Operating Systems' Final Project Report

Amir Faridi - 610300087

February 5, 2024

1 Introduction

This report outlines the process of implementing a multiplayer Tic Tac Toe game with options for playing on a 3×3, 4×4, or 5×5 grid. The game is designed to allow two players to compete against each other, taking turns to place their marker on the grid, with the goal of getting three (or more, depending on the grid size) of their markers in a row, either horizontally, vertically, or diagonally. I used *colorit* library to make the game more visually appealing. To install the library, run the following command:

```
pip install color-it
```

2 Implementation

The project is implemented in Python, using *threading* and *socket* libraries to create a server-client architecture. The server is responsible for managing the game, while the clients are responsible for displaying the game and sending the user's input to the server.

Class Board

This class is responsible for managing the game board, including the grid size, the current state of the board, and the methods for checking the game's state. The main methods of this class are:

- $is\_full()$: checks if the board is full
- $is\_valid()$: checks if the user's input is valid
- $action()$: updates the board with the user's input
- $check\_winner()$: checks if the game has a winner. To easily check for a winner, I used $all()$ built-in function to check if all the elements in a row, column, or diagonal are the same. This method made the code more readable.

- *check_tie*(): checks if the game is a draw
- *get_wiiner*(): returns the winner of the game

Class Client

This class is responsible for managing the client's connection to the server, and for displaying the game to the user. The main method for this class is *handle*(), which is responsible for sending the user's input to the server, and for receiving the server's responses. The communication between the client and the server is done using the *socket* library. The messages are sent and recieved as json objects in a certain format which is explained in the next section.
The game is made simple and the the only inputs that the user needs to provide are the row and column of the cell that they want to place their marker in, and the initial input for the size of the grid.

Class Server

- *__init__*(): initializes the server. It creates a socket and binds it to the host and port. There is also a *waiting_list*, which is a dictionary that maps the size of the grid to a list of clients that are waiting for an opponent to play with (which is obviously zero or one client). And there is an associated *mutex lock* for the *waiting_list* to prevent possible synchronization issues with this list.
- *listen*(): listens for incoming connections from clients. When a client connects, the server creates a new thread to handle the client's requests.
- *handle_client*(): After calling this function for a client, the server asks for the size of the grid from the client. Then it checks if there is a client in the *waiting_list* with the same grid size. If there is, the server creates a new game for them and removes them from the *waiting_list*. If there isn't, the server adds the client to the *waiting_list* and waits for another client to connect.
- *play*(): This function is called when there are two clients in the *waiting_list* with the same grid size. It creates a new game for them and starts the game. The game is managed by the server, and the clients are only responsible for displaying the game and sending the user's input to the server. The message passing between the server and the clients is done using json objects in a certain format. The message has several parts:
  - *board*: the current state of the board
  - *turn*: the current player's turn
  - *winner*: the winner of the game
  - *tie*: whether the game is a draw
  - *game_over*: whether the game is over

- *inp*: expectation of a input from the client
- *msg*: a message to be displayed to the client
- *invalid*: whether the input from the client is invalid
- *err*: an error message

The server works with these parameters and modifies them as the game progresses. The logic behind the game and the parts that are not explained in detail in the report are obvious.

3 Challenges

Other than the main idea of the project, that is, implementing a multi-threaded server-client architecture, the main challenge was the message passing between the server and the clients. At first, I used a simple string format for the messages, but it was not efficient and was hard to manage. Json objects made the code more readable and easier to manage.
Another approach to handle the board the logic behind the game was to place the main logic and the board section in the client side and check the consistency of the board in the server side. This approach is more efficient in the large and complex game, but I decided to go the other way to make the code more readable and simple since the game is not large and passing a board between the server and the client is not a big deal.