**Hanoi University of Science and Technology  
School of Information and Communication Technology**



**FINAL PROJECT**

**Caro Game with Socket Programming**

***Subject: Network Programming***

**Supervisor: Ph.D Truong Thi Dieu Linh**

**Group 3:**

|  |  |  |
| --- | --- | --- |
| **Student ID** | **Student name** | **Class** |
| 20176732 | **Tran Thai Duong** | **ICT 03 – K62** |
| 20176845 | **Nguyen Manh Phuc** | **ICT 04 – K62** |
| 20184238 | **Nguyen Thi Minh Chau** | **ICT 01 – K63** |

**Hanoi, 2021**

**Table of Contents**

[**Table of figures** 3](#_Toc75766072)

[**PART I: INTRODUCTION** 4](#_Toc75766073)

[**1. General description** 4](#_Toc75766074)

[**2. Project objectives** 5](#_Toc75766075)

[**PART II: DESIGN** 6](#_Toc75766076)

[**1. Basic flow of the application** 6](#_Toc75766077)

[**2. Function of the application** 7](#_Toc75766078)

[On the client side 7](#_Toc75766079)

[On the server side 8](#_Toc75766080)

[**3. State machines** 9](#_Toc75766081)

[*State machines for the main flow of application* 9](#_Toc75766082)

[*State machines for the in-game play* 11](#_Toc75766083)

[*State machines for in-game chat* 13](#_Toc75766084)

[**4. Protocol design** 14](#_Toc75766085)

[Register function 15](#_Toc75766086)

[Login function 16](#_Toc75766087)

[Join queue function 17](#_Toc75766088)

[Moves 18](#_Toc75766089)

[Request draw function 20](#_Toc75766090)

[Check ELO and rank, show the leaderboard 22](#_Toc75766091)

[Chat function 23](#_Toc75766092)

[Logout function 25](#_Toc75766093)

[**5. Network Model** 25](#_Toc75766094)

[**6. Database** 26](#_Toc75766095)

[**PART III: IMPLEMENTATION** 27](#_Toc75766096)

[**1.** **Server** 27](#_Toc75766097)

[**2.** **Client** 27](#_Toc75766098)

# **Table of figures**

[Figure 1: State machine for the main application - Client side 10](file:///E:\BKHN\Chous_Projects\networkprogramming_20202\project\reports.docx#_Toc70802584)

[Figure 2: State machine for the main application - Server side 11](file:///E:\BKHN\Chous_Projects\networkprogramming_20202\project\reports.docx#_Toc70802585)

[Figure 3: State machine for ingame play - Client side 12](file:///E:\BKHN\Chous_Projects\networkprogramming_20202\project\reports.docx#_Toc70802586)

[Figure 4: State machine for the ingame play - Server side 13](file:///E:\BKHN\Chous_Projects\networkprogramming_20202\project\reports.docx#_Toc70802587)

[Figure 5: State machine for the ingame chat - Client side 14](file:///E:\BKHN\Chous_Projects\networkprogramming_20202\project\reports.docx#_Toc70802589)

[Figure 6: State machine for the ingame chat - Server side 14](file:///E:\BKHN\Chous_Projects\networkprogramming_20202\project\reports.docx#_Toc70802588)

# **PART I: INTRODUCTION**

## **1. General description**

Internet has become popular for decades for its application: not only it is used for connecting people over the world and spreading information, but also for entertainment. One of the mostly used features of internet is to provide a variety range of games for users to enjoy themselves in their free time, or after a hard-working day.

Online games exist in different forms, two of which is web-game and client-game. Web games are implemented to be played on browsers, typically depends on various players and add-ons in order to function properly. On the other hand, web game does not require users to download or install any program on host machine, which may be more convenient for players that do not want to waste their device memory for some application they rarely use.

Client game needs user to fetch the program. Some either demand installation or not. Despite being heavier than web game in term of size, client game outweighs its counterpart on many aspects. Not only is it more stable, but also more independent from users’ platforms.

Currently, both methods are now widely used to build online games, even if it is a large-scale game with multiple players, or mini game that has each session lasts for some minutes only.

Caro (originally Gomoku) is a popular abstract strategy board game. Traditionally, it is played with black and white pieces on a 15x15 or 19x19 board. However, since these pieces are not typically moved after being placed, they can be replaced by X or O notes on papers and transformed into Caro game.

In this version, players put an X or an O according to their initial choice on their turn so as to create a horizontal, vertical or diagonal chain of 5 or more consecutive notes. By convention, if there is no agreement on both side at the start of the game, X will go first. Since its rule is simple, Caro is usually chosen by many people to play in their break time.

As technology becomes stronger day by days, Caro can now be played in different ways other than using pen and paper. People all around the world can meet in a Caro playground online to have some fun, as well as show their talent. Friendships are made using a small and interesting game put globally.

Besides making a Caro web game, which is lightweight but relies on various prerequisites, a client-based stand-alone game will save user from struggling with plugins for different browsers, but still ensure the convenience, as well as the fun for players.

## **2. Project objectives**

For this course, the main objective of the project is to understand the basis of socket programming. By the project, understandings of client-server and P2P models, different approaches of blocking and synchronization between hosts can be gained. Also, protocol design process, which is important for communication between devices, is also brought into practice.

The aim of the project is to make a stand-alone Caro game application that can connect to the internet and allow users to match with other players all over the world. As aforementioned, this implementation will bring convenience for user when enjoying the tradition of the game and do not have to concern about platform compatible.

Since the basis of Caro is easy to integrate into an application, the work needed to be focused on is network programming. In this project, a client-server model is used for the purpose of having a 3rd party system participate in matching players, which is more convenient than using P2P model. Sockets is built using asynchronous method with the help of Java libraries in order to handle multiple accesses and requests from hosts, thus enhance performance of the application. The design of protocol is based on practical use cases of the program. JSON is preferred for building the communication agreement between clients and servers for advantaging the well-structured and easy-to-query properties of the language.

# **PART II: DESIGN**

## **1. Basic flow of the application**

Caro stand-alone game application will give users a chance to register an account and play as a user in the system. In this case, they can choose to play in either normal or ranked mode. The ranked mode applies some scoring strategy in order to determine the ELO, as well as the rank of a user in the game.

Each play mode has a separated queue. After logging in, users are put into the hall until he or she decides the mode they want to join in. Then, they will be taken to the appropriate queue for waiting the system to find a game for them. Players in normal queue will be match randomly, while a matching method based on ELO is applied for ranked queue so as to find a suitable opponent.

In the game, players make their moves by turn, and information of the moves is forwarded to the other via the server. The clients decides if the game reaches the end and send a notification to the server. The server then sends a message to announce both players in the match that the game has come to the final, as well as their ELO after the match (in ranked mode).

Players in the match can chat to each other. Also, the game provides an option of requesting a draw when a player cannot decide a move. In this case, the request will also be sent to the opponent via server. If he or she accepts, the match ends with the result of draw. Else, the game continues normally.

Players can also choose to play as a guest in order to not registering an account. In this case, they will immediately be put in the normal queue, and their final result will not be recorded in the system.

## **2. Function of the application**

Based on the stated flow, there are some basic functions:

#### On the client side

The client is responsible for connecting user to the server, listen and forward in-game moves, and performs these functions:

- **Register**: user can register an account with a username and password to save their in-game information: their ELO, as well as their rank in the game system

- **Login**: used for user that has an account. A player that logs in with their registered username and password before a match will have their ranked result stored in the system database.

- **Play as guest**: Players can choose to play without registering an account or logging in. In this case, they cannot play in ranked mode, and their result will not be tracked by the system.

- **Join a game**: When playing as a user, players can choose to play in normal or ranked mode. Based on the option, they will be put into a proper ***queue*** for finding match. For ranked games, the game result will be stored on the server.

- **Request draw**: player in the game can request a draw to the opponent when he or she cannot decide the next move. If accepted, the match ends with a draw, and continue vice versa.

- **In-game chat**: players in the same match can have conversation with others via in-game chat.

- **Show rank**: Users in the system are ranked based on their ELO. This option show users’ current rank, as well as top 50 in the system.

#### On the server side

The server is responsible for receiving new connection from clients, processing and forwarding requests between hosts. Also, the server has to put users in hall or in appropriate queue based on users’ choice, finding a match, as well as recording and calculating users’ ELO and rank.

- **Receive register request:** The server receives register request with a username and password, check conditions and reply the client with a message of success or failure before saving the information into a database and put user into ***hall***.

- **Receive login request:** The server receives login request with a username and password, check if the username and password match the database in the system, and return a message of success or failure before logging the user in and put user into ***hall***.

- **Put users in *hall*:** Users after logged in are put into the ***hall*** for choosing a play mode.

- **Put users in *queue*:** User after choosing a play mode will be put into appropriate ***queue***: normal or ranked. If the player chooses to play as a guest, he or she will be sent to the normal queue immediately.

- **Find a match**: The server finds an opponent for player in the queue and send back information of found match. Players in normal queue are matched randomly, while players in ranked queue will match if their ELO satisfy conditions of the matching algorithm.

- **Receive and send moves information**: The server receives information about moves of each player in each turn and forward this to the opponents. If there is an announcement of ending the game, the server sends a notification to both players about the final state of the match, as well as their ELO after the game if it is in ranked mode.

- **Receive and send draw request**: since players have a choice of asking for a draw, server will take this request, forward it to the opponent and wait for response before the game ends or continues, based on this response.

- **Receive and forward chat message**: the server acts as a delivery to forward messages between users in a match. No data of message is stored in this case.

- **Receive checking rank request and returns rank**: the server queries top 50 players in the system, as well as the rank of the requester when receiving an ask for showing the rank.

## **3. State machines**

*State machines for the main flow of application*

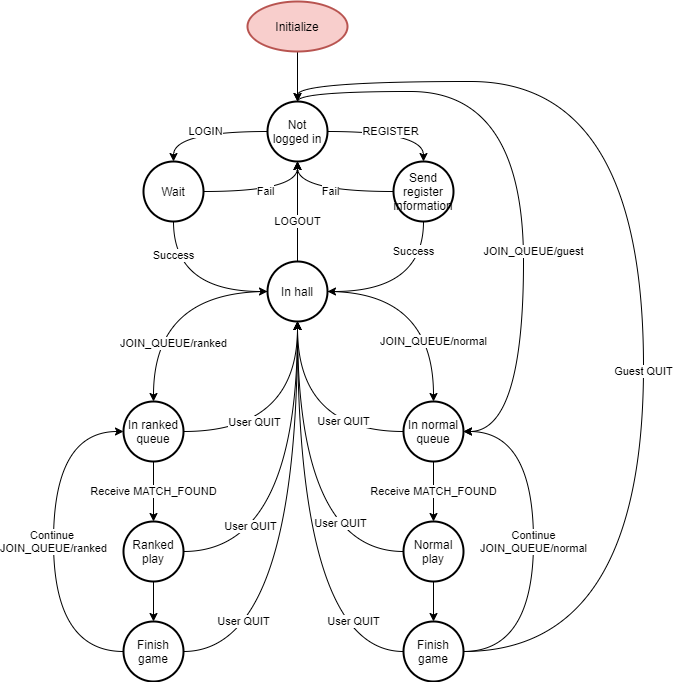


Figure 1: State machine for the main application - Client side

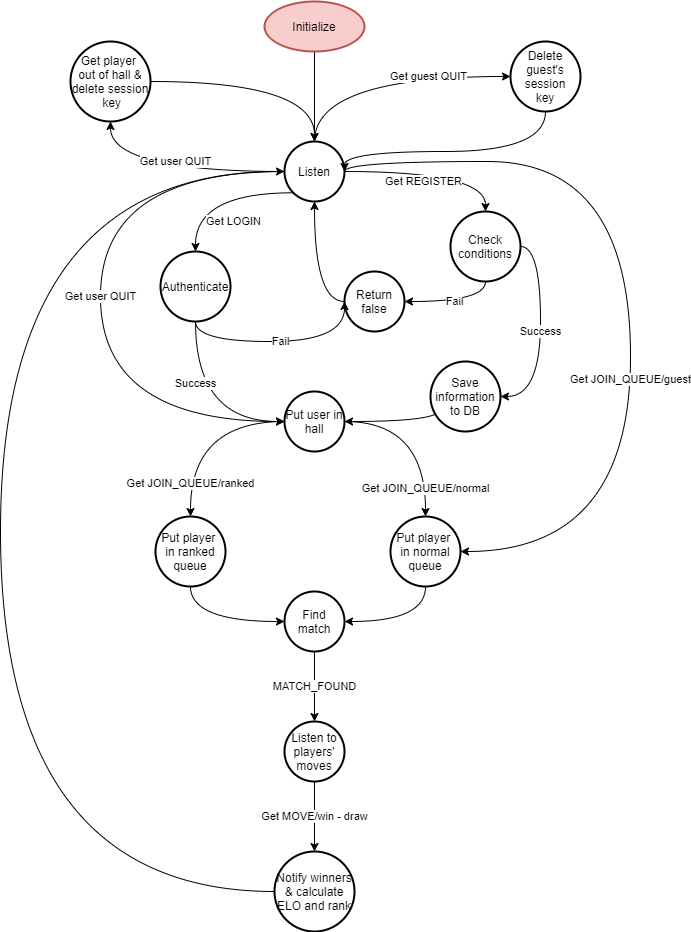


Figure 2: State machine for the main application - Server side

*State machines for the in-game play*

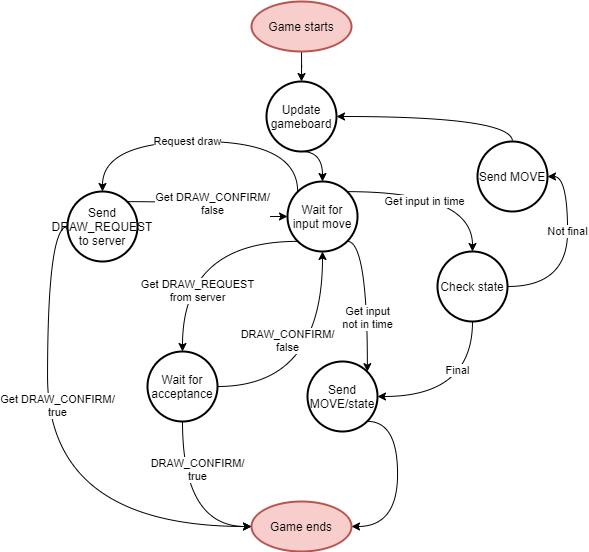


Figure 3: State machine for ingame play - Client side

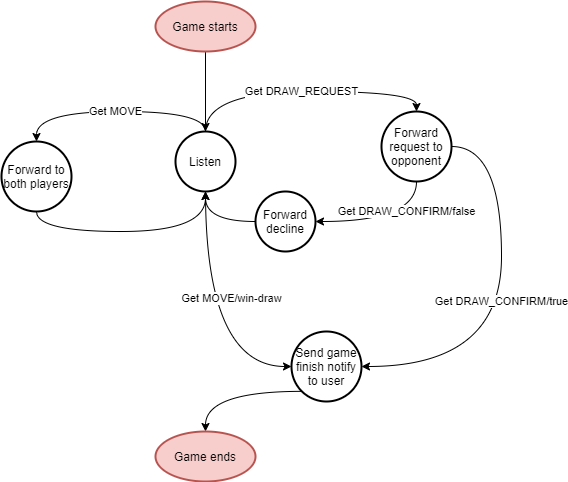


Figure 4: State machine for the ingame play - Server side

*State machines for in-game chat*

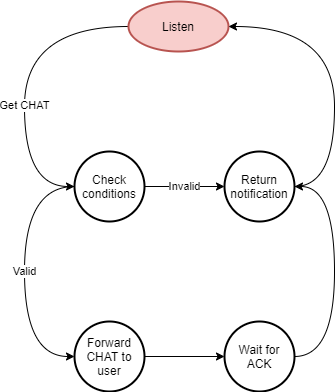


Figure 6: State machine for the ingame chat - Server side

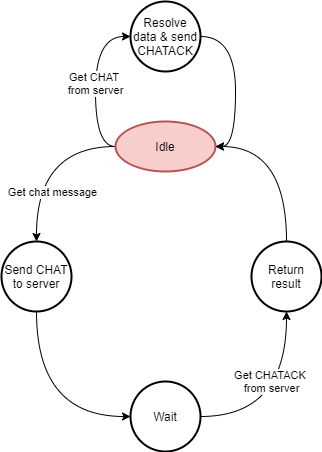


Figure 5: State machine for the ingame chat - Client side

## **4. Protocol design**

For convenience, the protocols of client and server are built with JSON language. The purpose for choosing this convention is due to:

- The well-structured property of JSON

- Fast query, since Java provides libraries for building and parsing JSON object to information and vice versa

The protocols follow the basic structure:

- Message from the client side: including a command code and information of the command

- Message from the server side: including a command code, information of the command, a status code for indicating the result of the function, and an error message for details of the failure (if there is)

Command codes are capitalized and uses underscores instead of white spaces. The information of the command code varies depending on the command code. The status code has two values: “success” or “error”. One return message from the server must have status code value. If the status code is “error”, the error field must clearly indicate the failure information.

In order to get the information of the message, server will check the command code before processing further data. On the client side, when receiving message from the server, after parsing the command code, it will check the status code. If the status code indicates an error, client will immediately check the error message without further processing additional information to save time and memory.

In this application, a field of session ID is used in some message. The reasons of this session ID are:

- **Security**: The session ID are refreshed frequently and is available for 5 minutes after the last activity of the user. This is to prevent cheaters to perform malicious behavior on users’ account, thus affects user experience when using the application.

- **Indication**: Using a pair of username and session ID, the system can easily keep track of the state of currently online users, as well as distinguishing users and guests.

- **Utilization**: Since session IDs are periodically checked, users that are not active for a long time will be considered offline. They will be removed from the hall to save memory and free space for new incoming players.

#### Register function

- From Client to Server: using POST method for security

{

"command\_code": "REGISTER",

"info": {

"username": "abc",

"password": "xyz",

}

}

- From Server to Client: using GET method

{

"command\_code": "REGISTER",

"info": {

"session\_id": "AOSJHBGASLGJB"/"",

"username": "duongdz",

"elo": 1500

},

"status\_code": "success"/"error",

"error": ""

}

When a player wants to register an account to save their game result on the server, he or she will request to server with REGISTER command with username and password. The server after receiving the command and information will check:

- If the chosen username exists in the database

- If the chosen name violates naming convention: username MUST NOT have prefix “anon”, since this is the special prefix for distinguishing if a player is a guest or registered user

If the username satisfies both conventions, the information is stored in database, and a message with status code “success” will be returned along with a new session ID for determining the login session of user. Otherwise, the register fails. Server will return a status code of “error” with a message indicating details of the error, and no session ID is provided.

#### Login function

- From Client to Server: using POST method for security

{

"command\_code": "LOGIN",

"info": {

"username": "duongdz",

"password": "abcdef"

}

}

- From Server to Client: using GET method

{

"command\_code":"LOGIN",

"info":{

"session\_id": "AOSJHBGASLGJB"/"",

"username": "duongdz",

"elo": 1500/0

},

"status\_code":"success"/"error"

"error":""

}

To log in, user send information of his username and password to the server with command LOGIN. The server then checks if the received information matches the saved information in database.

If true, then the login successes. The server then put user into the game’s hall and returns to client a message with status code of “success”, along with a session ID for determining the login session of user and a field of user’s current ELO for displaying on the client.

If the check fails, server sends a message of status code “error” and details of the failure. No session ID and ELO is provided in this case.

#### Join queue function

|  |  |
| --- | --- |
|  | |
|  | |  |
|  |
|  |  | |

- From Client to Server: Using POST method

{

"command\_code":"JOIN\_QUEUE",

"info":{

"mode": "normal"/"ranked",

"session\_id": "AOSJHBGASLGJB"/"",

}

}

- From Server to Client:

{

"command\_code":"JOIN\_QUEUE",

"info":{

"session\_id": "AOSJHBGASLGJB"/"ASLJGLJSBGEF",

"username": "duongdz"/"anon1234",

"player\_1": "duongdz"/"anon1234",

"match\_id": "1234",

"opponent": "phuc"/"anon1235",

"opponent\_elo": "1523",

}

"status\_code": "success"/"error",

"error": "",

}

To join a new game, player must choose to play in normal or ranked mode. The information of the mode, as well as current player session ID will be sent to the server. Based on the information, the server put players in proper queue, and return a message with the player’s username and session ID for confirmation, status code of success or failure with error message.

If a player chooses to play as a guest, his session ID will be empty. In this case, the server returns a random username along with a session ID for player to join the normal queue, and information about success of the command.

After having found an appropriate opponent for the match, server will send a notification to both user of the match about the found game. The client will take the information and prepare the game board. The match ID is used for both sides to keep track of current game.

#### Quit queue function:

- From Client to Server: Using POST method

{

"command\_code":"QUIT\_QUEUE",

"info":{

"username": "phuc",

"session\_id": "AOSJHBGASLGJB",

}

}

- From Server to Client:

{

"command\_code":"QUIT\_QUEUE",

"info":{

"username": "phuc",

}

"status\_code": "success"/"error",

"error": "",

}

#### Moves

- Using POST method for security

- From Client to Server:

{

"command\_code": "MOVE",

"info": {

"match\_id": 1234,

"session\_id": "AOSJHBGASLGJB",

"move\_player": "duongdz",

"move\_position": {

"x": "6",

"y": "9"

},

"state": "valid"/"invalid",

"result": "win"/"lose"/"draw"/""

}

}

{

"command\_code": "LISTEN\_MOVE",

"info": {

"match\_id": 1234,

"username": "phuc"

}

}

- From Server to Client

{

"command\_code": "MOVE",

"info": {

"match\_id": 1234,

"move\_player": "duongdz",

"move\_position": {

"x": "6",

"y": "9"

},

"state": "valid"/"invalid",

"result": "win"/"lose"/"draw"/""

},

"status\_code": "success"/"error",

"error": "",

}

For each turn, the client will wait to get coordinate of the move that the player plays. If the player places a move in time, the state of the move is valid, and its position will be recorded and sent to the server, along with the match ID of the game and session ID of user the do the moves for confirmation. If the gameboard does not reach the final state after the move, the result of the move is empty; otherwise, it will be “win” or “draw” accordingly. Sever will receive the information and forward the moves or notify winner.

If the state of the move is “invalid”, the player is considered lose and the result will be “lose”. In this case, the game immediately reaches the end and server will notify about the result instead of forwarding moves.

After receiving a MOVE command that indicates the final state of the game, the server will send a message to notify both players about the result of the game, including the winner, the loser and ELO of each player after the game for displaying on the client side. Also, the ELO and rank of players are updated in the leaderboard database on the server side.

The field “draw” is used for easily parsing data: the client will first check this field and only perform further process if the value of this field is “false”.

#### Request draw function

- Using POST method for security

- From Client to Server:

{

"command\_code": "DRAW\_CONFIRM",

"info": {

"match\_id": 1234,

"session\_id": "AOSJHBGASLGJB",

"move\_player": "duongdz",

"acceptance": true/false

}

}

{

"command\_code": "DRAW\_REQUEST",

"info": {

"match\_id": 1234,

"session\_id": "AOSJHBGASABCD",

"move\_player": "phuc",

}

}

{

"command\_code": "LISTEN\_DRAW",

"info": {

"match\_id": 1234,

"session\_id": "AOSJHBGASABCD",

"username": "phuc",

}

}

- From Server to Client:

{

"command\_code": "DRAW\_REQUEST",

"info": {

"match\_id": 1234,

"move\_player": "phuc",

},

"status\_code": "success"/"error",

"error": ""

}

{

"command\_code": "DRAW\_CONFIRM",

"info": {

"match\_id": 1234,

"move\_player": "duongdz",

"acceptance": true/false,

"session\_id": "AOSJHBGASLGJB",

},

"status\_code": "success"/"error",

"error": ""

}

When player wants to request a draw, DRAW\_REQUEST command will be sent with the match ID and the username that perform the request. The request will be forwarded to the opponent via the server, and the receiver can have options of accept or refuse. If he accepts the request in time, the “acceptance” field of DRAW\_CONFIRM message will be true and vice versa.

When the server received a DRAW\_CONFIRM command it will forward this message to the player that raised the request. If it is an acceptance, the server immediately broadcast a notification of the ending state of the game to both players, otherwise, the game continues.

#### Check ELO and rank, show the leaderboard

- Using GET method

- From Client to Sever:

{

"command\_code": "LEADERBOARD"

"info": {

"session\_id": "AOSJHBGASLGJB",

"username": "duongdz"

},

}

- From Server to Client:

{

"command\_code": "LEADERBOARD",

"info": {

"username": ["duongdz", "phuc", "einh", "lanhonglee"],

"elo": [1390, 1490, 1470, 1460],

"rank": [69, 1, 2, 3]

},

"status\_code": "success"/"error",

"error": ""

}

Users can request to check his current position on the leaderboard as well as the top 50 players in server. After receiving command LEADERBOARD, the server returns a message including 3 fields as information:

- A list of usernames

- A list of ELO points according to usernames

- A list of ranks according to usernames

At index 0 of each list is the information of the current user. From index 1, the lists indicate ELOs and ranks of the top 50 players in the system. Using 3 lists instead of one list of objects will reduce complexity and save space while ensure the information are still kept in order since they are indexed and will not be messed up on transmission and parsing process. On the client side, parse will get the information and map them according to their indexes.

#### Chat function

- Using POST method for security

- From Client to Server

+ Message

{

"command\_code": "CHAT",

"info": {

"from\_user": "duongdz",

"to\_user": "phuc",

"message": "helloooooooooo",

"message\_id": "CHAT1234",

"match\_id": 1234

}

}

{

"command\_code": "LISTEN\_CHAT",

"info": {

"username": "duongdz",

"match\_id": 1234

}

}

From the client, the chat message is fetched and sent to the server with the target user’s username, message ID (randomly generated and is valid for current match only) and the match ID. The server will forward this to the other player in the game.

- From Server to Client

+ Message

{

"command\_code": "CHAT",

"info": {

"from\_user": "duongdz",

"to\_user": "phuc",

"message": "helloooooooooo",

"message\_id": "CHAT1234",

"match\_id": 1234

}

"status\_code": "success"/"error",

"error": ""

}

#### Quit game function:

- From Client to Server: Using POST method

{

"command\_code":"QUIT\_GAME",

"info":{

"username": "phuc",

"session\_id": "AOSJHBGASLGJB",

"match\_id": 1234

}

}

- From Server to Client:

{

"command\_code":"QUIT\_GAME",

"info":{

"username": "phuc",

"session\_id": "AOSJHBGASLGJB",

}

"status\_code": "success"/"error",

"error": "",

}

#### Logout function

- Only have message from Client to server, using POST method

{

"command\_code": "LOGOUT",

"info": {

"username": "duongdz",

"session\_id": "AOSJHBGASLGJB",

}

}

After receiving LOGOUT request from user, server will delete the user’s session key in the database. The same operation is done on the client side in order to ensure security of the application on both sides.

#### Update user function

- From Client to Server: Using POST method

{

"command\_code":"UPDATE\_USER",

"info":{

"username": "phuc",

"session\_id": "AOSJHBGASLGJB"

}

}

- From Server to Client:

{

"command\_code":"UPDATE\_USER ",

"info":{

"username": "duongdz",

"win\_rate": "100",

"num\_game\_played": "10",

"num\_game\_won": "10",

"rank": "1",

"elo": "1523",

}

"status\_code": "success"/"error",

"error": "",

}

## **5. Network Model**

* In this application, a client-server model is used. There are advantages when using this model over P2P model:

- When users are connected to the game, a central server responsible for putting them into hall and queue is better for administration than having some of the client taking the part of the manager turn by turn.

- A central server playing the role of the matcher shows convenience over letting clients freely finding the game.

- Since the application has a ranked system, using a server to determine ELO and rank of each player, as well as saving this information and returning it to the client when needed is more appropriate than having each client saving the database on their machine.

The application is expected to accept multiple users, as well as process multiple requests in a short period of time. Therefore, the choice of asynchronous socket programming is used

An asynchronous server allows multiple sockets to be simultaneously managed in one or multiple threads. In this case, single-threaded is preferred for scalability. Multithreaded will cause a waste of system resources, as well as raising shared resources problems when the number of concurrent connections extends.

The client and server agree on a common protocol in order to send and receive messages and functions accordingly.

* The application uses asynchronous socket in order to create connection and communicate between clients and server.

In this model, client or server makes the non-blocking call and start other background tasks instead of waiting for the request to be completed. When the operation is done, a completion callback is generated to finish the I/O operation.

Asynchronous socket communication brings several advantages:

- Allow multiple connection at the same time with good quality: while blocking (in blocking socket calls) prevents the program from completing other operation until the request is returned, and polling (in selecting) consumes computer memory and decrease performance, asynchronous socket allows background works to be continue when the I/O call is being processed, and only notifies when the call is completed.

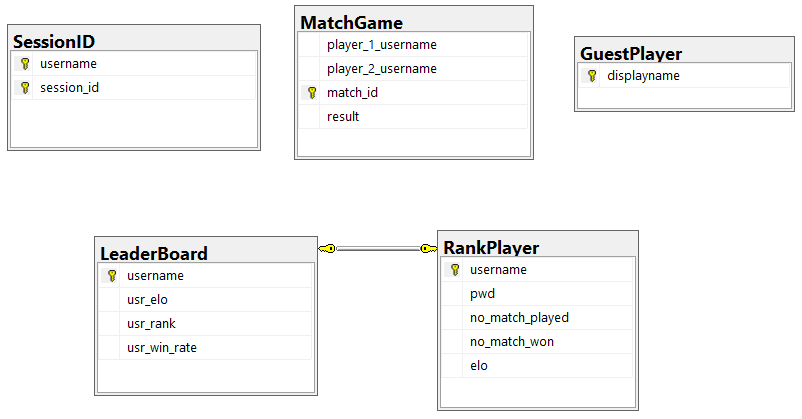
- Save resources and avoid threading problems: Although creating threads for handling multiple connection at the same time also allow many clients to communicate with the server, this waste the memory since each thread need its own space for saving its metadata. Also, because threads use the same memory space of the program and separated threads may have operation on the same resource at the same time, deadlocks may raise. Asynchronous socket can be implemented on one thread only, which does not require additional memory, prevent mutexes and other synchronizing problems of multithreaded programs.

Clients and server communicate using TCP protocol. TCP has mechanisms for guaranteeing transmission of packets, so that the program can be eliminated from having to check if the message is successfully sent. The data stream is not complicated and there is no requirement as for streaming services; therefore, TCP is considered suitable.

Since there is the need of keeping track of each action performed on the client side, the communication is stateful, and information of the client is saved in a database on the server side for further references.

## **6. Database**

In this program, we use SQLite to save database, with the diagram as shown

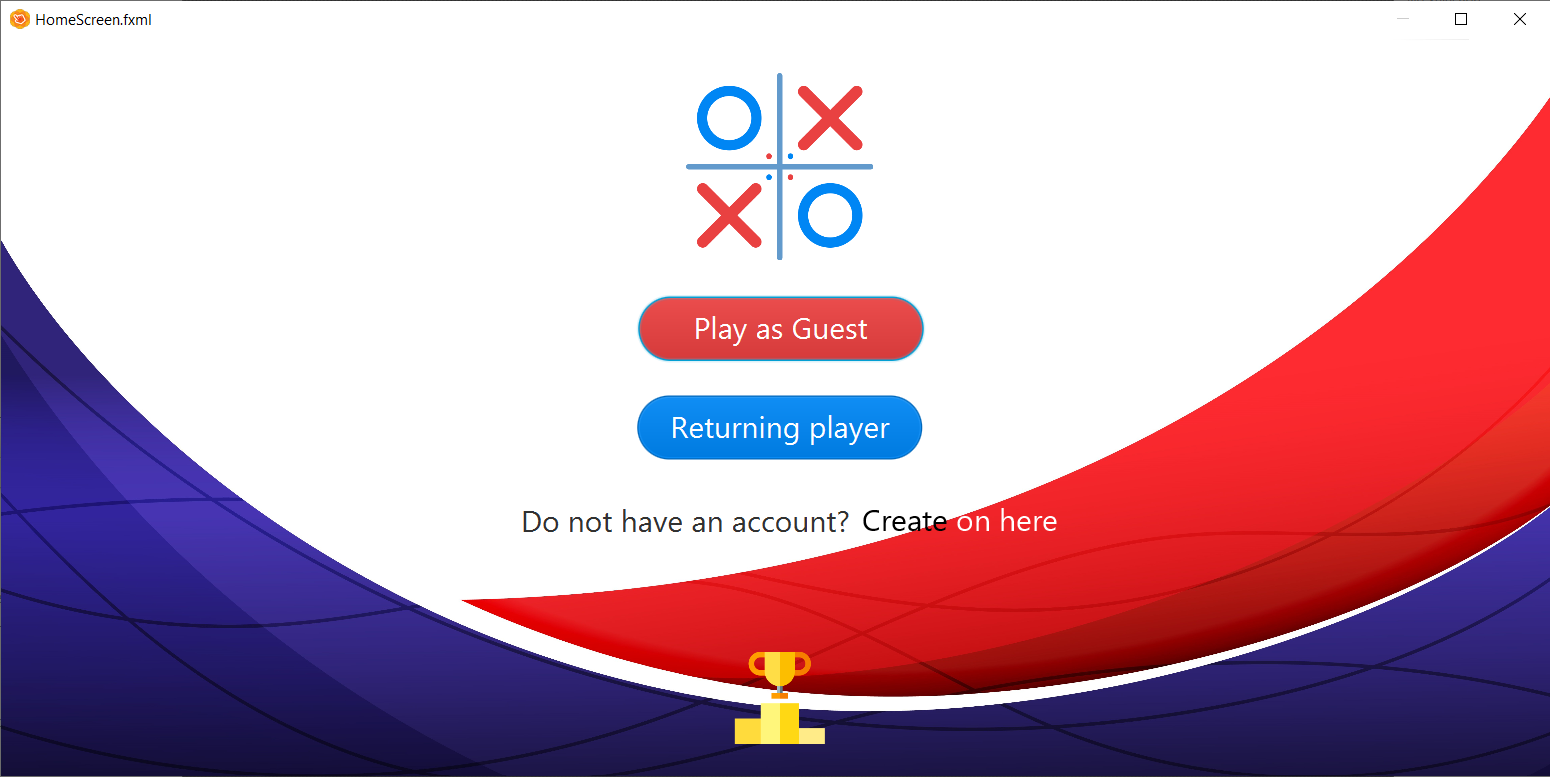


## **7. GUI Design**

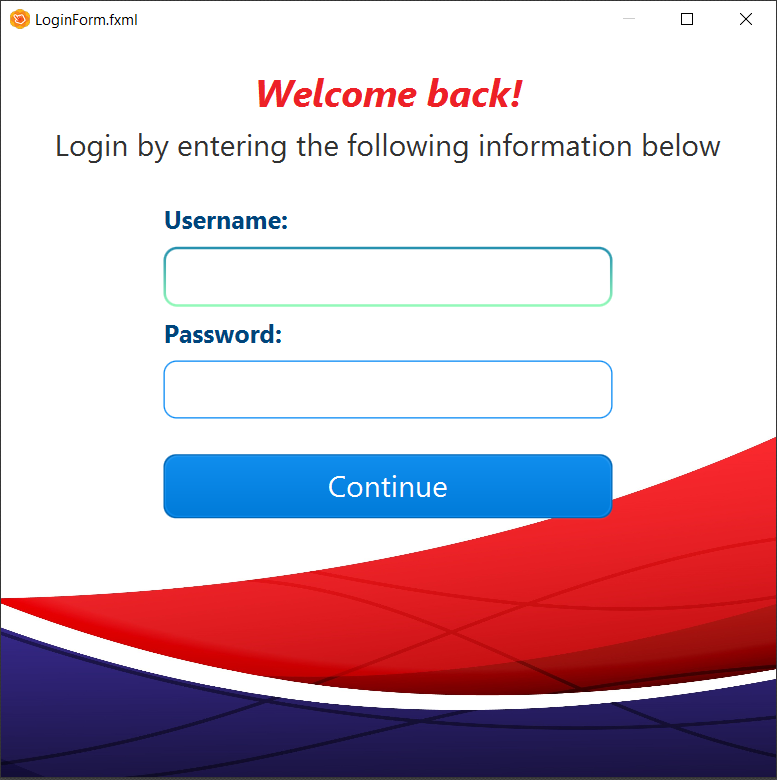
For GUI, we use JavaFX and SceneBuilder to design the game’s graphical interfaces

There are a total of 7 screens:

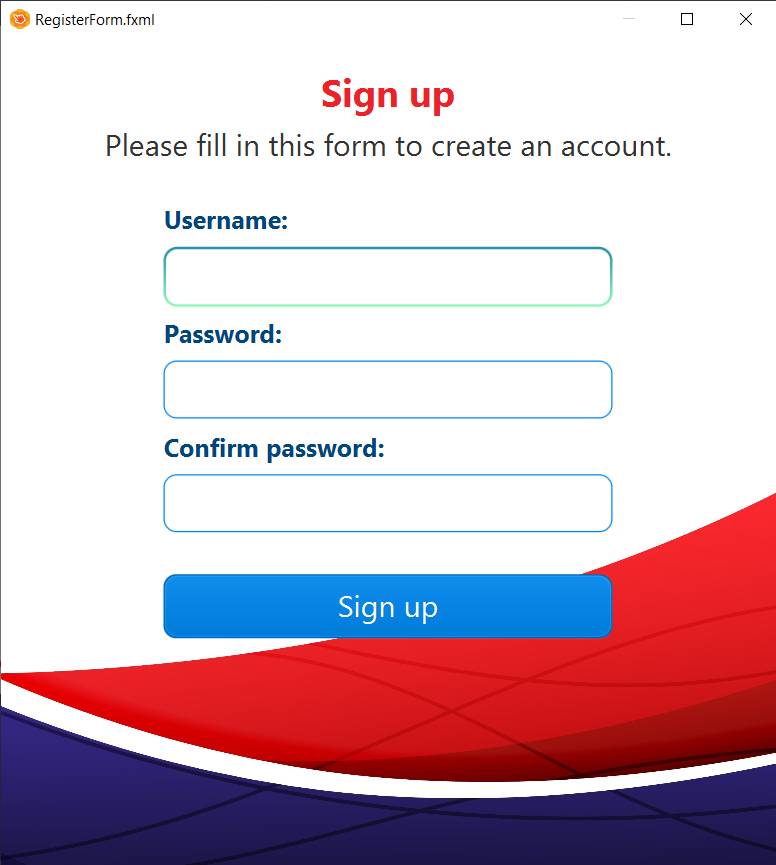
* Home Screen:



* Login Screen:



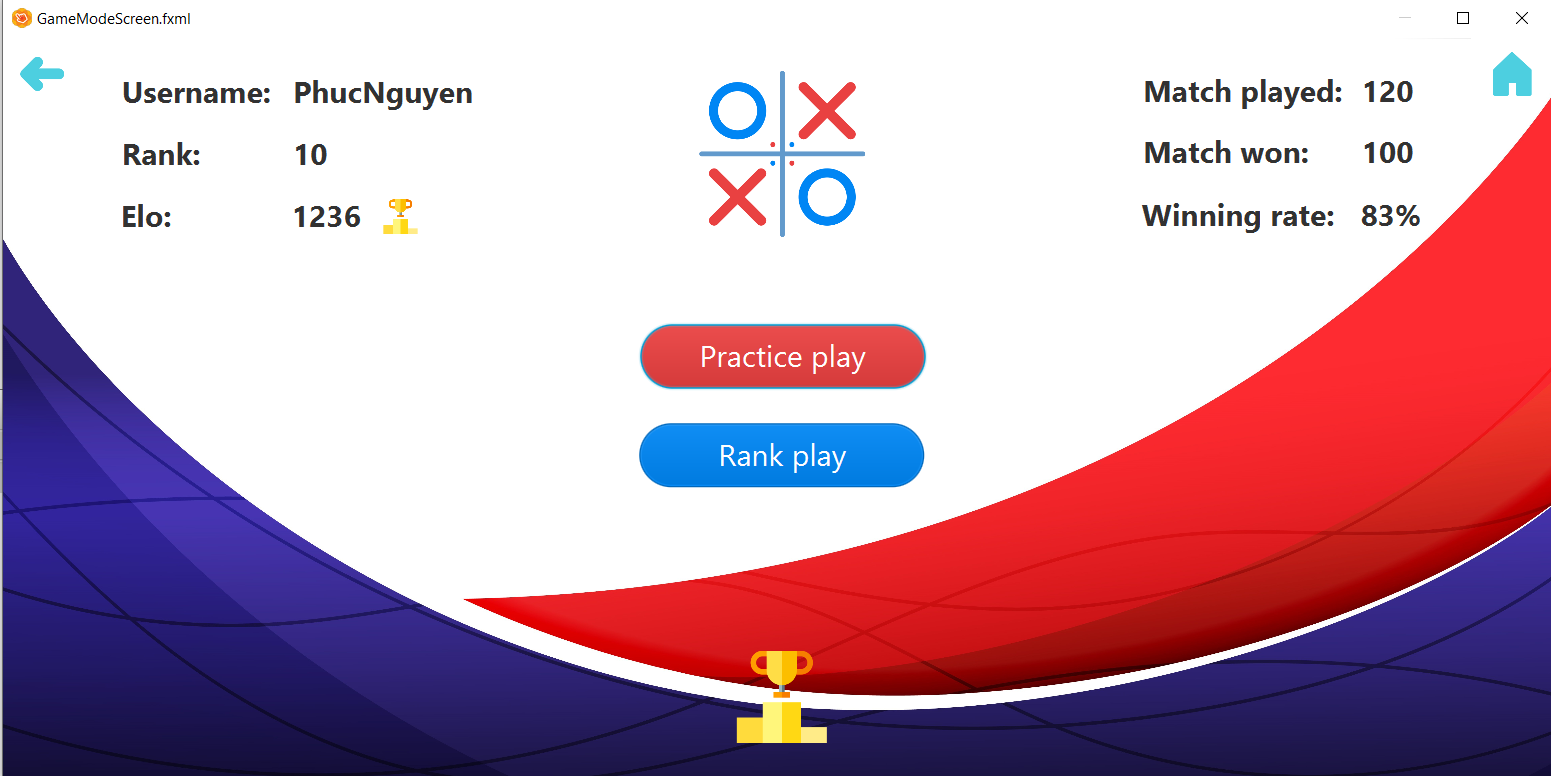
* Register Screen:



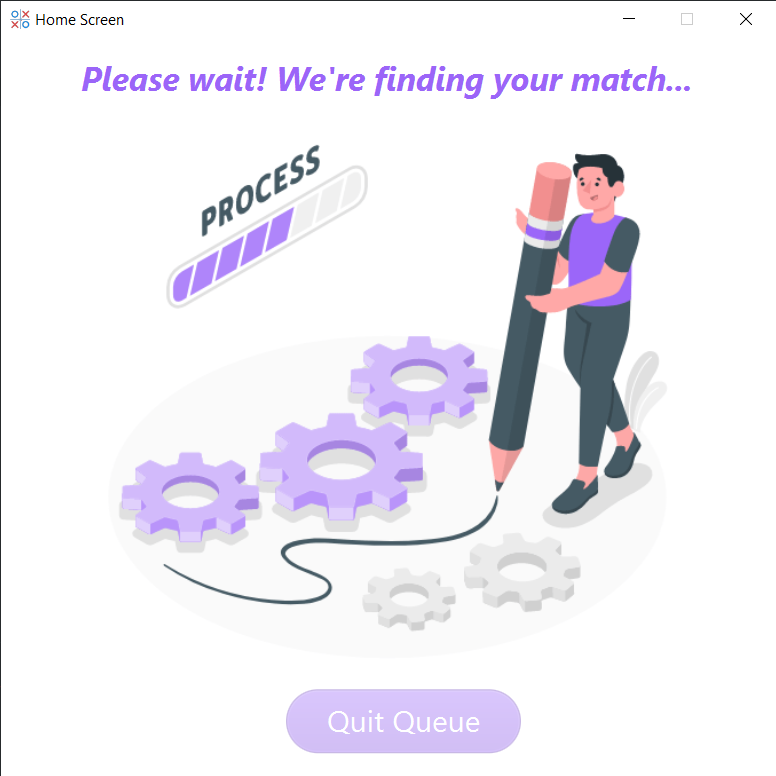
* Leaderboard Screen:



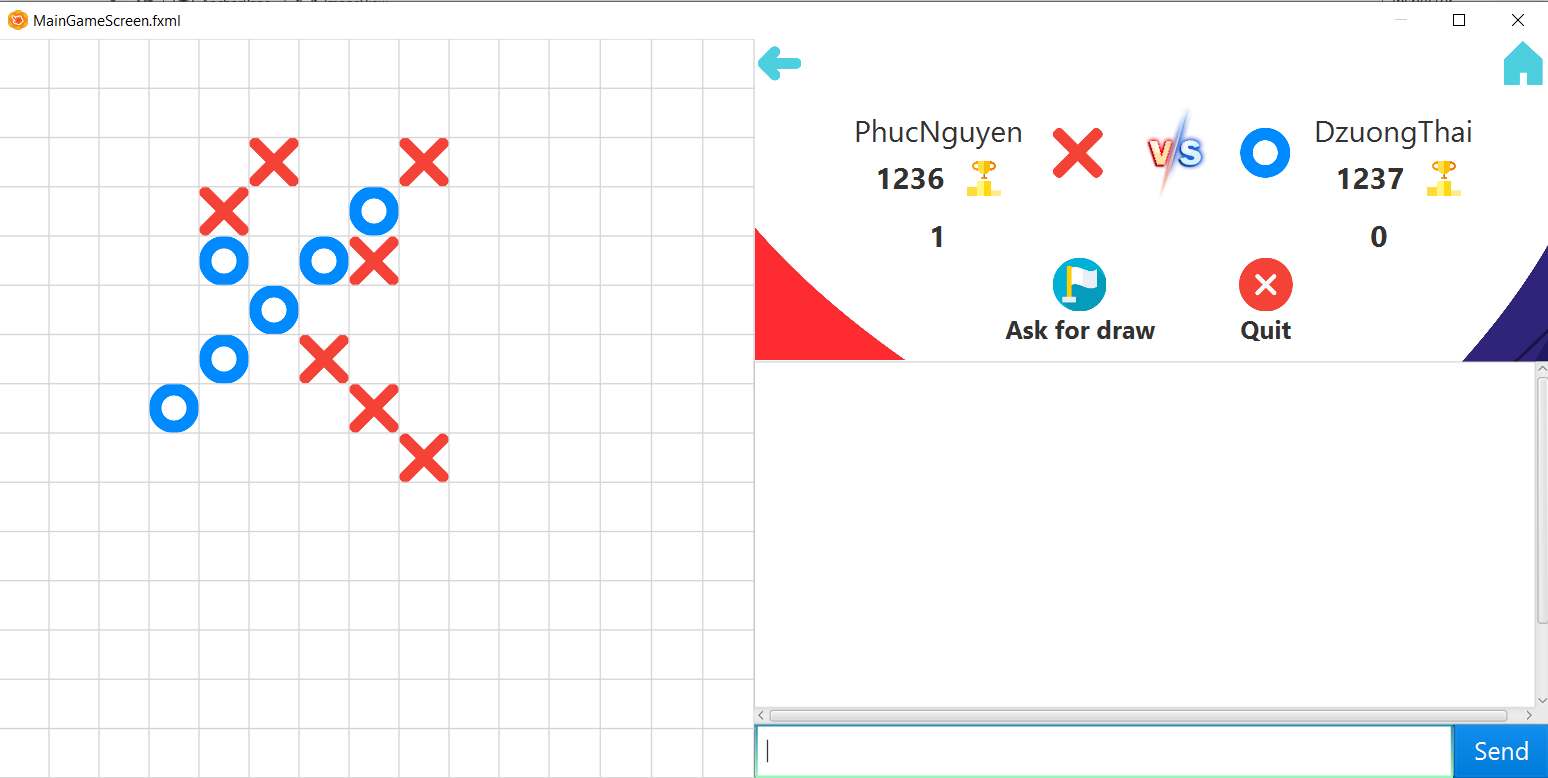
* Game Mode Screen:



* Waiting Screen:



* Main Game Screen:



# **PART III: IMPLEMENTATION**

## **Server**

## **Client**