SEMESTER TRAINING REPORT

Remote Chess

Submitted in partial fulfillment of requirements
for the award of the degree

Bachelor of Technology
In
Information Technology
To
IKG Punjab Technical University, Jalandhar

SUBMITTED BY:
Name: Khem Singh
Roll no.: 2003346
Semester: 8th
Batch: 2020 - 2024

Under the guidance of
Mr. Kamaljit Singh
Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Chandigarh Engineering College-CGC, Landran
Mohali, Punjab - 140307

Table of Content

LG
Soft India

LG Soft India Private Limited
Embassy Tech Square, Marathahalli-Sarjapur Outer Ring Road,
Bangalore - 560 103, India
T. +91-80-6615-5000 F:+91-80-6615-5100
Website: www.lgsoftindia.com
CIN: U85110KA1998PTC023521

Date: 13-May-24

TO WHOMSOEVER IT MAY CONCERN

This is to certify that **Khem Singh** a student of **B.Tech** (Roll No. **2003346**) in **Chandigarh Engineering College** is a participant of project internship program at LG Soft India Pvt. Ltd. Bangalore, from 10th Jul 2023 to 30th Jun 2024 as a partial fulfillment for the award of his Degree Program.

He is working on a project entitled **"Vehicle Middleware Team"** with **LG Soft India Pvt. Ltd. Bangalore** from Jul 2023 to till date.

He was found to be sincere and professional in his conduct.

Sincerely,

Date: 13-May-2024

Shuvam Das
Shuvam Das

Team Leader

LG SOFT INDIA PRIVATE LIMITED
BANGALORE
INDIA

Date:

College Project Coordinator:

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the report entitled "**Remote Chess"** in partial fulfilment of the requirement for the award of Degree of **Bachelor of Technology in Information Technology** and submitted to **Chandigarh Engineering College-CGC Landran Mohali** is an original piece of project work carried out by me during the period from January 2024 to May 2024 under the supervision of Mr. Kamaljit Singh.

The matter embodied in this report has not been submitted by me for the award of any other degree from any other University/Institute.

NAME: Khem Singh
Roll No: 2003346

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Mr. Kamaljit Singh Assistant Professor Department of Information Technology Chandigarh Engineering College - CGC Landran, Mohali, Punjab	**Dr. Amanpreet Kaur** HOD IT Department of Information Technology Chandigarh Engineering College - CGC Landran, Mohali, Punjab

ACKNOWLEDGMENT

I take this opportunity to express my sincere gratitude to the Director- Principal Dr. Rajdeep Singh Chandigarh Engineering College, Landran for providing this opportunity to carry out the present work. I am highly grateful to the Dr. Amanpreet Kaur HoD-IT, Chandigarh Engineering College,

Landran (Mohali). I would like to expresses my gratitude to other faculty members of Information Technology department of CEC, Landran for providing academic inputs, guidance & Encouragement throughout the training period. The help rendered by Mr. Kamaljit Singh, Supervisor for Experimentation is greatly acknowledged. Finally, I express my indebtedness to all who have directly or indirectly contributed to the successful completion of my software training.

LIST OF FIGURES

LIST OF ABBREVIATION

JS	JAVASCRIPT
STUN	Session Traversal Utilities for NAT
NAT	Network Address Translation
UI	User Interface
JPEG	Joint Photographic Experts Group

ABSTRACT

This project presents the development of a web-based, two-player online chess game. Players establish connections through a unique ID system, allowing them to face off on a virtual chessboard.

The game facilitates real-time, strategic competition between players by leveraging the power of JavaScript libraries. Chess.js ensures the validity of each move, upholding the integrity of the game. PeerJS enables seamless peer-to-peer communication, transmitting move data and updating the game state for both players simultaneously.

Beyond core gameplay functionality, the project lays the foundation for future development. The report explores potential avenues for enhancement, such as the implementation of a matchmaking system to connect players of similar skill levels.

The addition of features custom username and sound for enhanced gameplay analysis is also included in this project.

The potential for a mobile-friendly version of the game is considered, broadening the game's accessibility and player base. This project demonstrates the successful creation of a functional online chess platform, laying the groundwork for further development and expansion of its features, ultimately creating a more robust and engaging online chess experience.

INTRODUCTION

A Web-Based Platform for Two-Player Chess Matches

This document serves as an introduction to Remote Chess, a web-application designed to facilitate two-player chess matches over the internet. The following sections will outline the motivations behind the project, its core functionalities, and the user experience considerations that informed the development process.

Project Rationale and Objectives

The primary impetus for the development of Remote Chess was to broaden the accessibility of chess. By eliminating the requirement for a physical chessboard and pieces, Remote Chess allows individuals with an internet connection to engage in the strategic game of chess, regardless of geographical constraints. This fosters inclusivity and encourages participation from a wider audience, potentially including new players who may not have access to traditional chess equipment.

Furthermore, Remote Chess strives to replicate the real-time, turn-based nature of physical chess matches. This enables players to experience the intellectual challenge and strategic decision-making inherent in chess as it unfolds move by move.

Core Functionalities

Remote Chess offers a comprehensive suite of features designed to provide a seamless and engaging online chess experience:

- **Peer-to-Peer Connections:** The application leverages peer-to-peer connection technology, facilitating direct connections between players without relying on a central server. This approach streamlines the connection process and potentially reduces latency issues.
- **Visually Representative Chessboard:** The integration of the ChessboardJS library empowers Remote Chess to present a visually appealing and interactive chessboard. Players can observe the game state and the consequences of their moves in real-time on the virtual board, mirroring the experience of a physical chess game.
- **Integrated Chat Functionality:** Recognizing the social dimension of chess, Remote Chess incorporates a chat feature. This allows players to communicate with their opponents during the game, fostering a sense of community and potentially enabling strategic discussions or friendly banter.
- **In-Game Management Tools:** To ensure a smooth gameplay experience, Remote Chess provides convenient in-game buttons for resigning when necessary, offering a draw for a peaceful resolution of the match, and initiating a rematch to continue the strategic battle.

User Experience Considerations

The development of Remote Chess prioritized the creation of a user-friendly interface. This includes the utilization of clear visuals, intuitive controls, and a design that caters to players of all skill levels, from beginners to experienced chess players.

Remote Chess offers a seamless online chess experience, but this functionality relies on the power of several key libraries. Let's delve into these libraries and explore their contributions to the application:

- **PeerJS:** This JavaScript library serves as the backbone for establishing peer-to-peer connections between players. By bypassing the need for a central server, PeerJS facilitates direct communication and potentially reduces latency issues, ensuring a smooth and responsive gameplay experience.
- **Chess.js:** Core chess logic is handled by the Chess.js library. This library provides comprehensive functionalities for representing the chessboard state, validating moves, and simulating the game engine. It essentially acts as the chess arbiter within the application, ensuring adherence to the rules and regulations of chess.
- **ChessboardJS:** The visual representation of the chessboard comes courtesy of the ChessboardJS library. This library transforms the application interface into a virtual chessboard, allowing players to visualize the game state, make their moves, and observe the consequences in real-time. By mimicking the aesthetics of a physical chessboard, ChessboardJS enhances the user experience and fosters a more immersive gameplay environment.
- **jQuery:** This popular JavaScript library provides utility functions for DOM manipulation and event handling. It likely plays a role in managing user interactions with the interface elements, such as button clicks, chat message updates, and potentially updating the visual chessboard based on game state changes.

Library Integration: A Collaborative Effort

These libraries work in concert to deliver the functionality of Remote Chess. PeerJS establishes the connection, Chess.js governs the game logic, ChessboardJS renders the visual representation, and jQuery facilitates user interactions within the application. The seamless integration of these libraries is crucial for creating a cohesive and enjoyable online chess experience.

Remote Chess takes you on a strategic journey through a series of web pages, each designed to facilitate a smooth and engaging online chess experience. Here's a breakdown of the typical flow you'll encounter:

1. Welcome and Player ID Selection (index.html):

- The starting point is likely an index.html page that welcomes you to Remote Chess.
- This page might provide a brief introduction to the application and its features.
- You'll likely encounter a form where you can enter your desired Player ID. This ID will be used to identify you during the game.
- Upon entering your ID and clicking a button, you'll be redirected to the next page.

2. Connecting with an Opponent (connect.html):

- The connect.html page serves as the battleground for initiating a chess match.
- You'll likely see your Player ID displayed for confirmation.
- This page might present a form where you can enter the Player ID of your opponent.
- Clicking a button , after entering the opponent's ID triggers the peer-to-peer connection process using PeerJS.
- The page might display a visual indicator or message while the connection is being established.

3. The Chessboard and Gameplay :

- Once the connection is established, you'll likely be redirected to the game interface .
- This page will be the heart of your chess experience, featuring a visually appealing chessboard rendered by ChessboardJS.
- You'll be able to see your opponent's username or ID.
- The chessboard will allow you to make moves by clicking on squares and dragging pieces.
- The application will likely utilize Chess.js library to validate your moves, update the game state on both players' screens, and potentially handle turn management.
- Chat functionality might be integrated into this page, allowing you to communicate with your opponent during the game.

4. Game Over and Post-Match Options:

- Upon reaching a checkmate or stalemate situation, the game will likely end.

- The application might display a clear message indicating the winner or a draw.
- Depending on the design, you might see options for resigning gracefully before checkmate or offering a draw during the game.
- A rematch button could be available, allowing you to challenge your opponent to another game.
-

Additional Considerations:

- The specific flow of pages might vary depending on the implementation details.
- Some projects might combine functionalities like player ID selection and opponent connection into a single page.
- Error handling and informative messages are crucial aspects for guiding users through unexpected scenarios during connection or gameplay.

LITERATURE SURVEY

Literature Survey: Online Chess Platforms and Development

The development of online chess platforms has flourished alongside advancements in web technologies. This literature survey explores existing research and functionalities offered by prominent online chess platforms, highlighting areas relevant to the development of the two-player online chess game project.

1. Existing Online Chess Platforms:

Chess.com and Lichess.org: These popular platforms offer a wide range of features, including matchmaking systems based on player ratings, ranked games, puzzles, and educational resources. They demonstrate the potential for expanding the current project beyond basic two-player functionality. (https://www.chess.com/ & https://lichess.org/)

Studies on Online Chess Platforms: Research by [3] examines the impact of online chess platforms on educational outcomes, particularly in mathematics and cognitive development. This suggests the potential for incorporating educational elements within the project, catering to players seeking to improve their chess skills. (https://saintlouischessclub.org/research)

2. Development Techniques and Libraries:

A Survey of Deep Learning on Chess [4]: This paper explores the application of deep learning techniques in chess engines. While not directly applicable to the current project, it highlights the ongoing advancements in AI for chess and the potential for future exploration of AI-powered opponents. (https://www.ijera.com/papers/vol10no4/Series-5/G1004053135.pdf)

Online Chess Portal-Learning and Playing [2]: This research investigates the design of online chess portals, emphasizing the importance of user interfaces and learning resources. It reinforces the value of a user-friendly interface and the potential for incorporating tutorials or basic chess lessons within the project. (https://www.econstor.eu/bitstream/10419/196101/1/ofel-2019-p425-441.pdf)

3. Core Functionalities for Online Chess Games:

Literature Review of Chess Engines [1]: This survey provides a comprehensive overview of chess engine development. While focused on AI engines, it highlights essential functionalities like move validation, board representation, and check/checkmate detection, which are crucial aspects implemented in the project using Chess.js. (https://www.ijert.org/research/literature-survey-of-chess-engines-IJERTCONV5IS01199.pdf)

4. Peer-to-Peer Communication and Real-Time Gameplay:

Research on PeerJS, the library utilized in the project, is limited within the scope of this survey due to the project's focus. However, exploring existing literature on PeerJS and its functionalities in facilitating real-time communication for online games could provide further insights.

This literature survey highlights the landscape of existing online chess platforms and development techniques. By incorporating valuable aspects from existing research and platforms, the current project has the potential to expand its functionalities and cater to a broader range of chess enthusiasts.

PROJECT DESIGN

3.1 Block diagrams

A block diagram is a graphical representation of a system or process that uses simple, standardized shapes to depict the major components or functional units, their relationships, and the flow of information or signals between them. It is a way to illustrate the architecture or structure of a system in a clear and concise manner.

The following block diagrams depicts the following processes:

1. Players access the online chess game through their web browsers.
2. Players identify themselves .
3. Players connect to the server using unique IDs.
4. The server facilitates a PeerJS connection between the players.
5. Chessboard libraries on each player's device visualize the chessboard.
6. Chess.js libraries on each player's device handle game logic, including move validation and board state management.
7. When a player makes a move, javascript code validates it and generates move data .
8. PeerJS transmits the move data in real-time to the opponent's device.
9. The opponent's device receives the move data, updates the Chess.js board state, and visually reflects the move on the chessboard.
10. This cycle continues, enabling real-time gameplay between the two players.

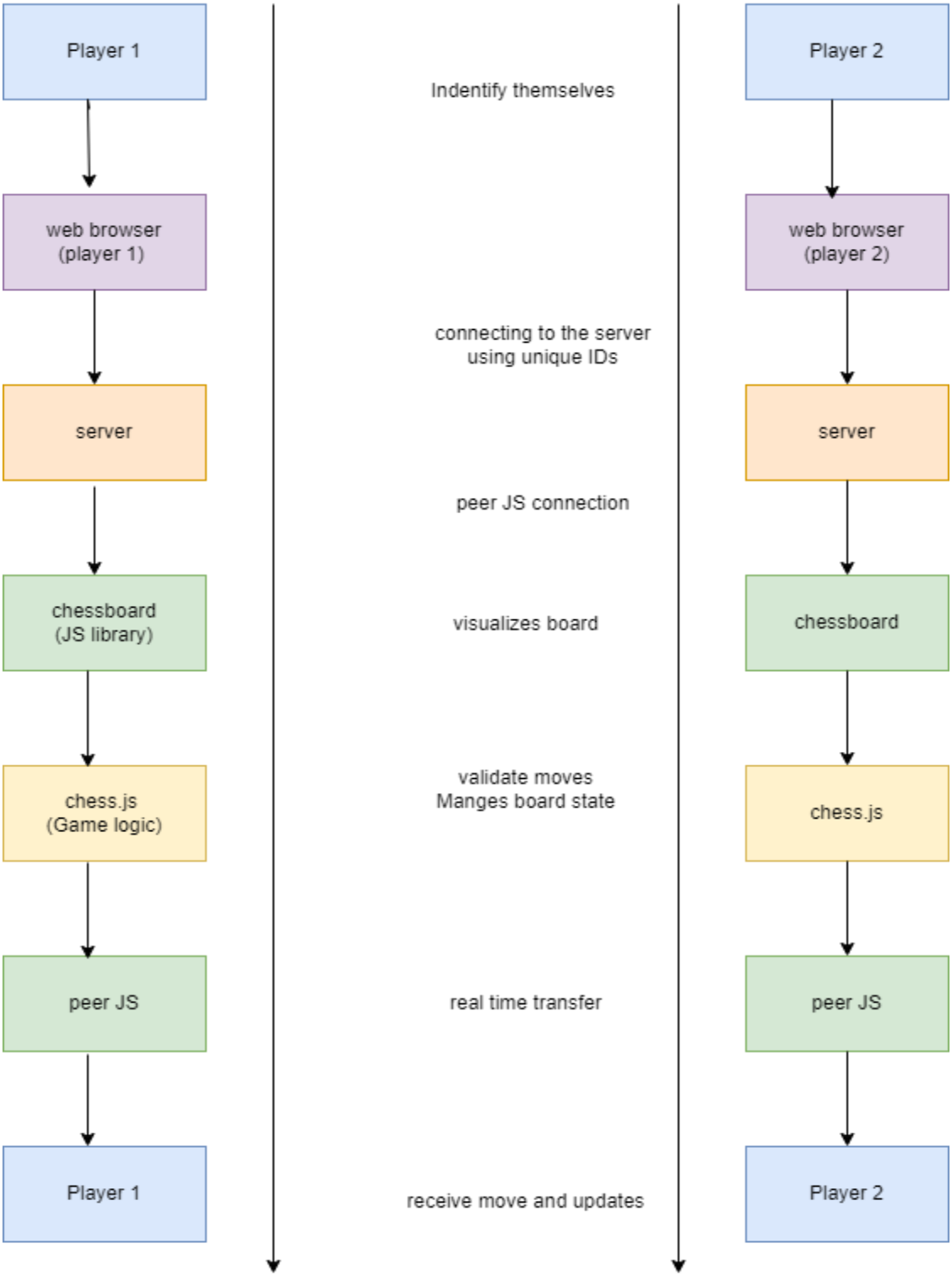


Fig 1 Design of Flow Diagram on Modular Level

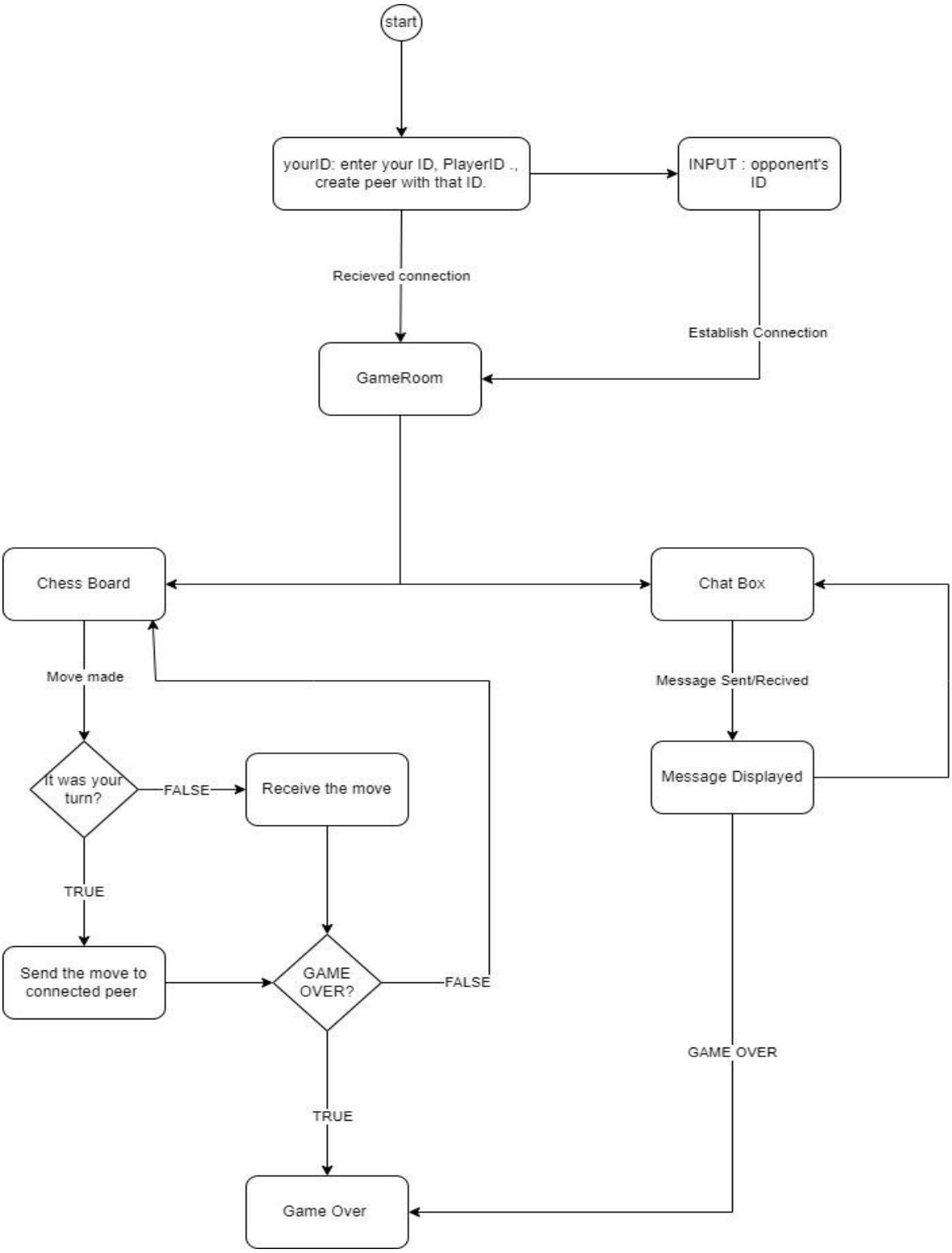


Fig 2 Design of Flow Diagram on Logical Level

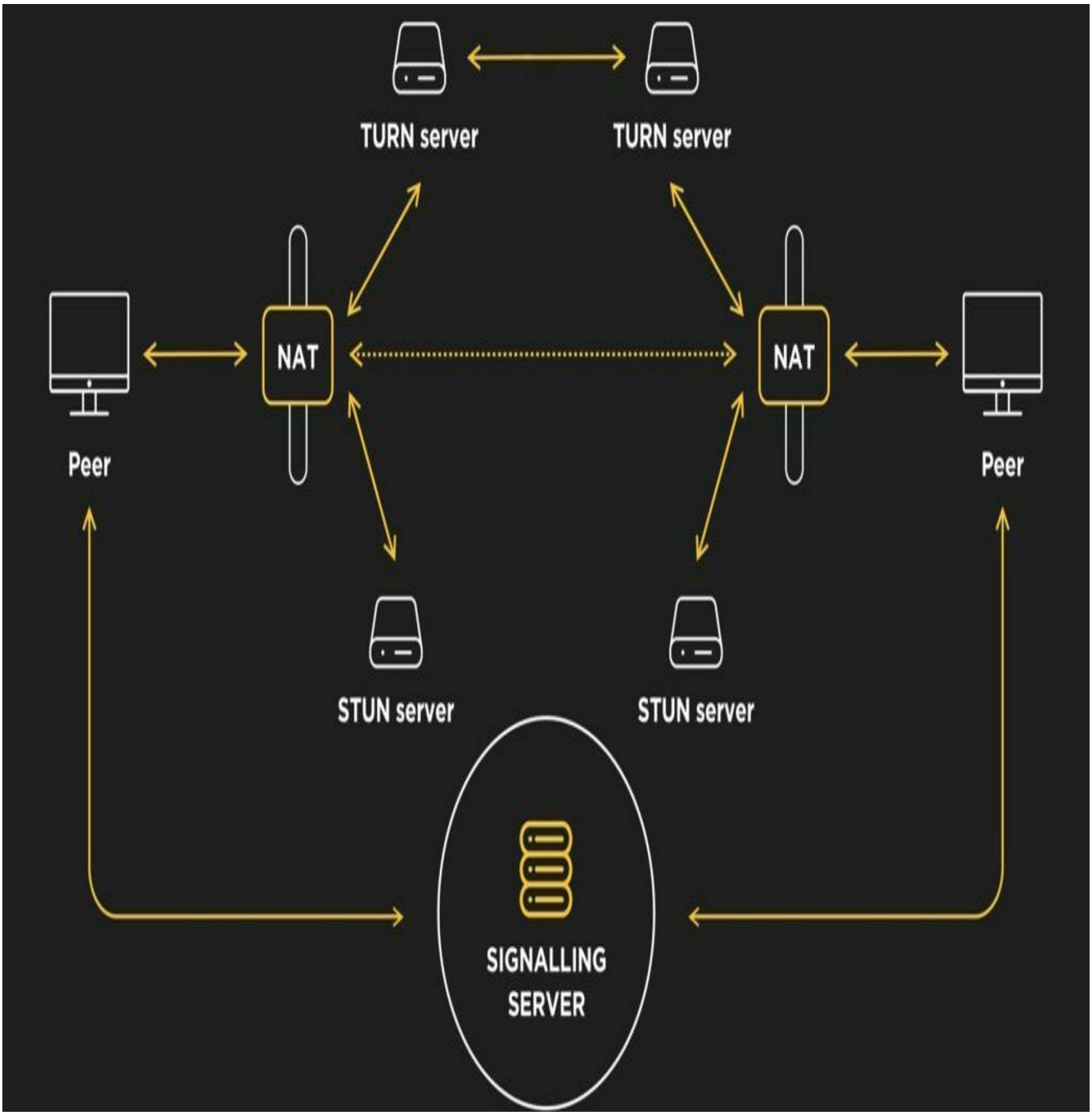


Figure 3 Peer to Peer , peer js connectivity

PeerJS provides default signaling server, or one can write their own, since the aim of the project was to establish the connection only, we used the default provided PeerJS signaling server.

Internally PeerJS API uses webRTC.

3.2 User Interface Design

Currently the Project Consists of 4 UI pages:

- **Screen to Enter User ID:**

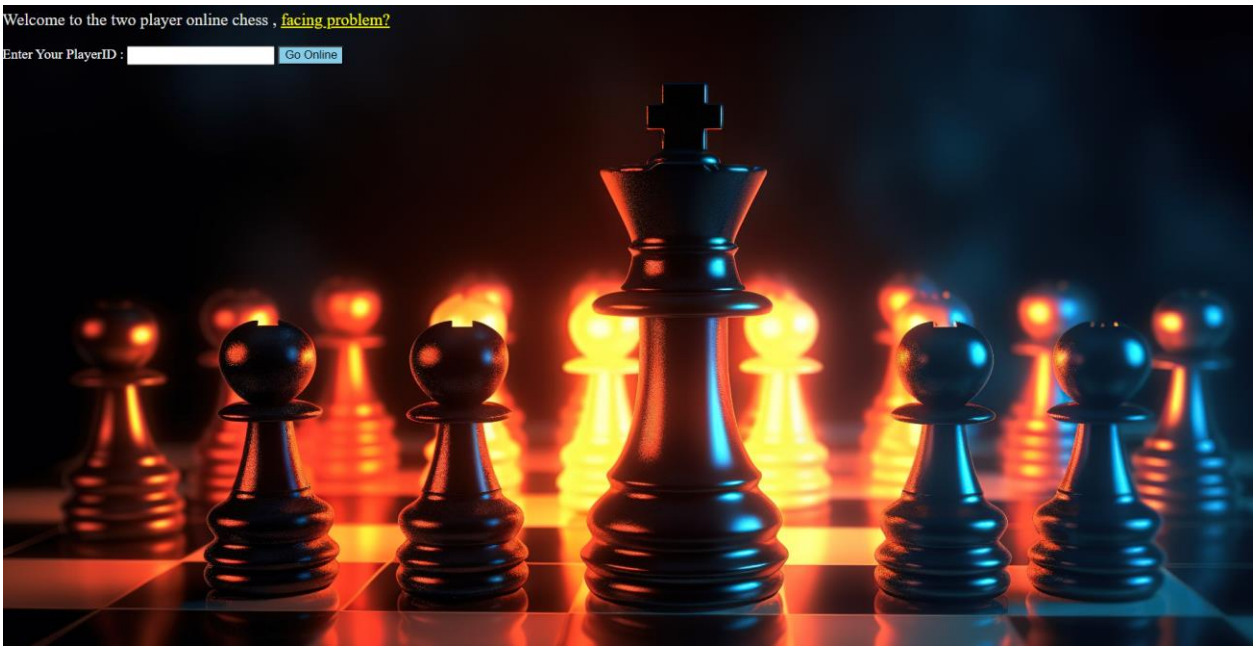


Figure 4 : User Id input screen

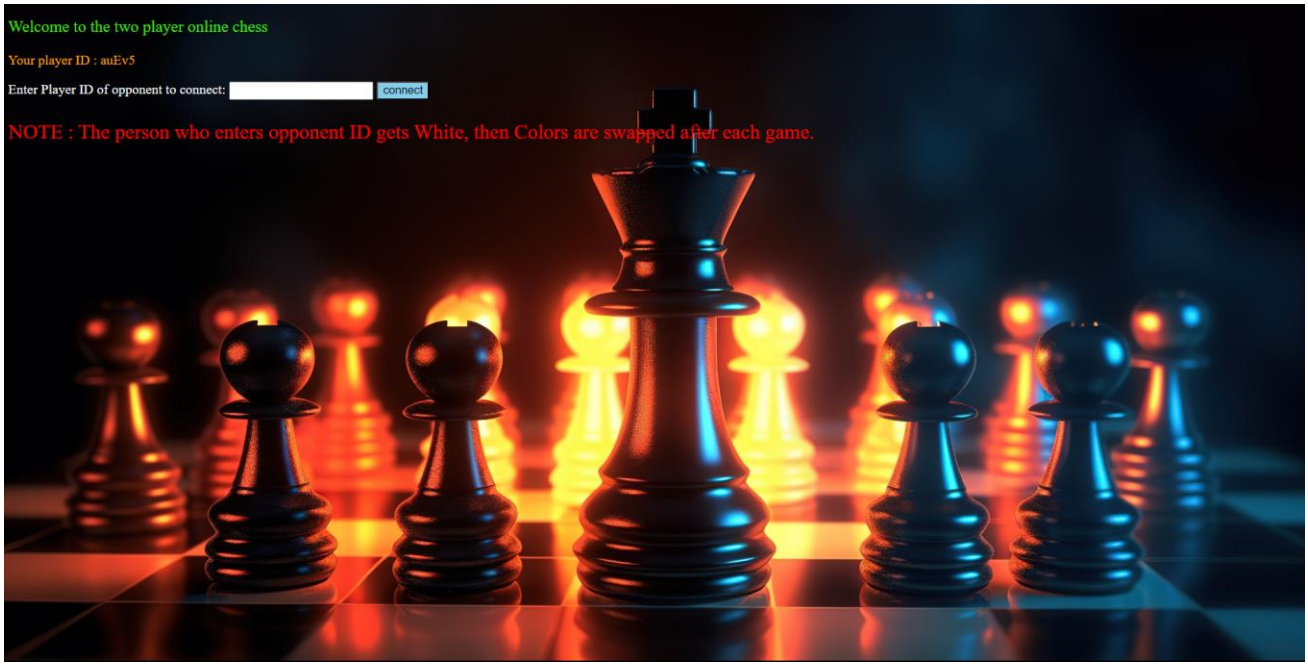
The Initial User Interface: A Formal Examination of Remote Chess' First Page

The inaugural user interface (UI) element of Remote Chess, likely designated as "index.html", serves as the point of entry for the online chess experience. This webpage lays the groundwork for subsequent interactions by introducing the application, facilitating player identification, and initiating the connection process. Let's embark on a detailed exploration of the potential elements encountered on this initial page.

- **Introductory Exposition:** The page commences with a formal welcome message that introduces the user to Remote Chess. This message should provide a concise overview of the application's purpose, explicitly stating its function as a platform for facilitating two-player chess matches over the internet. Additionally, it may highlight key features that enhance the user experience, such as real-time gameplay and an intuitive interface.
- **Visual Ambiance:** The aesthetic design of the first page plays a critical role in establishing a positive first impression. Thematic imagery or background elements that evoke chess (e.g., a chessboard or chess pieces) can effectively create a visually engaging atmosphere, subtly immersing the user in the world of chess from the outset.

- **Player Identification Protocol:** A fundamental element of the first page is a designated form for user identification. This form allows players to establish their in-game persona by entering a desired Player ID. This ID will subsequently be used to distinguish the player during the game and potentially facilitate the connection process with their opponent. The form itself may consist of a text input field where the user can enter a chosen username or a unique alphanumeric identifier.
- **Initiating the Chess Encounter:** Upon finalizing the selection of their Player ID, users will encounter a designated button (potentially labeled "Go Online" or "Start Playing") that triggers the next step in their online chess journey. Clicking this button initiates a series of behind-the-scenes processes, which may involve storing the chosen ID and redirecting the user to the subsequent webpage where opponent connection becomes possible.

⇨ **Screen for establishing Connection**



⇨

Fig 5: Screen for establishing Connection

The Arena of Connection: A Formal Analysis of Remote Chess' Second Page

The second webpage within the Remote Chess application, likely designated as "connect.html", serves as the designated arena for establishing a connection with an opponent, thereby enabling the commencement of a chess match. This page builds upon the foundation laid by the initial interface by facilitating player identification confirmation, opponent selection, and initiating the peer-to-peer connection process. Let's delve into a detailed examination of the potential elements encountered on this secondary webpage.

- **Player ID Verification:** Upon successful navigation from the first page, the user's chosen Player ID, as established on the preceding webpage ("index.html"), will likely be displayed prominently on the "connect.html" page. This serves the purpose of verifying the accuracy of the selected identifier and potentially allows the user to make any necessary modifications before proceeding.
- **Opponent Selection Mechanism:** A fundamental element of the second page is a designated mechanism for selecting the opponent with whom the user desires to engage in a chess match. This mechanism may consist of a text input field where the user can enter the unique Player ID of their chosen opponent. Alternatively, the application might employ a matchmaking system, if implemented, to automatically pair the user with another available player of a similar skill level.

- **Connection Initiation Protocol:** Once the user has identified their opponent (either by manually entering their ID or through an automated matchmaking system), a designated button (potentially labeled "Connect" or "Challenge") will be available for them to initiate the peer-to-peer connection process. Clicking this button triggers a series of actions within the application, including utilizing the PeerJS library to establish a direct connection between the user's device and the opponent's device, bypassing the need for a central server.
- **Visual Feedback on Connection Status:** The design of the "connect.html" page should incorporate visual feedback mechanisms to inform the user about the current status of the connection attempt. This may involve the implementation of a progress bar, a changing icon, or informative text messages that update the user on the progress of establishing the connection.

⇨ **GameRoom UI for game play**

Figure 6 : Game Room UI page

The Chessboard Awaits: A Detailed Analysis of Remote Chess' Third Page

The third webpage within the Remote Chess application.embodies the core of the online chess experience. This page transforms from a static interface into a dynamic battleground where players engage in strategic chess matches. Let's explore the potential elements that bring this virtual chessboard to life.

- **Visually Appealing Chessboard:** The centerpiece of the page is a visually captivating chessboard rendered using the ChessboardJS library. This library empowers the application to translate the traditional chessboard layout into a digital format, allowing players to observe the game state and the consequences of their moves in real-time. The design should be clear and aesthetically pleasing, faithfully replicating the squares, markings, and chess pieces found on a physical chessboard.
- **Player Information Display:** The page should prominently display relevant information for both players. This may include usernames or Player IDs, potentially accompanied by visual indicators of the current player's turn. Additionally, the application might incorporate features to display elapsed time information or move history logs for reference during the game.
- **Move Execution Mechanism:** A crucial element of the page is the mechanism for players to execute their chess moves. This likely involves an intuitive interface that allows users to click on squares and drag pieces to their desired destinations on the virtual chessboard. The application, powered by the Chess.js library, should validate the legality

of each move according to the official rules of chess, ensuring fair and strategic gameplay.

- **Real-Time Game Updates:** The page should seamlessly update the chessboard after each valid move is executed by either player. This real-time update functionality, facilitated by the peer-to-peer connection established earlier, ensures both players have a synchronized view of the game state, fostering a dynamic and engaging chess experience.
- **Integrated Chat Functionality:** The application incorporates a chat window within the page, allowing players to communicate with each other during the match. This feature adds a social dimension to the online chess experience, enabling players to share strategies, offer friendly banter, or simply engage in casual conversation while competing.
- **In-Game Management Tools:** To enhance the user experience and cater to unforeseen circumstances, the page provides convenient in-game management tools. These tools include buttons for resigning gracefully if the game situation warrants it, offering a draw for a peaceful resolution, or initiating a rematch to challenge the opponent to another game.

⇨ **Game over Screen** (Fig 5)

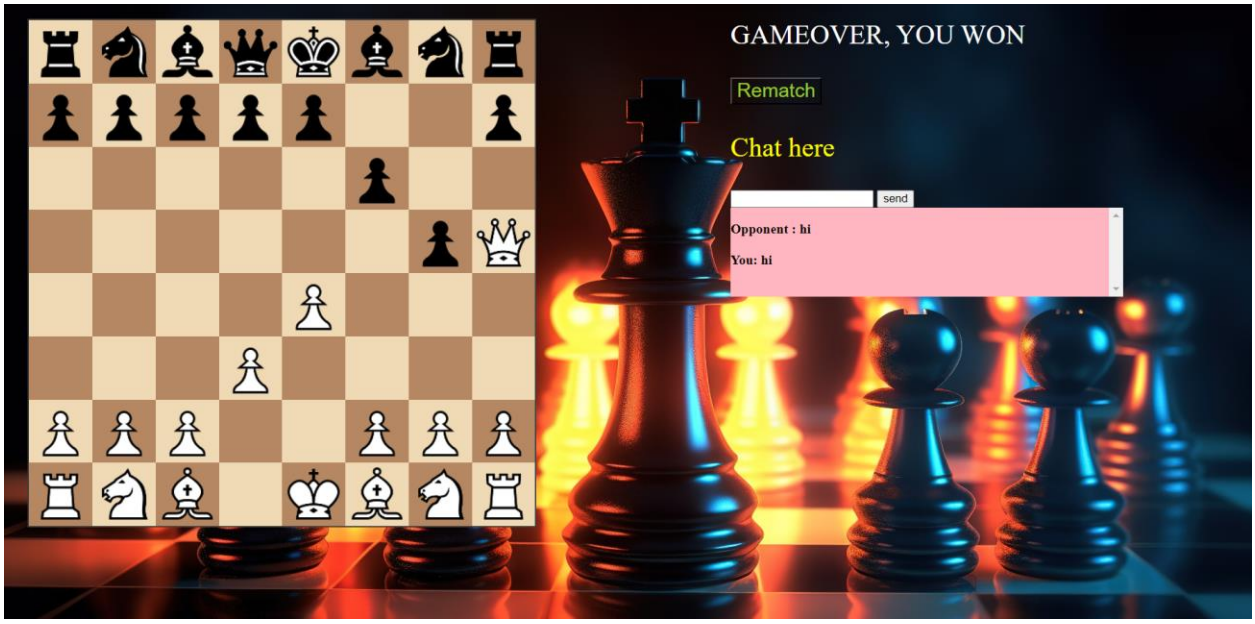


Fig 6 White's perpective

Fig 7 Black's perspective

The final webpage encountered within Remote Chess serves as the denouement of the online chess match. This concluding page reflects the outcome of the game and provides options for

players to move forward. Let's delve into the potential elements users might encounter on this concluding page.

- **Game Status Display:** The most prominent element of the final page should be a clear and unambiguous display of the game's outcome. This may involve a large text message declaring "Checkmate" for the victor or "Draw" if the game concludes in a stalemate. Additionally, visual cues, such as highlighting the checkmated king or displaying the final position on the chessboard, could be used to reinforce the game's conclusion.
- **Player Information Recap:** The page might reiterate relevant player information, potentially displaying usernames or Player IDs alongside the final result. This serves as a clear reminder of who participated in the match and reinforces the outcome for both players.
- **Post-Game Options:** The concluding page should provide players with clear options for their next course of action. This might include buttons for:
    - **Rematch:** A prominent button labeled "Rematch" allows the players to challenge each other to a new game, fostering a sense of camaraderie and potentially enabling a rematch series.

For designing UI we used chessboard.js opensource library and CSS, each piece is an individual image that is moved when a move is made .

⇨ **Chat Box Design**

Chat box is div , it's height expands upto max-height:30vh , after that scrolling is enabled so as to not make the chess board go out of view.

The following images shows these transitions of chat box :

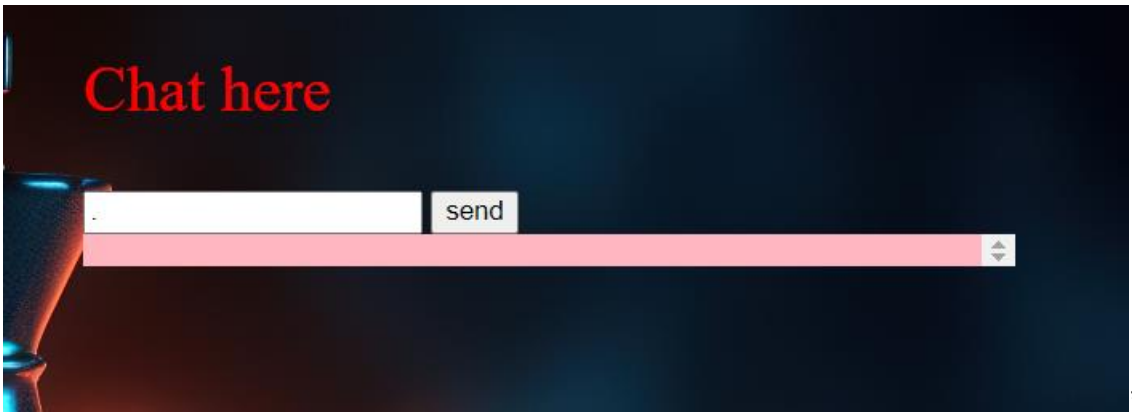


Fig 8 Initial Chat box

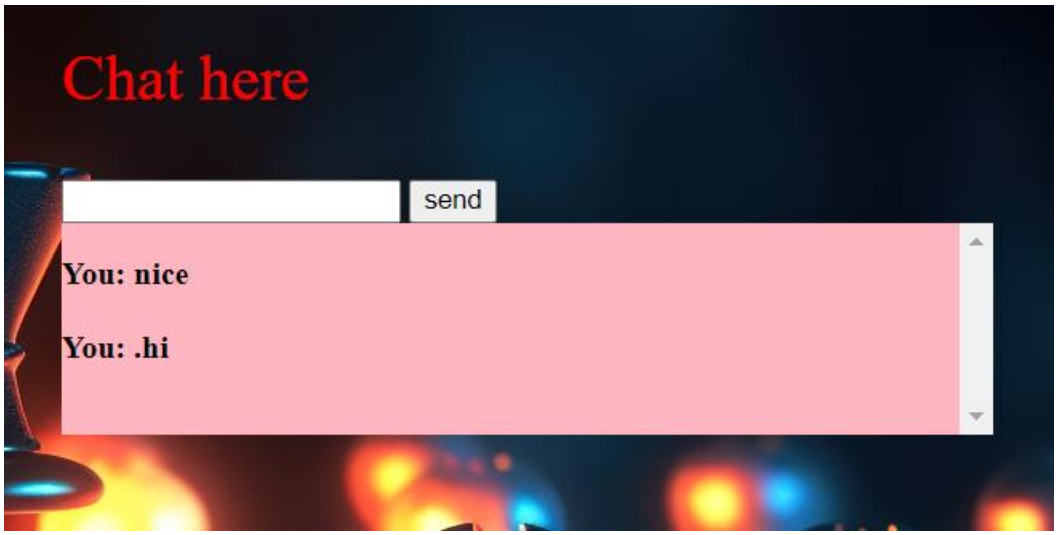


Fig 9 Expanding chat box

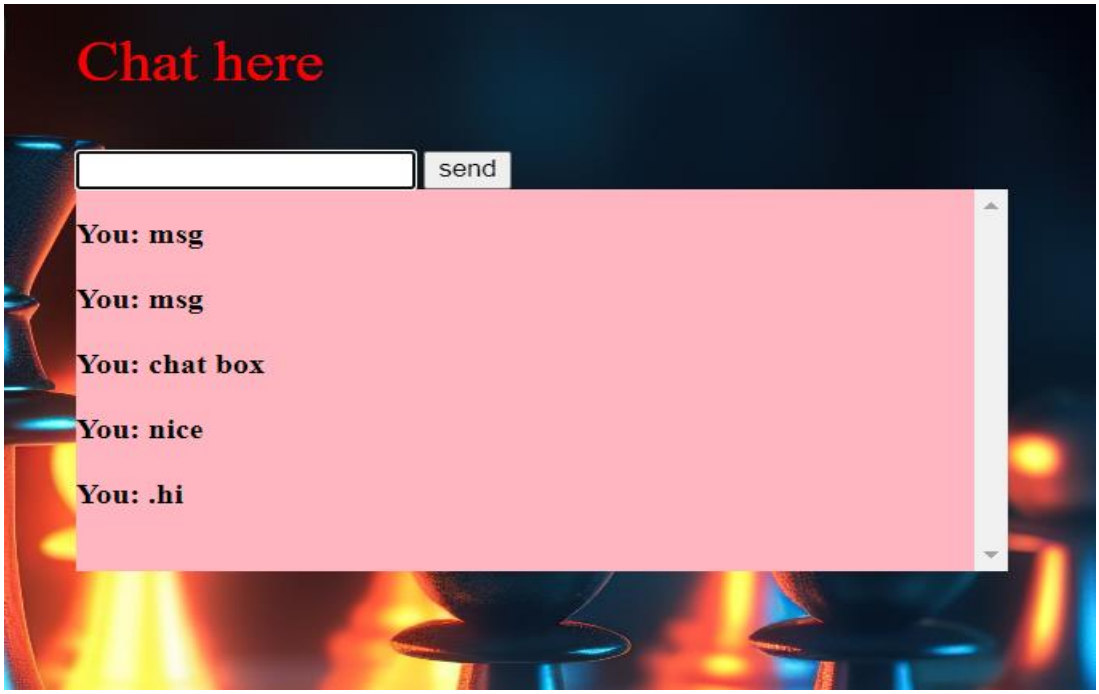


Fig 10 Expanding chat box

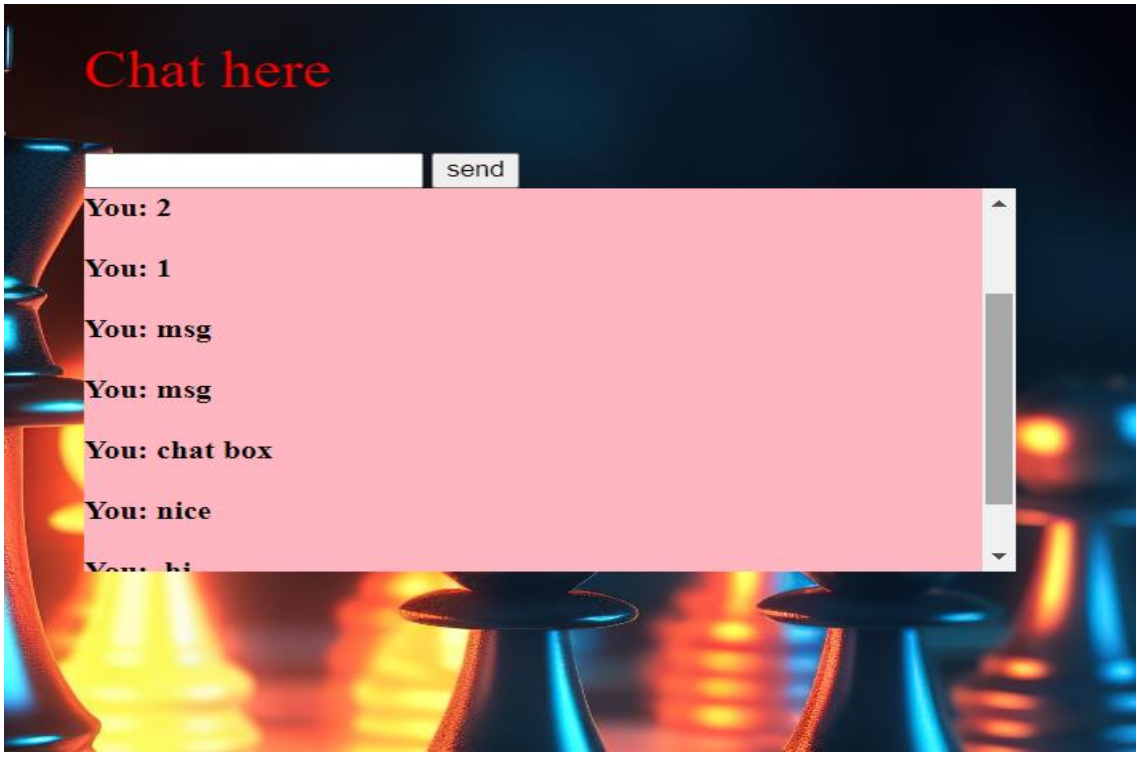


Fig 11 Scroll enabled

Scroll is enabled after chat box height becomes more than 30vh.

PeerJS connectivity in this Remote Chess project facilitates peer-to-peer communication between two players, enabling them to establish a direct connection for online chess gameplay. Here's a detailed breakdown of how PeerJS is used:

1. Peer Object Creation and ID Generation:

- `var peer = new Peer(makeid());`
    - This line creates a new `Peer` object using the PeerJS library. This object acts as the communication endpoint for the player.
    - `makeid()` (not shown in the code snippet) is a function likely responsible for generating a unique identifier for the player. This ID helps distinguish players during connection establishment.

2. Peer Connection Events:

- **`peer.on('open', function(id) { ... });`**: This defines a function that executes when a successful peer connection is established. It retrieves the player's assigned ID from PeerJS and displays it on the webpage.
- **`peer.on('connection', function(conn) { ... });`**: This defines a function that executes when another player connects to this peer. It retrieves information about the connected player stored in the `conn` object.

3. Initiating a Connection (`startConnect()` function):

- This function retrieves the opponent's ID from an input field (likely an HTML element).
- It uses the `peer.connect(opponentID)` method to initiate a connection with the player identified by the `opponentID`.
- If the connection is successful, the `peer.on('connection')` function will be triggered, establishing communication between the players.

4. Data Exchange and Message Handling:

- Once connected, players can exchange data using PeerJS methods. The provided code snippet likely uses the following functionalities:
    - Sending messages: A function like `sendMsg()` might be used to send chat messages or game updates (move information) to the opponent. It would use `conn.send(message)` to send data over the established connection.
    - Receiving messages: The `handleRec()` function listens for incoming data from the opponent using `conn.on('data', function(data) { ... });`. This function then parses the received data and performs actions based on its content. It might handle different message types like chat messages, move updates, opponent actions (resign, draw offer), and rematches.

5. Overall PeerJS Role:

- PeerJS acts as a facilitator for peer-to-peer communication. It takes care of the following:
    - Establishing direct connections between players.
    - Handling connection negotiation and signaling (exchanging information for establishing the connection).
    - Providing reliable data channels for message exchange between connected players.

Implementation

4.1 Brief

This project is implemented using HTML, CSS, JAVASCRIPT .

Javascript libraries used in this project are PeerJS , Chess.js, chessboard.js , we discussed about them in introduction.

4.2 Source Code and Explanation

Below is the source code with explanations :

HTML CODE:

Some portion of the HTML code will be introduced dynamically in JS code , the below code is for the index.html file and then followed by connect.html , which acts as home page for our website, It imports and links required modules ,it display's your peerID and also provides an input box for entering another peer's ID with an connect button.

When the connect button is pressed or clicked it triggers the start connection in JS file.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>remote chess</title>
    <link rel="stylesheet" href="style.css">
</head>
<body id="pagee">
    <p style="color: rgb(231, 234, 229); font-size: larger; ">
Welcome to the two player online chess , <a
style="color:yellow" href="help.html">facing problem?</a>
```

```
    </p>
    <script>
    function makeID(){
        let idInBox = document.getElementById("myID").value;
        if(idInBox.length == 0 ) return ;
        localStorage.setItem("myID", idInBox);
        window.location.href = "connect.html";
    }
    </script>
    <p style="color:white">
        Enter Your PlayerID :
        <input type="text" id="myID">
        <button style = "background-color: skyblue;"
id="LoginButton" onclick="makeID()">
            Go Online
        </button>
        <script>

document.getElementById("myID").addEventListener("keypress",
function(event) {
            if (event.key === "Enter") {
            event.preventDefault();
            document.getElementById("LoginButton").click();
            }
            });
        </script>
    </p>


</body>
</html>
```

Connect.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>remote chess</title>
    <link rel="stylesheet" href="style.css">
    <link rel="stylesheet"

href="https://unpkg.com/@chrisoakman/chessboardjs@1.0.0/dist/ch
essboard-1.0.0.min.css"
    integrity="sha384-
q94+BZtLrkL1/ohfjR8c6L+A6qzNH9R2hBLwyoAfu3i/WCvQjzL2RQJ3uNHDISd
U"
    crossorigin="anonymous">
    <script
src="https://unpkg.com/peerjs@1.5.2/dist/peerjs.min.js"></scrip
t>
    <script src="https://code.jquery.com/jquery-3.5.1.min.js"
    integrity="sha384-
ZvpUoO/+PpLXR1lu4jmpXWu80pZlYUAfxl5NsBMWOEPSjUn/6Z/hRTt8+pR6L4N
2"
    crossorigin="anonymous"></script>

<script
src="https://unpkg.com/@chrisoakman/chessboardjs@1.0.0/dist/che
ssboard-1.0.0.min.js"
    integrity="sha384-
8Vi8VHwn3vjQ9eUHUxex3JSN/NFqUg3QbPyX8kWyb93+8AC/pPWTzj+nHtbC5bx
D"
    crossorigin="anonymous"></script>
```

```
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/chess.js/0.10.2/che
ss.js" integrity="sha384-
s3XgLpvmHyscVpijnseAmye819Ee3yaGa8NxstkJVyA6nuDFjt59u1QvuEl/mec
z" crossorigin="anonymous"></script>
    <script src="connecter.js"></script>


</head>
<body id="pagee">
   <p style="color: rgb(231, 234, 229); font-size: larger; ">
Welcome to the two player online chess , <a
style="color:yellow" href="help.html">facing problem?</a>
</p>

    <p style="color:orange ; " id="playerID">
     Your player ID :

    </p>
    <p style="color:white" >
        Enter Player ID of opponent to connect:
        <input type="text" id="toConnectID">
        <button style = "background-color: skyblue;"
id="connectButton" onclick="startConnect()">
            connect
        </button>

    </p>
    <p style="color: red; font-size: 150%;">
        NOTE : The person who enters opponent ID gets White,
then Colors are swapped after each game.
    </p>

 <script>
```

```
 document.getElementById("toConnectID").addEventListener("keypr
ess", function(event) {
    if (event.key === "Enter") {
      event.preventDefault();
      document.getElementById("connectButton").click();
    }
  });
   </script>
</body>
</html>
```

CSS CODE:

The CSS code is there for assigning design properties to various elements present in our UI.

```
#queenButton{
 background-color: white;
 color : green;
}
#rookButton{
    background-color: black;
    color : green;
}
#knightButton{
    background-color: black;
    color : green;
}
#bishopButton{
    background-color: black;
    color : green;
}
#pawnPromotionOptions{
    color: red;
    position:relative;

}
#msgBoxContainer{
    width: 60vh;
    max-height:30vh;
    overflow-y: scroll;
    background-color:  lightpink;

}
#msgBox{
    width: 50vh;
```

```
    color: black;
    font-weight: bolder;
}
#pagee{

    background-image: url("img/connectPaper.jpg");
    height: 100vh;
    overflow:hidden;
    background-repeat: no-repeat;
    background-size: cover;
    background-color: black;
}
#myBoard{
    width: 78vh;
    position: relative;
    margin-right: 15%;
    float:left;
    padding-left:1%;
    margin-left:2vh;
}
#resignButton{

 margin-top:1vh;
 background-color: transparent;
 color : red;
 font-size:3vh;
}
#drawButton{
    margin-top: 1vh;
    margin-left: 4vh;
    background-color: transparent;
    color:green;
    font-size:3vh;
}
#acceptDrawButton{
```

```
    margin-top: 1vh;
    margin-left: 4vh;
    background-color: transparent;
    color:green;
    font-size:3vh;
    visibility: hidden;
}
#rematchButton{
    background-color: transparent;
 color : yellowgreen;
 font-size:3vh;
}
.gameoverpara{
    color:white;
    font-size: xx-large;
}
```

JavaScript CODE and Explanation:

Detailed Explanation of the Remote Chess JavaScript Code

The provided JavaScript code implements the core functionalities of the Remote Chess application, enabling online chess gameplay with features like peer-to-peer connections, move validation, chat, and user interaction. Here's a breakdown of the code step-by-step:

1. Initialization and Peer-to-Peer Connection

- **var game = new Chess();**: This line initializes a new Chess object using the Chess.js library. This object manages the game state, including the board layout, piece positions, and turn information.
- **var peer = new Peer(makeid());**: This line creates a new Peer object using the PeerJS library. PeerJS facilitates peer-to-peer connections between players, allowing them to communicate directly without a central server.
- **peer.on('open', function(id) { ... });**: This defines a function that executes when a successful peer connection is established. It retrieves the player's unique ID assigned by PeerJS and displays it on the webpage (likely using HTML elements).
- **makeid()**: This function (not shown in the provided code) is likely responsible for generating the player's ID. It might use a combination of random characters and numbers to create a unique identifier.
- **startConnect()**: This function handles initiating a connection with another player. It retrieves the opponent's ID from an input field (likely an HTML element) and uses the Peer object to connect to that specific player.
- **peer.on('connection', function(conn) { ... });**: This defines a function that executes when another player connects to this peer. It retrieves information about the connected player (stored in the conn object) and sets the opponent's ID. It also determines the player's color (white or black) based on the connection order and launches the game interface using the gameRoomLaunch() function.

2. Game Interface and User Interaction

- **gameRoomLaunch()**: This function sets up the visual elements for the chessboard, chat functionality, and other game controls. It likely interacts with HTML elements to create the following:
    - A chessboard using the ChessboardJS library, which allows for drag-and-drop or click-to-move interactions.
    - Display areas for player IDs to show who they are playing against.
    - Buttons for resigning, offering a draw, and sending messages in the chat.
    - A chat box to display messages from both players.
    - It also adds event listeners for user interactions with these elements (covered later).
- **Square Interaction:**

    - **squareClicked()**: This function handles a player clicking on a chess square. It checks if the clicked piece belongs to the current player (based on game state) and attempts to make a move using the Chess.js library's move() function.
    - **onDragStart()**: This function activates when a player starts dragging a chess piece. It validates if the move is legal based on the current turn and piece color using the Chess.js library's functionalities.
    - **onDrop()**: This function handles a piece being dropped on a new square. It calls the makeMove() function (explained below) to process the move and update the game state.
- **makeMove()**: This function attempts to make the move selected by the player using the Chess.js library's move() function. It performs the following actions:
    - Validates the move based on the game state and piece rules.
    - Handles promotions (pawn reaching the other side) by allowing the player to choose a promotion piece (queen, rook, bishop, or knight).
    - Updates the board visually using the ChessboardJS library to reflect the new piece positions.
    - Plays sound effects for move confirmation (using Javascript's audio capabilities).
    - Checks for check or checkmate conditions using Chess.js functions.
    - If checkmate is detected, the game ends, and the GameOverCheck() function is called.
- **GameOverCheck()**: This function checks if the game is over due to checkmate or stalemate usingChess.js` functions. If so, it displays a game-over message (likely using HTML elements) and handles cleanup based on the winner.

3. Chat Functionality:

- **sendMsg()**: This function is triggered when the user clicks the "send" button in the chat window. It retrieves the message content from an input field and sends it to the opponent using the PeerJS connection.
- **handleRec()**: This function listens for incoming data from the opponent using PeerJS. It handles different message types based on the first character

```
var game = new Chess();

function makeid() {
 return localStorage.getItem("myID");
}

var peer = new Peer(makeid());
var Conn;
```

```
let me;
let him;
let myCol='white';
var lastSquareClicked;
var lastSquareHovered;
//let toSend=false;
peer.on('open', function(id) {
    //alert("hello")
    let playerID =  document.getElementById("playerID");
    playerID.append(id);
    me=id;
  });

function startConnect(){
  let IDinbox = document.getElementById("toConnectID").value;
  Conn = peer.connect(IDinbox);
  him=IDinbox;
    gameRoomLaunch();
    //connection to other side
    handleRec();

}
 //connection from other side
 peer.on('connection', function(conn) {
       him=conn.peer;
       Conn = conn;
       myCol='black';
       gameRoomLaunch();
       handleRec();

  });
// reconnect
 peer.on('disconnected',() => {
  peer.reconnect();
 });
```

```
function rematch(){
  //alternate colors after each game
  game = new Chess();
  if(myCol == 'white')  myCol =  'black';
  else myCol = 'white';
  gameRoomLaunch();
  Conn.send('p');

}
function gameRoomLaunch(){
  var promotionOptionsHtml = " <div id='pawnPromotionOptions'>
Pawn Promtion Choice : <button id='queenButton'
 onClick='changePromotionChoiceToQueen()'> ♕ </button> <button
id='rookButton' onClick='changePromotionChoiceToRook()'> ♖
</button> <button id='bishopButton'
 onClick='changePromotionChoiceToBishop()'> ♗ </button>
<button id='knightButton'
 onClick='changePromotionChoiceToKnight()'> ♘
</button></div>";
  var cont=document.getElementById("pagee");
  cont.innerHTML="  <p style = 'color: skyblue'>HELLO,  welcome
to game room "+promotionOptionsHtml+" </p><p
style='color:orange ' id='yourID'>yourID :   </p><p
style='color:orange' id='oppID' > opponentID: </p><div
id='myBoard' onClick='squareClicked()'></div>";
  var yourID=document.getElementById("yourID");
  yourID.append(me);
  var oppID=document.getElementById("oppID");
  oppID.append(him);
  var resignButtonHtml = "<button id='resignButton'
onclick='resign()'> Resign🏳 </button>";
  var drawButtonHtml   = "<button id='drawButton'
onclick='drawOffer()'> Draw🤝 </button>";
  var acceptDrawButtonHtml = "<button id='acceptDrawButton'
onclick='acceptDraw()'> Accept Draw </button>";
```

```
  cont.innerHTML += resignButtonHtml + drawButtonHtml +
acceptDrawButtonHtml +"<div id='msgContainerDiv'><p style =
'color: yellow; font-size: 200% '> Chat here  </p> <input
type='text' id='msg'>  <button
id='sendButton'onclick='sendMsg()'> send </button> <div
id='msgBoxContainer'><p id='msgBox'> </p> </div> </div>";
  launchBoard();
  document.getElementById("msg").addEventListener("keypress",
function(event) {
    if (event.key === "Enter") {
      event.preventDefault();
      document.getElementById("sendButton").click();
    }
  });
}
function drawOffer(){
  addMsg("You: draw Offered");
  Conn.send('d');
}
function destroyDrawBox(){
  document.getElementById("acceptDrawButton").style.visibility
= "hidden";
}
function ActivateDrawBox(){
  document.getElementById("acceptDrawButton").style.visibility
= "visible";
}
function acceptDraw(){
   addMsg("You : draw Accepted")
   Game_end('d');
   Conn.send('a');
}
function resign(){
  //alert();
  let winner = (myCol == 'white') ? 'b' : 'white';
```

```
  Conn.send('r');
  Game_end(winner);
}
var Board;
function launchBoard(){

  var config={
    draggable: true,
    position: 'start',
    onDragStart: onDragStart,
    onDrop: onDrop,
    onMouseoverSquare: onMouseoverSquare,
    onSnapEnd : onSnapEnd,
    showNotation : false,
    orientation : myCol
  }
  Board = Chessboard('myBoard',config);

}
//drag to move
function onDragStart (source, piece, position, orientation) {
  // do not pick up pieces if the game is over
  if (game.game_over()) return false;
  // only pick up pieces for the side to move and respective
color
  if(Board.orientation() == 'white' && (piece.search(/^b/) !==
-1 || game.turn() === 'b'))
        return false;
  if(Board.orientation() == 'black' && (piece.search(/^w/) !==
-1 || game.turn() === 'w'))
        return false;
  lastSquareClicked=source;
}

function onDrop (source, target) {
```

```
  makeMove(source,target);
}
var promotionChoice='q';
function changePromotionChoiceToQueen(){
  document.getElementById("queenButton").style.backgroundColor
= "white";
  document.getElementById("rookButton").style.backgroundColor =
"black";
  document.getElementById("bishopButton").style.backgroundColor
= "black";
  document.getElementById("knightButton").style.backgroundColor
= "black";
  promotionChoice = 'q';
}
function changePromotionChoiceToRook(){
  document.getElementById("rookButton").style.backgroundColor =
"white";
  document.getElementById("queenButton").style.backgroundColor
= "black";
  document.getElementById("bishopButton").style.backgroundColor
= "black";
  document.getElementById("knightButton").style.backgroundColor
= "black";
 promotionChoice = 'r';
}
function changePromotionChoiceToBishop(){
  document.getElementById("bishopButton").style.backgroundColor
= "white";
  document.getElementById("rookButton").style.backgroundColor =
"black";
  document.getElementById("queenButton").style.backgroundColor
= "black";
  document.getElementById("knightButton").style.backgroundColor
= "black";
  promotionChoice = 'b';
```

```
}
function changePromotionChoiceToKnight(){
  document.getElementById("knightButton").style.backgroundColor
= "white";
  document.getElementById("rookButton").style.backgroundColor =
"black";
  document.getElementById("bishopButton").style.backgroundColor
= "black";
  document.getElementById("queenButton").style.backgroundColor
= "black";
  promotionChoice = 'n';
}

function getPromotion(){
  return promotionChoice;
}
function makeMove(source,target){
 // see if the move is legal
 var move = game.move({
  from: source,
  to: target,
  promotion: getPromotion()
})
// illegal move
if (move === null) {
  if(source !== target)
  lastSquareClicked="";
  return 'snapback';
}
// send move if legal
destroyDrawBox();
let moveS=source+target;
Conn.send('m'+ moveS + getPromotion());
playMoveSound();
removecolorSquares();
```

```
colorSquare(source);
colorSquare(target);
if(game.in_check()) {
  let targetColor='b';
  if(Board.orientation() == 'black')
  targetColor='w';
   colorKing(targetColor);
 }
}
 //update board , because in moves like castling chessboardjs
needs support from chess js for proper update

function onSnapEnd(){
  //first updating the position then check game over , order
matters otherwise gameovercheck might set a game over board and
then board.position again set it .. causing irregular display
 Board.position(game.fen());
 GameOverCheck();
}
// end of drag to move

//click to move

function onMouseoverSquare (square, piece) {
  lastSquareHovered=square;
}
function squareClicked(){
 var pc = game.get(lastSquareClicked);

 if(pc.color != Board.orientation().charAt(0) ) return ;
 let snapped = makeMove(lastSquareClicked,lastSquareHovered);
 lastSquareClicked=lastSquareHovered;
 //update board
 if(snapped !== 'snapback'){
```

```
  //first update the position then check game over , order
matters otherwise gameovercheck might set a game over board and
then board.position again set it .. causing irregular display
  Board.position(game.fen());
  lastSquareClicked = "";
  GameOverCheck();
}
}
//end of click to move
function GameOverCheck(){
  if(game.game_over()){
    let winner;
     if(game.in_checkmate() == true){
        if(game.turn() == 'b')
         {
              winner='white';
         }
         else {
          winner='b';
         }
     }
     else{
      winner='d';
     }
     Game_end(winner);
   }
}

function sendMsg(){
  var msg=document.getElementById("msg");
  if(msg.value != ""){
  Conn.send('c'+msg.value);
  addMsg("You: " + msg.value);
  msg.value="";
}
```

```
}
//this function is required because Conn obj is defined only
after the connection gets established.
//also it reduces lines of code, we are avoiding writing it
above twice !. moreover Conn data can only be recieved after
connecting
// so it's just better to make this peice of code come into
picture when it is required.
function handleRec(){
 Conn.on('open',function(){
   Conn.on('data', function(data) {
    // alert(data);
    if(data.charAt(0) =='c'){
    data = data.slice(1);
    addMsg("Opponent : " + data);
    playChatMsgSound();
   }
   else if(data.charAt(0) == 'p'){
    //alternate colors after each game
    game = new Chess();
    if(myCol == 'white')  myCol =  'black';
    else myCol = 'white';
    gameRoomLaunch();
   }
   else if(data.charAt(0) == 'a'){
    addMsg("Opponent : draw Accepted")
    Game_end('d');
   }
   else if(data.charAt(0) == 'd'){
     addMsg("Opponent: draw Offer");
     playChatMsgSound();
     ActivateDrawBox();
   }
   else if(data.charAt(0) == 'r'){
    let winner = myCol;
```

```
    Game_end(winner,true);
   }
   else {
     playMoveSound();
     data = data.slice(1);
     var promotedTo = ''+ data.charAt(4);
    game.move({
      from: data.charAt(0)+data.charAt(1),
      to: data.charAt(2)+data.charAt(3),
      promotion: promotedTo
    })
       Board.position(game.fen());
       removecolorSquares();
       colorSquare(data.charAt(0)+data.charAt(1));
       colorSquare(data.charAt(2)+data.charAt(3));
       if(game.in_check()) {
        let targetColor='w';
        if(Board.orientation() == 'black')
        targetColor='b';
         colorKing(targetColor);
       }
       GameOverCheck();
   }
   });
 });

}

function addMsg(msg){
  var cont=document.getElementById("msgBox");
  cont.innerHTML=msg+"<br><br>"+cont.innerHTML;
}
function Game_end(winner,byResign=false){
  playGameOverSound();
  var cont=document.getElementById("pagee");
```

```
  var msgContainerDiv =
document.getElementById("msgContainerDiv").innerHTML;
  cont.innerHTML="";
  cont.innerHTML += "<div id='myBoard'></div>";
  var config={
    draggable: true,
    position: Board.position(),
    onDragStart: onDragStart,
    onDrop: onDrop,
    onSnapEnd : onSnapEnd,
    showNotation : false,
    orientation : Board.orientation()
  }
  Board = Chessboard('myBoard',config);
   if(winner == Board.orientation())
    cont.innerHTML += "<p class='gameoverpara'> GAMEOVER, YOU
WON </p>";
   else if(winner == 'd')
    cont.innerHTML += "<p class='gameoverpara'> GAMEOVER, IT'S
A DRAW </p>";
   else
    cont.innerHTML += "<p class='gameoverpara'> GAMEOVER, YOU
LOSE </p>";

    if(byResign == true)
     cont.innerHTML += "<p class='gameoverpara' >Opponent
resigned </p>";
     var rematchButtonHtml = "<button id='rematchButton'
onClick = 'rematch()'> Rematch </button>"
     cont.innerHTML += rematchButtonHtml;
     cont.innerHTML += msgContainerDiv;

    document.getElementById("msg").addEventListener("keypress",
function(event) {
      if (event.key === "Enter") {
```

```
        event.preventDefault();
        document.getElementById("sendButton").click();
      }
    });
}
function playMoveSound() {
  var audio = new
Audio('https://images.chesscomfiles.com/chess-
themes/sounds/_MP3_/default/move-self.mp3');
  audio.play();
}
function playGameOverSound(){
  var audio = new
Audio('https://images.chesscomfiles.com/chess-
themes/sounds/_MP3_/default/game-end.mp3');
  audio.play();
}
function playChatMsgSound(){
  var audio = new
Audio('https://images.chesscomfiles.com/chess-
themes/sounds/_MP3_/default/notify.mp3');
  audio.play();
}

var whiteSquarecolor = '#8d8ce6'
var blackSquarecolor = '#8d8ce6'
var checkSquarecolor = '#b30727'
function removecolorSquares () {
  $('#myBoard .square-55d63').css('background', '')
}

function colorSquare (square,checkSquare=false) {
  var $square = $('#myBoard .square-' + square)

  var background = whiteSquarecolor
```

```
  if ($square.hasClass('black-3c85d')) {
    background = blackSquarecolor
  }
  //color king square red
  if(checkSquare == true){
    background = checkSquarecolor;
  }
  $square.css('background', background)
}
// called when king is in check
function colorKing(targetColor){
  var toSearch = 'k'.charCodeAt(0);
  if(targetColor == 'w')
      toSearch = 'K'.charCodeAt(0);
  var fen = game.fen();
  var kingSquare = findKingSquare(fen,toSearch)
  colorSquare(kingSquare,true);
}
// calculates king's square
function findKingSquare(fen,toSearch){
  var num = 0;
for (let i = 0; i < fen.length; i++) {
var ch = fen.charCodeAt(i);
if(ch == toSearch) break;
    if(ch == '/'.charCodeAt(0)) continue;
if(ch >= '1'.charCodeAt(0) && ch <= '8'.charCodeAt(0))
  num += (ch - '0'.charCodeAt(0));
else
 num += 1;
}
var file = String.fromCharCode('a'.charCodeAt(0) + num % 8 );
var rank = 8 - Math.floor(num / 8);
 return (file+rank);
}
```

```
// messages format
/*
m = move , c = chat , r = resign , d = draw offer , a = accept
draw , p = rematch
*/
```

More Insights :

The concept of symmetrical nature in peer-to-peer (P2P) code doesn't directly translate to the code itself, but rather to the way peers interact within the P2P network. Here's a breakdown of the concept:

Symmetry in P2P Networks:

- In P2P networks, both peers act as equals. There's no central server dictating communication or roles.
- Each peer has the capability to initiate connections, send data, and receive data from other peers.
- The code running on each peer can be largely similar, reflecting this symmetrical behavior.

Code Structure and Symmetry:

- The code for P2P applications often involves functions for:
    - Establishing connections (both initiating and accepting)
    - Sending data (messages, game updates, etc.)
    - Receiving data and processing it
- These functionalities might be implemented in a similar way on both peers, with minor variations depending on the specific application logic.

Example: Remote Chess with PeerJS

In the Remote Chess example using PeerJS, the code on both players' machines might share similar structures for:

- **Connection Handling:** Both players use PeerJS functions like `peer.on('open')` and `peer.on('connection')` to handle connection establishment and incoming connections.
- **Data Exchange:** Both players might have functions like `sendData(data)` to send messages or game updates to the connected peer.
- **Game Logic:** While the specifics of game logic (move validation, board updates) might reside in a separate library (Chess.js), both players would interact with this library in a similar way.

Breaking Symmetry (Sometimes Necessary):

- In some P2P applications, there might be a slight asymmetry, like a "host" and a "client" role. The code on these peers might have some differences to reflect their specific roles.
- However, the overall philosophy is to minimize asymmetry for simpler development and maintenance.

Key Takeaways:

- Peer-to-peer code doesn't have perfect code symmetry line by line, but both peers often have similar functionalities for connection, data exchange, and potentially game logic interaction.
- The symmetrical nature reflects the equal roles of peers in a P2P network.

The draw and resign functionalities in the provided code are implemented through buttons that trigger message sending via PeerJS:

1. Draw Offer:

- Function: `drawOffer()`
    - Sends a message with prefix `'d'` to the opponent using `Conn.send('d')`.
    - Updates the chat window with a message indicating the draw offer ("You: draw Offered").
    - Calls the `destroyDrawBox()` function (explained later).

2. Resign:

- Function: `resign()`
    - Determines the winner based on the current player's color (`myCol`).
    - Sends a message with prefix `'r'` and the winner color (`b` or `white`) to the opponent using `Conn.send('r'+winner)`.
    - Calls the `Game_end(winner)` function to display the game over screen with the winner information.

3. Accepting Draw Offer:

- Function: `acceptDraw()`
    - Sends a message with prefix `'a'` to the opponent using `Conn.send('a')`.
    - Updates the chat window with a message indicating draw acceptance ("You: draw Accepted").
    - Calls the `Game_end('d')` function to display the game over screen with a draw result.

4. Handling Draw Offers from Opponent:

- Function: `handleRec()` (inside `peer.on('open')`)
    - Listens for incoming data on the PeerJS connection.
    - If the data starts with `'d'`, it signifies a draw offer from the opponent.
    - Extracts the remaining message content (ignored in this case).
    - Updates the chat window with a message indicating the opponent's draw offer ("Opponent: draw Offer").
    - Calls the `ActivateDrawBox()` function to display a button allowing the user to accept the draw.

5. Draw Offer UI Elements:

- Function: `gameRoomLaunch()`
    - Creates an HTML button element with ID "drawButton" and onclick event set to `drawOffer()`.

6. Accepting Draw UI Element:

- Function: `gameRoomLaunch()`
    - Creates an initially hidden HTML button element with ID "acceptDrawButton" and onclick event set to `acceptDraw()`.
- Function: `ActivateDrawBox()`
    - Makes the "acceptDrawButton" visible, allowing the user to accept the draw offer.
- Function: `destroyDrawBox()`
    - Hides the "acceptDrawButton" after the user sends their own draw offer or the game progresses.

The promotion functionality in the provided code is implemented through a combination of user selection, message sending, and game logic updates:

1. User Promotion Choice:

- Four buttons are created for Queen (`queenButton`), Rook (`rookButton`), Bishop (`bishopButton`), and Knight (`knightButton`).
- Each button has an onclick event handler that calls a corresponding function:
    - `changePromotionChoiceToQueen()`: Sets the internal `promotionChoice` variable to 'q' and visually highlights the Queen button.
    - `changePromotionChoiceToRook()`: Sets `promotionChoice` to 'r' and highlights the Rook button (similar logic for Bishop and Knight).

2. Sending Promotion Information:

- The `makeMove()` function handles both click-to-move and drag-to-move actions.
- When a legal move is made involving a pawn reaching the last rank (promotion possible), this function includes promotion information in the message sent to the opponent via PeerJS.
- It calls the `getPromotion()` function to retrieve the current user's chosen promotion piece (Queen, Rook, Bishop, or Knight).
- The message format for moves includes the source square, target square, and promotion piece appended together (e.g., `'e7e8q'` for pawn on e7 promoting to Queen on e8).

3. Receiving and Applying Promotion:

- The `handleRec()` function listens for incoming data on the PeerJS connection.
- If the data starts with `'m'`, it signifies a move from the opponent.
- The data contains the source square, target square, and an optional promotion character at the end (e.g., `'d7d8r'`).
- The `game.move()` function is used to update the game state, including the promotion piece specified in the message.
- The board position is then updated using `Board.position(game.fen())`.

4. Visual Cues:

- The `colorSquare()` function is used to highlight the source and target squares of the move.

Overall, the code allows users to select their desired promotion piece, sends this information with the move data, and applies the promotion on the opponent's board and locally.

Here are some additional points to note:

- The initial value of `promotionChoice` is set to 'q' (Queen) by default.
- The code includes checks to ensure promotion is only offered when a pawn reaches the last rank.

Result and Analysis

All the basic Funtionalities required to play the such as Logical ascpects and UI are fully functional and running.

Connection between peers is working smoothly with lose of any data, and the latency is very low.

The additional features has been added without disrupting currently functional features. The design of the code is flexible to changes and allows ease in implementing new features.

We have also taken reviews for few candidates who are chess players themselves, and we got many positive reviews with some suggestions , we have implemented them in our project.

These reviews played an impoertant role for deciding the set of features that we implemented in second phase of our project , they are key part of this project.

We have hosted our project at https://remotechess.netlify.app , you can experience the gameplay here . The link is alone sufficient and self-explanatory of the results achieved.

Also one interesting point to be noted is that in peer to peer connection we only need to write single JS file, and this symmetry of code makes it very elegant, both peers are independent and have the same JS file code running, and hence we do not need to write separate code for client and server, as each peer is itself both. Peer to peer reduces lines of codes.

Timelines

- **January 2024 - Project Initiation and Research:**

  13th Jan 2024 - Define project scope and objectives.
  20th Jan 2024 – Exploring various implementation options and technologies

- **February 2024 - Development:**

  10th Feb 2024 – Development start.
  24th Feb 2024 – Basic Functionalities implemented.

- **March 2024 – Documentation and refinement:**

  9th Mar 2024 – improved functionalities and UI.
  15th Mar 2024 - Start project documentation.
  20th Mar 2024 – Implementation Plans for additional feauters

- **April 2024 – Second Phase of development:**

  10th Apr 2024 – implementation of addition features started
  30th Apr 2024 – additional features implementation finished along with issue fixes

- **May 2024 – Making of final report:**

  1-10 May 2024 – Final report writing and analysis.

References

- **Project Hosted on : https://remotechess.netlify.app**
- **ChessJS library : https://github.com/jhlywa/chess.js/blob/master/README.md**
- **ChessBoard JS library : https://chessboardjs.com/docs**
- **Github : https://github.com/khemsinghbhati/chess-two-players-online**
- **PeerJS : https://peerjs.com/docs**