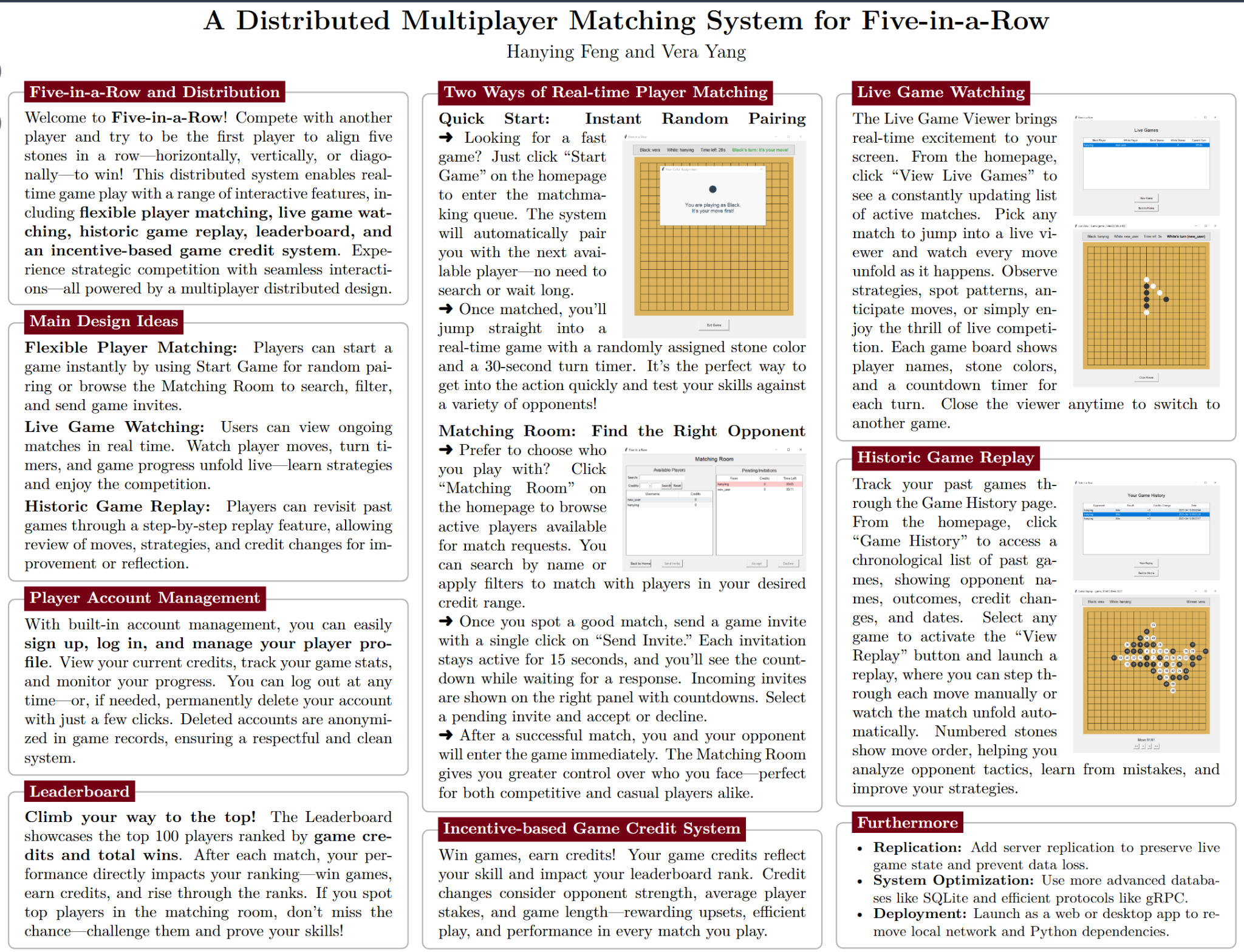
CS262 Final Project Engineering Notebook

– A Distributed Multiplayer System for Five-in-a-Row

Hanying Feng and Vera Yang

**Link to repo:** <https://github.com/verayang01/CS262-final-project>

**Poster:**



## Introduction

Five-in-a-Row, also called Gomoku, is a classic strategy board game where two players compete to place five of their stones–black or white–in a consecutive row (horizontally, vertically, or diagonally) on a 19x19 grid. This project builds a distributed multiplayer system for playing Five-in-a-Row online, featuring the following functionalities:

* User account creation and authentication (login/signup)
* Real-time gameplay with forced game rules, turn timers, and smooth disconnection handling
* Matching room for player searching, game invites, and private matching
* Live game viewer mode
* Game history and historic game replays
* Player statistics maintenance and leaderboard ranking
* User account management
* Incentive-based game credits and ranking system

We developed the entire system using Python and designed and implemented the client GUI using TKinter. Throughout this project, we utilized knowledge learned from class, such as TCP socket connections, client and server communication, wire protocols, and error handling. We have found that the real-time nature of the online pairing and game playing posed new challenges to the development of this application, and we found it rewarding to find solutions to these challenges by building up the application step-by-step. In this document, we summarize our design specifications, implementation details, testing, and our thought processes to present how we put these building blocks together.

## Design Specification:

### Authentication

1. **Functionality Details and User Logic**

When the user first runs the code, a login page will appear. The login page contains two fields that allow the user to enter a username and password. When the user clicks the “Login” button, the entered information will be sent to the server for authentication, and the user can log in successfully if the credentials are correct. The user needs to sign up before logging into an account with a new username. Each user can only be logged in once, and they must log out before logging in from another client.

When the user clicks the “Sign Up” button, they will be directed to the sign-up page. This page also contains two fields that allow the user to enter a username and password. The username must not duplicate any existing signed-up usernames, and both the username and password must not be empty or contain any spaces or newline characters. After entering a valid username and password, clicking the “Sign Up” button will create the account and redirect the user to the login page so they can log in and play the game. There is also a “Back to Login” button at the bottom that allows the user to return to the login page without creating a new account.

1. **Development Process and Things We Learned**

* We initially did not have a sign-up function, and users could create a new account simply by entering a new username and clicking the “Login” button on the login page. Later, we decided it would be clearer and more organized to require users to sign up before logging in. As a result, we added a sign-up page to allow users to create new accounts.
* In the beginning, the only restrictions on the username and password were that the username could not be duplicated (i.e., it must not already exist in the database) and could not be empty. While developing other functionalities, we introduced the newline character as a delimiter for messages sent between the client and server. To prevent parsing errors, we added further restrictions requiring that the username and password must not contain spaces or newline characters.
* At first, we did not restrict repeated logins for the same username. However, we realized that it didn’t make sense for the same account to be logged in at multiple locations and playing games with different players simultaneously. Therefore, we introduced a new logged\_in\_users set to track currently online users and ensure that each username can only be logged in at one time.

### Homepage

1. **Functionality Details and User Logic**

After a successful authentication and logging in, the user enters the home page. The homepage shows a welcome message at the top. The middle box shows the game credits of the logged in user and the number of players online that are both updated in real time. The homepage provides seven main options for the user:

1. Start Game: The quick start option that allows the user to join the matchmaking queue and start a game with the next online player in the queue.
2. Matching Room: The user can join the matching room and pair up with other players who also wish to find a match. The user can also send and receive game invites.
3. View Live Games: Watch ongoing games between other players.
4. Game History: Review past game history and credit changes and replay historic games.
5. Leaderboard: View the top 100 players ranked by game credits.
6. Delete Account: Permanently remove the account and associated data.
7. Log Out: Log out of current account.
8. **Development Process and Things We Learned**

* We started by sending a single request to the server to show the credits and number of online players, but then we realized that it would only make sense if these stats update automatically as they change. Thus, we added polling on the homepage to allow the numbers to update lively. For instance, the user will see the number of online players change as other players log in and out.
* At first we only used a single server class for handling all client requests, but soon we found it challenging to build all functionalities with a single class, so we divided the server into several classes by their roles, such as UserManager, GameManager, MatchMaking, GameServer, etc.
* We encountered errors when returning to the homepage and we found that the errors occurred because there were other polling processes called by other UIs that were still going on in the backend. Thus, we make sure that all other polling processes are stopped before leaving its associated page and returning to home.

### Start game

#### Game Playing and Live Game Handling

1. **Functionality Details and User Logic**

After pairing up with another player, the user enters the game immediately. The user will be randomly assigned a stone color–black or white. In any case, the black stone plays first. The user will see a message indicating their stone color and if they move first or second. The top bar of the game page shows the stone colors and the associated users. It also shows the time left for the current turn. Players should finish a turn in 30 seconds. If they do not place a stone in 30 seconds, the system will randomly place a stone for them and switch turns. On the top right the user can also see who is playing the current turn. The user can place a stone by clicking any empty intersections on the 19\*19 board. The board will update lively.

The winner is the first player who achieves consecutive five stones horizontally, vertically, or diagonally. When a win is detected, the game ends and the user sees a game over message that indicates who the winner was. The message will also show the game credit changes for the user. During the game, the user can exit at any time by clicking “Exit Game” at the bottom. However, this will count the opponent as the winner. There is no draw. If the board is full and no player achieves consecutive five stones, the white stone wins because the black stone has a first move advantage.

1. **Development Process and Things We Learned**

* The first problem we encountered was that after the user clicks “Start Game” and sees a waiting screen, they can click the button again and open another waiting screen, which did not make sense. Thus, we ensure that we clear the home UI before showing the waiting UI such that double waiting UIs will not appear.
* We initially did not include a turn timer. However, we thought about the case when it takes forever for the opponent to place a stone, making the other player wait. To enhance the user experience, we included a 30 second countdown timer for each turn. The players are required to place a stone within the timeframe, or the system will place a stone for them at a random place. To implement this timer, we thought about two ways–countdown by the server or by the client. In order to ensure that both players have a consensus on the remaining time of the current turn, we decided to implement the countdown on the server’s end by adding a “time\_remaining” field in the game state. The clients will then poll the remaining time as part of a game state polling.
* For invalid moves, initially we designed pop-up windows for users when they are trying to make an invalid move, say, when it is not their turn or the move is out of boundary. However, we think it makes more sense if the invalid moves are silently disabled. For example, when a user randomly clicks on the board when it is not their turn, the system should ignore their actions instead of popping up error windows saying their moves are invalid.
* After the game ends, at first we pop the user back to the home screen immediately. However, we later thought that it would be better if the user can remain on the board to review the game if they would like to. Thus, we enhanced this feature by adding an additional polling after the game ends, such that the final game board is shown and the user can then click “Exit Game” to return to the home screen.
* To handle the case where the opponent closes the window and disconnects, we encountered some problems in the beginning. We added a function on the client side to send a PLAYER\_DISCONNECTED message to the server, but on the server side, the server will also handle a PLAYER\_DISCONNECTED message itself when the client disconnects. This double calling caused connection errors and caused the game result window to not show up properly for the remaining player. We later discovered this repetitive call and removed the function called on window close, such that player disconnected is only handled once, and the result window should show to the remaining player.

#### Game Credit Calculation

1. **Functionality Details and User Logic**

The credit system is designed to reflect the competitiveness, difficulty, and efficiency of a game in a scalable and fair manner. When a game ends, the system adjusts both players’ credits based on the outcome. The winning player gains credits while the losing player loses credits. The change of credits will be displayed on a pop-up window when the game ends.

1. **Development Process and Things We Learned**

* In the beginning, stone\_efficiency was calculated as 1 / sqrt(S). However, this caused an error when S = 0, which can happen if a player exits the game or disconnects at the very start. To address this, we modified the formula to stone\_efficiency = 1 / sqrt(max(S, 9)), which prevents division by zero and avoids large credit changes when a player quits early. We assume the exiting player loses in the minimum possible number of steps, which is 9.

### Matching Room

1. **Functionality Details and User Logic**

The matching room page allows users to view potential opponents currently in the room and manage game invitations in real time. On the left panel, users see a dynamic list of available players, including their usernames and game credits. This list can be refined using a search bar or filtered by credit range, making it easy to find opponents of similar skill or desired competitiveness. This feature enhances the user experience by supporting purposeful matchmaking and accommodating both casual and ranked play. Users can select a player from the list and send a game invitation. Only users who are in the matching room appear in the list of available players, as we assume all users in the room are open to sending or receiving invitations. Each user can send only one invitation at a time and must either cancel the current invitation or wait 15 seconds for it to expire before sending another.

On the right panel, users can view their pending game invitations. Each invitation displays the sender’s username, current credits, and a countdown timer showing how much time remains before the invitation expires. Invitations are valid for 15 seconds, and those with fewer than 5 seconds remaining are highlighted in red to prompt timely responses. Users can select a pending invitation and choose to either accept or decline it using the corresponding buttons below.

1. **Development Process and Things We Learned**

* One problem we encountered during development was that the filtered results in the user list panel were not persistent. The filtered user list will appear for a second and then the list changes back to the unfiltered one immediately. We solved this problem by applying the search filter dynamically and applying the filters every time we call update\_users\_list. In addition, we made the selection in the list persistent, such that if the user selects a player in the original list, and after applying the filter the selected player is still there, the selection will be automatically populated. These enhancements were implemented to achieve a smoother user experience.
* We then encountered a bug where an invitation would reappear with a 15-second countdown after it had expired. This happened because expired invitations were not properly removed on the client side. To fix this issue, we added logic to update\_invitation\_timers to check whether an invitation has expired and, if so, call \_remove\_invitation. We also noticed that expired invitations remained in the server-side match\_requests dictionary. To address this, we implemented a \_clean\_expired\_requests function, which is called by the \_check\_timers helper method that updates server state every second.
* Initially, we sent a MATCH\_CANCEL message to clients when the server received a response from the invitation receiver. This was aimed to clear out the match request after it had been handled, but it caused confusion and unwanted pop-ups for users who had already started a game. To improve the user experience, we introduced a new handle\_match\_response function on the client side. If an invitation is accepted, the server sends a MATCH\_FOUND message to both users, and handle\_match\_response updates the UI of the waiting page. If the invitation is declined, a new message type MATCH\_DECLINED is sent instead, and handle\_match\_response notifies the sender appropriately. Now, MATCH\_CANCEL is only sent when the sender explicitly cancels the request, and only the receiver’s client handles its removal from both frontend and backend.
* While testing edge cases, we discovered that if the invitation sender disconnected from the server immediately after sending an invitation, the request still appeared on the receiver's panel. This caused an error if the receiver tried to accept it. To handle this, we added logic to detect and cancel any active outgoing invitations when a user disconnects. The server calls \_cancel\_match\_request, and a MATCH\_CANCEL message is sent to the receiver to remove the now-invalid invitation.

### Live Game Viewer

1. **Functionality Details and User Logic**

The Live Game Viewer function allows users to watch ongoing games in real time. When users click the “View Live Games” button in the homepage, they are first presented with a list of currently active games, each showing player names, the total number of stones placed so far, and the current turn. This list updates automatically as new games start or existing games finish. Users can select any game from the list and click the “View Game” button to open a live board view. The game board then displays the current state of the selected game and updates automatically as players make moves. Users can only view one game at a time, and they have to close the current game board in order to select and view another live game. This feature enhances engagement by allowing spectatorship and makes it easy for users to follow competitive matches or learn strategies from other players in real time.

1. **Development Process and Things We Learned**

* After selecting a live game and clicking the “View Game” button, a board pops up showing the current state of the selected game. To avoid redundant code, we reused the GameUI class with view\_mode=True to draw the board and stones in real-time. The biggest challenge in this feature was updating the board state properly. Initially, only the board appeared—no stones were shown, and it never updated. To fix this, we added a polling function poll\_game\_stat in the LiveGameViewer class, which sends a GET\_GAME\_STATE message every 0.25 seconds to retrieve the latest game data. When the client receives the response, it calls update\_game\_state in the LiveGameViewer class to organize the data and then calls game\_ui.update\_game\_state to reflect the updates on the live game viewer interface. In the GameUI class, when viewer\_mode=True, we ensured that users cannot place stones on the board since they are not players in the game, and we modified parts of the UI to improve the viewer experience. With these functions in place, the board updates correctly, and users can smoothly view live games.
* We also encountered a bug where, after the game ended, the viewer’s client sometimes failed to update the last stone placed and did not show the game-over window. We realized this was because the client stopped polling immediately after receiving ‘game\_over’=True in the game state data, but the LiveGameViewerUI page may not have updated its internal status in time and therefore failed to call the \_show\_game\_over\_popup function. To resolve this, we added the \_handle\_final\_state function to handle the case where the received game state data contains ‘game\_over’=True but the internal self.game\_over is still False. This function updates self.game\_over to True, adds the final stone to the board, and calls \_show\_game\_over\_popup to display the game result.
* In the scenario where one player disconnects, the game ends for both players. However, the viewer’s board would freeze without any notification. To address this, we added a variable self.last\_update\_time to record the time when polling data was last received. If no data is received for more than 2 seconds, we call the handle\_disconnect function to display a popup notifying the viewer that one player has disconnected and the game has ended. This improves the viewer’s experience by providing timely feedback about game termination.

### Game History

1. **Functionality Details and User Logic**

The Game History and Replay feature allows users to review their completed matches and analyze past gameplay. After clicking the “Game History” button in the homepage, the users will be directed to the history UI page that displays a list of all previous games in reverse chronological order. Each entry shows key game information such as opponent name, game result (win or loss), credit changes, and the date the game was played. When a user selects a specific game from the list, the “View Replay” button becomes active, and an empty game board is initialized for playback after clicking it. Users can step through the match move-by-move using forward (“>”) and backward (“<”) buttons or watch an automatic playback using the fast-forward (“>>”) and start-from-beginning (“<<”) controls. Each stone is labeled with the number of steps to indicate their order of placement, helping users visualize the flow of the match. This feature supports both casual review and strategic analysis, making it easy for players to reflect on their performance and learn from past games.

1. **Development Process and Things We Learned**

* We first implemented the HistoryUI class to display all game history for the user, including basic information such as the opponent’s username, win/loss result, credit changes, and timestamp. When a user selects a game and clicks the “View Replay” button, a GameReplayUI class is created and an empty board pops up. To avoid redundant code, the GameReplayUI class inherits from the GameUI class and sets replay\_mode=True. Initially, an “Exit Game” button was shown at the bottom of the replay UI page because it was inherited from GameUI. However, we realized that clicking this button would result in an error, as it sends a PLAYER\_DISCONNECTED message to the server. To fix this, we modified the code in the GameUI class to ensure the “Exit Game” button is not displayed when replay\_mode=True.
* We initially used a list (move\_history) to store move history data on the server side. However, we later discovered that the saved move history could become mixed up if two games were played simultaneously. To address this, we changed move\_history to a dictionary, where the key is the game\_id and the value is the corresponding list of move history data. This change ensures that game data is saved correctly even when multiple games are happening concurrently.

### Leaderboard

1. **Functionality Details and User Logic**

The leaderboard function provides a ranked overview of player performance across the game platform. It displays a list of top 100 players sorted primarily by their total game credits, which reflects both skill and consistency. For each player, the leaderboard shows their rank, username, current game credits, number of wins, and number of losses. This allows users to quickly compare their standing against others and see who the top players are. When users click the “Refresh” button, the latest leaderboard will be retrieved and displayed. Users can scroll through the list to view rankings or locate specific players, encouraging motivation and engagement within the game’s competitive ecosystem.

1. **Development Process and Things We Learned**

* The leaderboard function is relatively simple compared to the other functionalities. The logic is to retrieve all user information from the server database and sort the users by game credits (in descending order), number of wins (descending), and number of losses (ascending). Initially, the leaderboard page only had a “Back to Home” button. Later, we realized that since there was no polling function on the page, the only way users could view the updated leaderboard was by returning to the homepage and then navigating back to the leaderboard page, which was inconvenient. To improve this, we added a “Refresh” button that allows users to manually update the leaderboard and view the latest rankings.

### Delete Account

1. **Functionality Details and User Logic**

On clicking “Delete Account” on the homepage, a pop up window will appear to ask the user to confirm account deletion. If “no”, no changes will be made. If “yes”, the user’s account is deleted and all associated data is removed. The user will be brought back to the authentication UI. The user’s username will show as “account deleted” in all game histories when viewed by other users.

1. **Development Process and Things We Learned**

* One problem we encountered when developing account deletion is that there would be pop-up windows saying “lost connection to server” when returning to the authentication UI. We found that this error occurred because after the account is deleted and the current username is set to None, the polling process from the matching room UI is not stopped and still polls for stats that contain the username entry, even though the user has already been removed. As a result, the lost connection error occurs every time the user tries to delete their account after having entered the matching room. Thus, we solved this problem and enhanced the implementation by ensuring a smooth exit from the matching room and stopping polling before exiting the matching room UI.

### Log Out

1. **Functionality Details and User Logic**

On clicking “Log Out” on the homepage, the user will be logged out of their current account and will be redirected to the authentication page. In addition, if a user disconnects from the server, the user is logged out and leaves the queue automatically.

1. **Development Process and Things We Learned**

* One bug we encountered was that the number of online players did not change as a user logs out. We found that the user needs to be removed from the current logged in users to reflect the change in the number of online players.

## Test and Documentation

We conducted a comprehensive unittest for the game application, including tests on shared models, database, user manager, match making, game manager, game server, client UIs, and game client. The tests cover all classes and functionalities provided by the application. We also tried to come up with different edge cases to test the correctness of error handling.

Finally, we added comments and docstrings throughout the codebase to improve readability and maintainability. Each class, method, and function includes a clear description of its purpose, inputs, and outputs to help future developers understand and navigate the code easily. In addition, we created a comprehensive README.md file that explains how to set up and run the system and outlines the main features.

## Future Work

Due to the limited time available for the final project, there are several issues and improvements that could be addressed in future work:

* **Add Replication**: After implementing the core functionalities proposed in our initial design (such as starting a game and reviewing game history), we chose to prioritize developing additional features to enrich the gameplay and explore distributed systems concepts and had limited time to explore replication. These advanced features include the matching room, live game viewer, leaderboard, and account deletion. We still plan to explore the possibility of adding replication in the future when more time is available.
* **Use a More Advanced Database**: We currently use JSON files to store user data, past games, and live game state. However, JSON files are susceptible to read/write conflicts that may lead to inconsistencies when serving large numbers of players. A better long-term solution would be to migrate to a more advanced database system, such as SQLite, which supports concurrent access and can eliminate this issue at the root.
* **Use a More Efficient Communication Protocol**: Our current system uses a JSON wire protocol for communication between the server and clients. Although this protocol is simple and easy to implement, it is not very efficient. In the future, we hope to revisit and explore more efficient communication protocols such as gRPC.
* **Deploy as an Online Website or Downloadable Software**: Currently, the system runs only on devices with Python installed, and both the server and clients must be on the same network due to IP restrictions. In the future, we would like to deploy the program as an online website or standalone software, so users can access the game without needing Python. Additionally, we hope to remove the limitation that requires the server and clients to be on the same local network, making the system accessible over the internet.