Tic-Tac-Toe Game!

Author: Qiuyi (Chew)
Mail: qiuyifeng1997@outlook,com

Background

This software is an implementation of classic tic-tac-toe game[1] which is designed in a distributed manner. The system provides the ability for multiple players to connect and play tic-tac-toe, including advanced features like chatting and ranking.

The system has 2 independently deployable parts: the server and the client. In this typical server-client architecture, players use a client with GUI to connect to the server for matching, playing and chatting.

System Architecture

Overview

In this server-client pattern system, servers and clients communicate with each other through a protocol based on Java RMI[2]. Clients can and only can find the remote object of servers by the location(URL) of the registry, which makes a server unique. Every client has a username, and we assume that the username of different clients cannot be the same. Thus the server can distinguish clients with usernames.

A client needs to login to the server and waits for a match.If there are at least 2 players login, the server matches every 2 players for a game until the number of logged players is less than 2. After matching, a game will automatically start and 2 players take turns to make moves until an endgame condition meets, that is either a winner produced or there is a draw produced. The player who plays first is chosen randomly by the server. After an endgame, players can choose to rematch or exit.

Protocol

The protocol is defined by the RMI interfaces of the server and the client. The interfaces are shown in figure1. The function signature may not be identical to those in the actual source codes because the author omits some implementation details for readability. The figure shows that the return value of server interfaces is a `Response` object, which contains a status code, an error message and the actual data. The server always tries to return a `Response` Object, setting the status code to an error code if there is an error happening in the process that requests. Clients can display the error message for players to better understand what happened in the server. The definition of status code is shown in table 2.

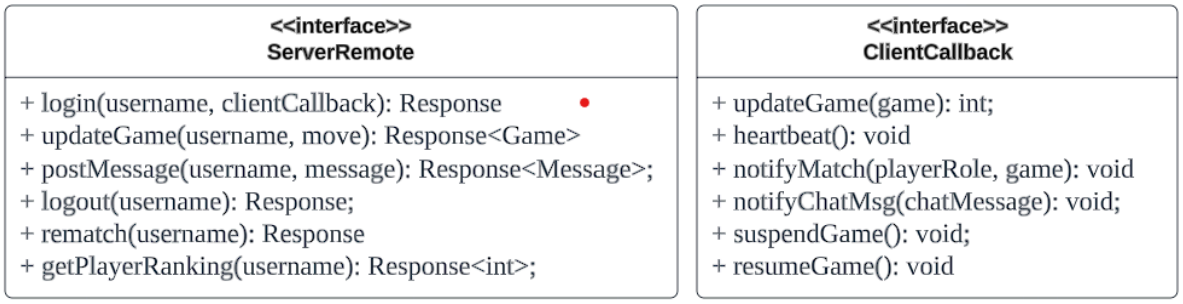

Figure1 RMI Interfaces designs

Status Code	Meaning
200	Success
400	Bad request
500	Internal error in server

Table 2 Status Code

The Server can also invoke the methods defined in the `ClientCallback` interfaces, representing the methods that client provides for server to callback. For example, if the game is updated by a move by a player, the server will call `updateGame` for both clients to update the status of the game distributedly. To avoid error, all the methods are assumed to be idempotent so that if a method is called multiple times with the same parameter, the effect will only be once.

Server Overview

The architecture of the server is shown in figure 3.

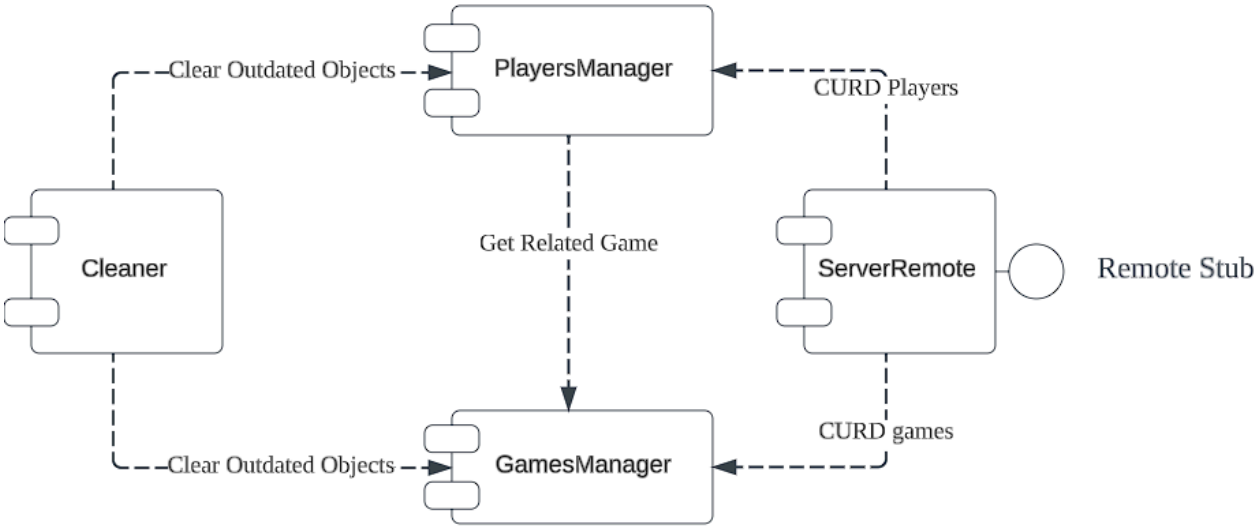

Figure3 Server Architecture

As the figure 3 illustrates, there are 4 main components in the server:

- *ServerRemote* is responsible for managing the RMI registry and provides the remote object of the server for clients to connect.
- *PlayersManager* manages the player resources, including the usernames, status and player rankings.
- *GamesManager* manages the game resources and updates game state after every move players make.
- *Cleaner* is the timed task that removes outdated resources to keep the system space-efficient. We will cover this in the next section.

Note that the stub is located by a unique location (ip + port), which means 2 servers cannot run on the same port of a machine.

The server handles the chat message by broadcasting it to both clients (if alive). The server itself does not store any chat messages.

After an endgame, the player can choose to rematch a new game. If the player does that, the server would move the player to the active player list to wait for a match. The active player list is simply a queue that pop the continuous 2 players to make a match.

In summary, key features implemented in servers:

- Actively register and match players to start games
- Update game states after players make moves and calculate endgames.
- Optionally rematch the players in a finished game
- If a player quits in a game, the alive opponent will be declared winner.
- Handle chat messages in the game.
- Player ranking system.
- Timed heartbeat checks and allows reconnection for disconnected player

Client Overview

The client provides a GUI application for human players that is powered by JavaFX, shown in figure 4.

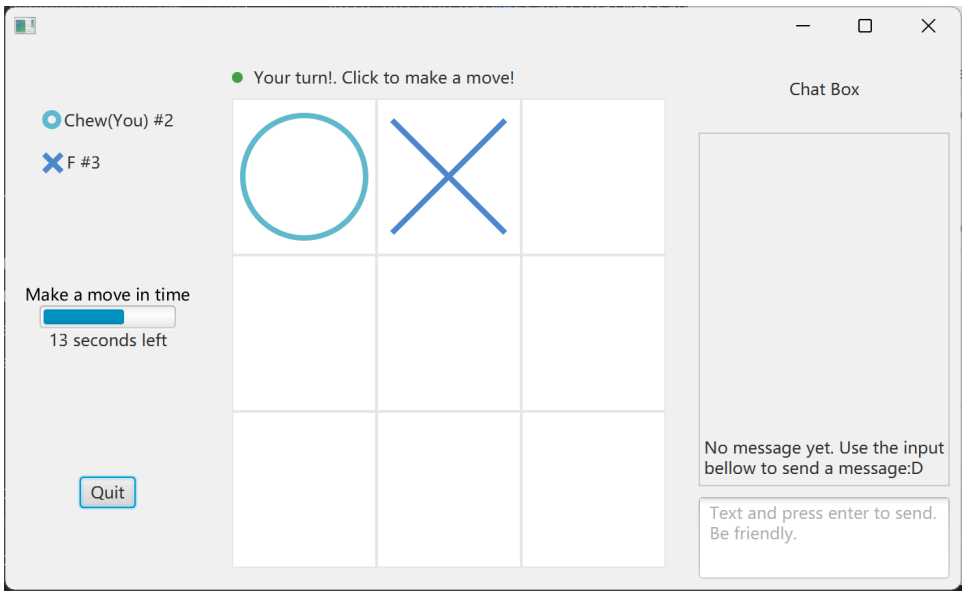

Figure 4 Client GUI

The actual gameplay of tic-tac-toe displays in the centre. There is a chat space on the right of the interface that allows users to chat with each other. The game information including the names of players and the role they play and the timeout counter are displayed on the left of the board. If the counter ran out of time, the client would make a random move on a vacant tile. The ranks of players are displayed after the username with format “#rank”. The “Quit” button on the left bottom side allows the user to quit the game. If the user quit during a game, the opponent will automatically be declared as the winner.

As shown in figure 1, the client relies on the callback messages that are sent by servers. Besides, servers and clients must maintain the heartbeat signals. If a heartbeat signal is failed to send, clients would regard the server to be unavailable and quit the game. The part

of design will be covered in the next section.

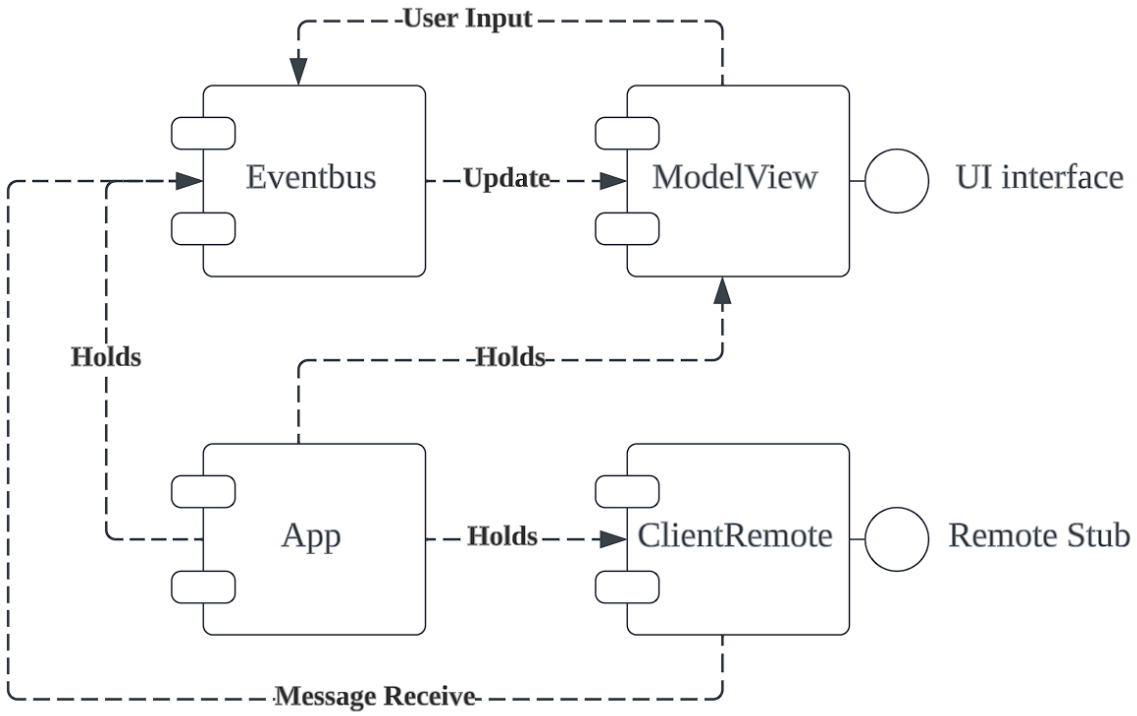

Figure 5 Client architecture

The architecture of the client is visualised in figure 5. The *App* component is the manager of all components. *Eventbus* is a publish-subscription model that handles multiple events in the client, including user inputs and server callbacks. *ModelView* is the component that provides the user interface and subscribes the events in Eventbus. *ClientRemote* manages the remote stub. When it receives any messages from the server, it would publish to the Eventbus. For example, if the ClientRemote accepts an *UpdateGame* message from the server, it submits to the Eventbus. Then all the UI components (Here “UI Components” means JavaFX components) in ModelView which subscribes to the *UpdateGame* message will update its state and rerender. This pub-sub pattern enables the application to decouple GUI from other parts of the client.

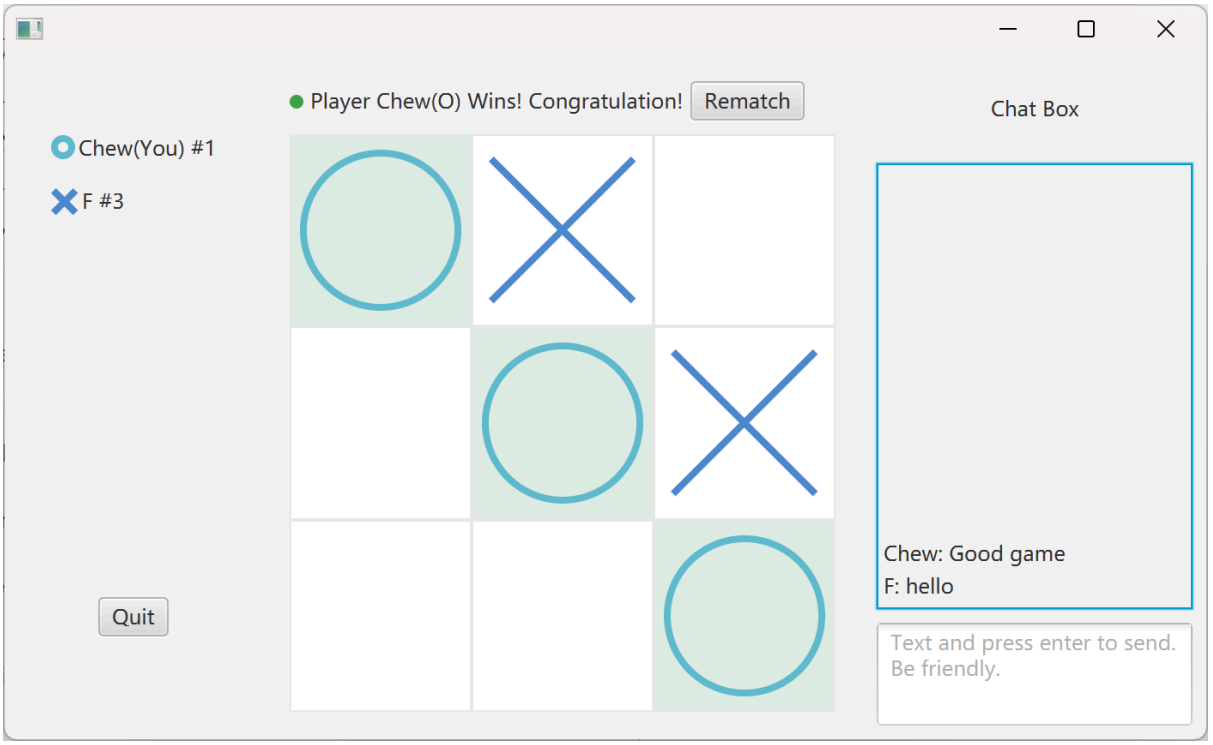

Figure 6 Endgame

Figure 6 gives an example of an endgame. The GUI displays the winer and players can examine the board. The chat box is also available in an endgame, as long as players have not left the game. There is a rematch button on the top where players can send a rematch request to the server to find and start a new game.

Key features implemented in clients:

- Visualised Tic-tac-toe gameplay
- 20 seconds timeout mechanism
- Messages and game state display
- Check latest heartbeats and decides the existence of the server

Key Components Designs

HeartBeat Checks and Reconnections

As mentioned in the previous sections, clients and servers must maintain a regular heartbeat check. The HearBeat checks happen every 3 seconds, starting from the server sending a callback message to clients (call the heartbeat() function on remote stub in figure 1). The heartbeat callback method does not have any parameters or return value, reporting only the connectivity of the server-client communication channel. That is, unless there is an exception in the function invocation, the server would regard the clients to be alive. Otherwise, the server would see clients as disconnected. From the state view of the player on the server side, the protocol is shown in figure 7.

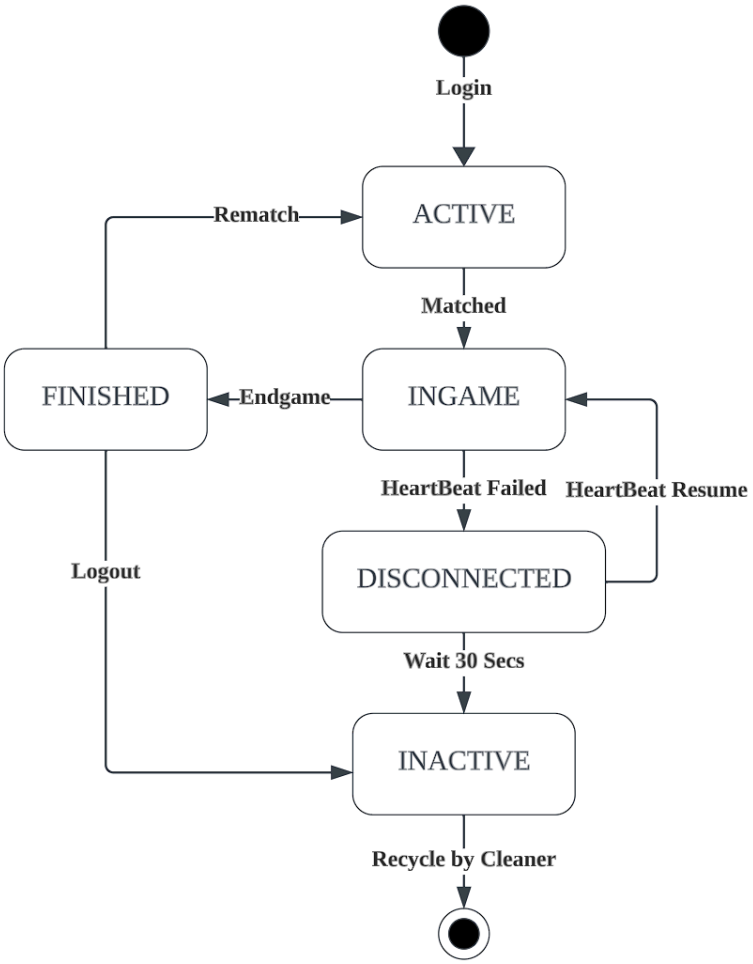

Figure 7 Player State Change

The player will be assigned the ACTIVE state after login and INGAME after a successful match. If the player is INGAME state and the heartbeat check fails, the server will put the player to a DISCONNECTED state and wait for 30 seconds. In the meantime, the server will call *suspendGame()* on the callback of the live client. If the client manages to reconnect within 30 seconds, the game will resume by calling *resumingGame().* Conversely, if the client failed to manage to reconnect, the server would declare the alive player to be the winner and end the game.

From the perspective of clients, things are rather simple. Clients will do 2 things in the heartbeat check mechanism:

- After receiving the suspendGame message, it stops the client-side timeout countdown (left side in figure 4), and resumes on receiving a resumeGame message from the server.
- If clients do not receive a heartbeat check request from servers for 10 seconds, it assumes the server to be dead and exits the game in 5 seconds.

Resource Cleaner on Servers

The implementation of the server shown in figure 3 tells the existence of a cleaner component. It is responsible to check the freshness of objects in the server and remove

outdated ones, typically, the player resources and game resources. For example, when both players of a game are inactive and the game is finished, the cleaner will remove the outdated objects in PlayersManager and GamesManager.

This design keeps the server space-efficient if there are too many users using the system. And also, the cleaner removes the responsibility for PlayersManager and GamesManager to check the existence of resources and remove them, which would likely cause errors by concurrency. Instead, the removal of outdated resources are done by a timed, single task and guarded by a single lock, which makes this design also improve the concurrency safety. The cleaner is timed in a 20 seconds interval by default, which is an assumed parameter and not well examined yet.

Player Rankings

The server holds the marks and calculates the updated ranking after every endgame. If a user wins a game, it gains 5 points. Conversely, the user loses 5 points while losing a game. If there is a draw, both players earn 2 points. The ranking of a player is primarily based on its points. If the points of two players are the same, they will be ranked by the lexical order of their player name.

The implementation of the ranking component is by 2 data structures: one hashmap to save the points, one linked list represents the rank of player. There will be an update on the linkedlist to calculate the latest rankings and update the linked list. This update operation is locked to prevent the corruption of data caused by concurrency.

Critical Analysis

Pros

- Space-Efficient. The server benefits from the design of the cleaner to recycle space resources timely.
- Responsive GUI. The render of the GUI on clients based on the pub-sub mechanism on the client side. That means the client will only perform necessary updates after receiving an event.
- Fault Tolerance. The clients and servers constantly check the heartbeat signals. Servers allow users to reconnect to continue a game if there's an accidental crash in clients. And the client gracefully exits the game if a server crash is detected.

Cons

- The communication protocol uses RMI, which makes it impossible for implementation of clients or servers in other languages.
- The server uses its own specific location for the RMI registry. Thus it's hard to scale. One more desirable approach is an independent registry of servers.
- The calculations of rankings are made on every endgame, which is not feasible if there is a huge amount of ongoing games and it deteriorates the performance of servers by blocking the update of player points. It can be fixed by adopting the

calculation to be a timed and off-line manner. For example, calculating the rankings on every minute based on a copied history data on this particular minute.

Conclusion

This report describes a system that allows remote players to play classic tic-tac-toe games. In addition to basic gameplays, the system implements features including chatting, client-side timeout, reconnection for disconnected players and so on. The protocol of this system is based on Java RMI. Heartbeat checking mechanism is employed in this system to make servers and clients aware of the existence of each other and make responsible actions. The design and implementation of this system has both pros and cons as described in the critical analysis section.

Reference

[1] Tic-tac-toe (2023) Wikipedia. Available at: https://en.wikipedia.org/wiki/Tic-tac-toe (Accessed: 05 October 2023).

[2] *An overview of RMI Applications* (no date) *An Overview of RMI Applications (The JavaTM Tutorials > RMI)*. Available at: https://docs.oracle.com/javase/tutorial/rmi/overview.html (Accessed: 05 October 2023).