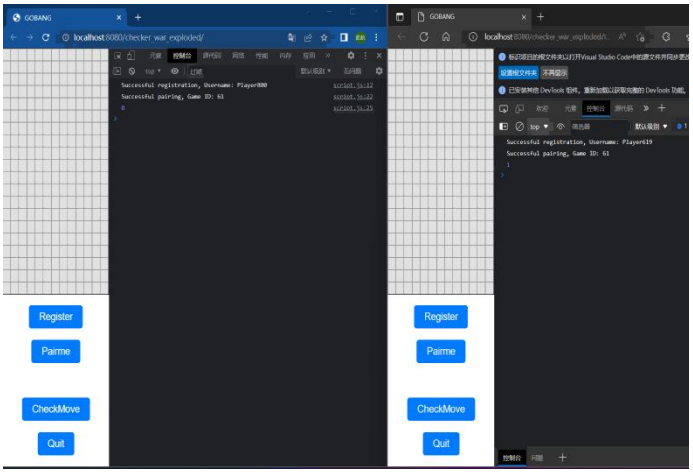


```
C:\Users\42239\Desktop\711a2\server-kcui996\bin\Debug\net7.0\server-kcui996.exe
Server started. Waiting for connections...
Client connected. IP: 127.0.0.1, Port: 58045
Client connected. IP: 127.0.0.1, Port: 58048
Sent response to IP: 127.0.0.1, Port: 58045, Thread ID: 9. Received request: GET /register

Sent response to IP: 127.0.0.1, Port: 58045, Thread ID: 9. Received request: GET /pairme?play
er=Player377

Sent response to IP: 127.0.0.1, Port: 58048, Thread ID: 6. Received request: GET /register

Sent response to IP: 127.0.0.1, Port: 58048, Thread ID: 6. Received request: GET /pairme?play
er=Player312

Sent response to IP: 127.0.0.1, Port: 58045, Thread ID: 9. Received request: GET /mymove?play
er=Player377&id=22&move=left

Sent response to IP: 127.0.0.1, Port: 58048, Thread ID: 6. Received request: GET /mymove?play
er=Player312&id=22&move=right

Sent response to IP: 127.0.0.1, Port: 58048, Thread ID: 6. Received request: GET /theirmove?p
layer=Player312&id=22

Sent response to IP: 127.0.0.1, Port: 58048, Thread ID: 6. Received request: GET /quit?player
=Player312&id=22

Sent response to IP: 127.0.0.1, Port: 58045, Thread ID: 9. Received request: GET /theirmove?p
layer=377&id=22
```

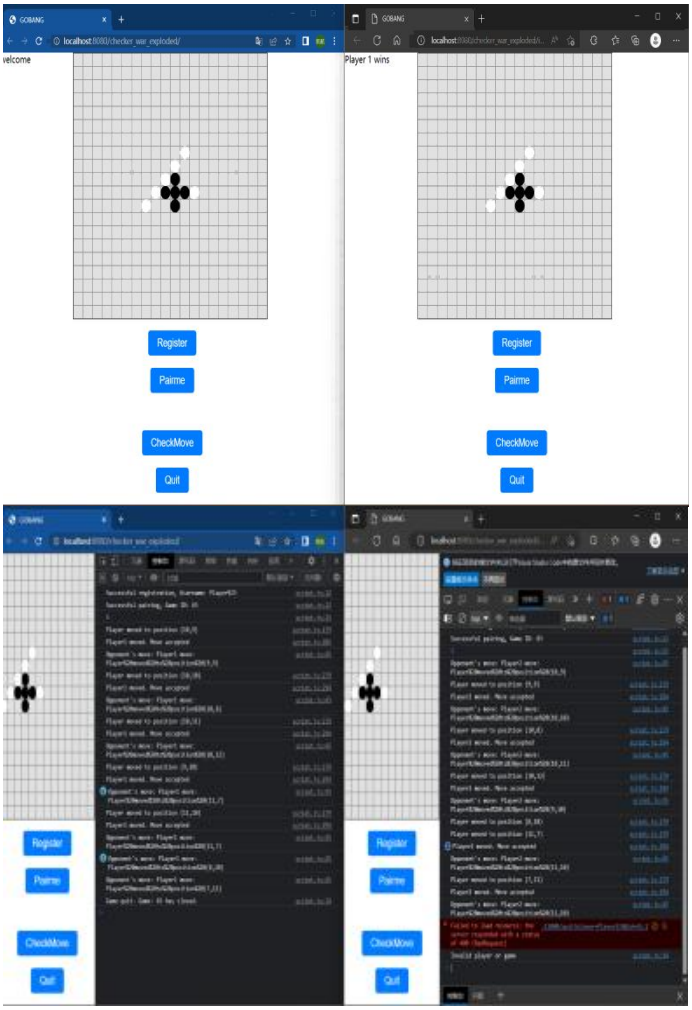


The server program performs perfectly in Telnet for all the tests, fulfilling all the requirements specified in the assignment. In the aforementioned tests, Player 1 is created and then matched, generating a game ID. Player 2 searches for an available game ID for matching and enters the game, where Player 1 makes a move and Player 2 can retrieve the move. Once a player quits, the other player is unable to make further moves. When a player exits, attempting to make a move will also result in an error.

With the addition of the client, the client is developed using the **WEB-JS** framework and compiled with **IDEA**[4]. **First, run the server, and then run the client. Open two browsers, such as Edge and Google Chrome. Each browser represents one player. Click the "register" button and the "pair" button in each browser respectively.**



In the game, 0 represents Player 1 with black pieces, and 1 represents Player 2 with white pieces. Players can click on the positions on the board to make moves. Black pieces move first, and before each move, the "checkmove" button needs to be clicked to retrieve the move of the other player (I have not implemented real-time response for the other player to receive and display the move on the board immediately after one player makes a move).



This client implements all the functionalities except for real-time display of moves. However, due to lower-level code logic and framework, the server's CPU usage is high during the game, resulting in increasing lag. When one player makes a move, it takes a long time for the other player to receive a response after clicking the "check" button.

IV. CONCLUSION

My report provides a detailed description of the design and implementation of a multithreaded game server for two-player games, along with the corresponding functionalities and implementations of the server and client. The server is developed using the C# programming language and synchronized server sockets for connection, communicating with game clients through the HTTP REST protocol. The client is developed using the WEB-JS framework and allows for remote connection and game operations through web browsers.

The server design includes multithreading to handle player requests, record connection information, support various HTTP REST endpoints, and ensure appropriate concurrency control and thread safety. The client implements the game interface, logic, and remote connection functionality.

During testing, the server is tested using Telnet and performs HTTP command tests in different command prompt

windows. The client is tested by running it in two different browsers and performing game operations by clicking the corresponding buttons, successfully completing most of the functionalities.

Throughout the implementation process, special attention is given to concurrency control and thread safety to ensure secure information sharing among threads. With appropriate concurrency control, the server is able to handle multiple concurrent connections and maintain active connections to serve clients.

Overall, my multithreaded game server has achieved good results in design and implementation, meeting the requirements. Through this project, we have gained in-depth knowledge of multithreaded programming, network communication, and how to design and implement a powerful game server.

V. REFERENCE

[1] Microsoft Build. "A tour of the C# language" Accessed on May 14, 2023, Available online: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

[2] Microsoft Build. "Use Sockets to send and receive data over TCP" [Online]. Accessed on May 14, 2023, Available online: https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/sockets/socket-services#create-a-socket-server

[3] Wikipedia. "Telnet" [Online]. Accessed on May 14, 2023, Available online: https://en.wikipedia.org/wiki/Main_Page

[4] Wikipedia. "Gobang" [Online]. Accessed on May 19, 2023, Available online: https://zh.wikipedia.org/zh-cn/%E4%BA%94%E5%AD%90%E6%A3%8B

[5] M. Perrenoud, "An Introduction to Socket Programming in .NET using C#", CodeProject, May 29, 2008. [Online]. Available: https://www.codeproject.com/Articles/32646/An-Introduction-to-Socket-Programming-in-NET-using.

[6] Stack Overflow, "How to create a simple remote desktop application in C#?," Stack Overflow, Jun 17, 2012. [Online]. Available: https://stackoverflow.com/questions/10658934/how-to-create-a-simple-remote-desktop-application-in-c-sharp.

[7] Xiao Bai. "Write gobang using native JS" [Online]. Accessed on May 16, 2023, Available online: https://zhuanlan.zhihu.com/p/386221152

[8] "Native JavaScript implementation of the Gobang game" [Online]. Accessed on May 18, 2023, Available online: https://zhuanlan.zhihu.com/p/380797981