CS5700 Homework Assignment 04
Communicating States Across the Network

Name: Chun Sheung Ng
Title: Summary Document of a Networked Tic-Tac-Toe Game

Introduction

Tic-Tac-Toe is a classic two-player game played on a 3x3 grid. Players take turns placing their symbols ('X' or 'O') in the grid's empty cells, attempting to form a horizontal, vertical, or diagonal line with their symbols. This report describes the design principles and logic used in developing a networked Tic-Tac-Toe game, allowing two players to play the game remotely. The program runs in both client and server modes, implemented in C using the sockets API for communication. The game supports actions, such as starting a new game, maintaining player turns, updating the game board, determining the winner, and gracefully closing the connection.

Design Principles

2.1. Modularity and Reusability

To promote modularity and reusability, the program is divided into separate functions, each responsible for specific tasks. These functions include server_mode, client_mode, start_game, handle_client_turn, handle_server_turn, and check_game_state. This separation makes the code more readable, maintainable, and easier to debug.

2.2. Synchronization and Communication

The game uses TCP sockets for reliable communication between the client and server. A server socket listens for incoming connections, while the client socket connects to the server. Once connected, both players can communicate with each other by sending and receiving messages. The program ensures that both players have an updated view of the game state by synchronizing the game board after each move.

2.3. Error Handling

The program includes error handling for various situations, such as invalid input, invalid moves, and failed communication. This ensures a smooth gaming experience for the players. For example, when a player inputs an invalid move, the program informs them of the error and requests a new input.

Game Actions and Logic

3.1. Starting New Game with Clear Board

The game begins with a clear board, and the server and client negotiate player symbols ('X' and 'O'). The server sends its chosen symbol to the client, who updates player and opponent symbols accordingly. Once assigned, the server initiates the game by sending a start message to the client. Both players receive their symbols, ensuring synchronized game states.

3.2. Side-taking Negotiation between Players

Upon successful client-server connection, the server uses the server_start_game function to ask the client if they want to start first. The server's message includes instructions for the client to reply, specifying which symbol they choose ('X' or 'O'). Answering 'Y' lets the client start first as 'X', while 'N' passes the first move to the server and assigns 'O' to the client. This negotiation empowers players to select their preferred side and symbol.

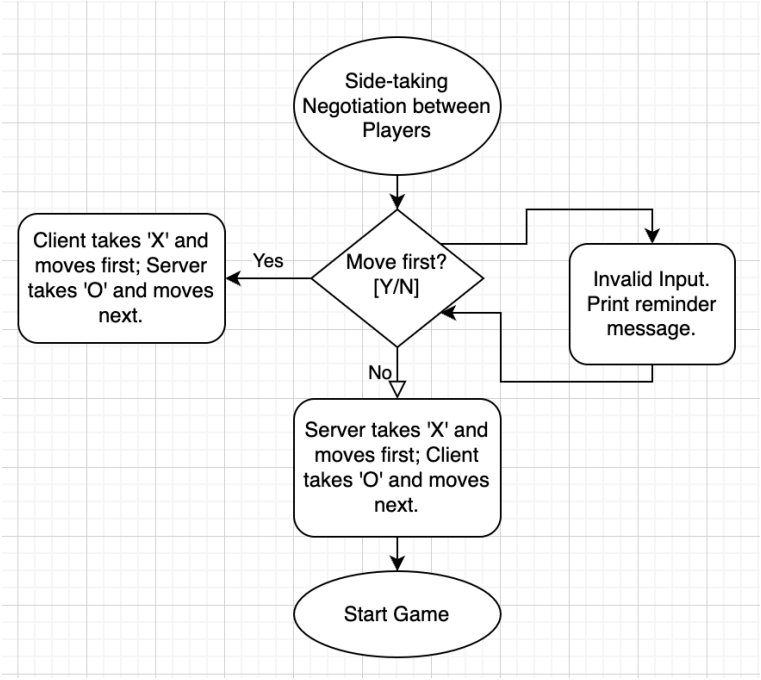

Figure 1: A flowchart showing the process of starting a new game

3.3. Maintaining State of Player Turns

The game state tracks turn information, switching between client and server turns after each move. During the server's turn, it waits for the client's move, validates it, and updates the game board. Similarly, during the client's turn, it sends its move to the server for validation and board updates. This process ensures that players only make moves during their turn, and the server manages board updates based on player input.

3.4. Communicating and Updating the Game Board

When a player makes a move, the updated board is sent to the other player. The receiving player parses the message and updates their board accordingly, ensuring consistency between both players. The game board is represented as a 3x3 character array, making it easy to parse and update. The server and client messages differ: the client sends two integers representing move indexes, while the server parses moves and updates the board for both sides. The client only sends move information, allowing it to play with other server-side programs by simply connecting and following game rules. The server primarily handles message parsing and board updates.

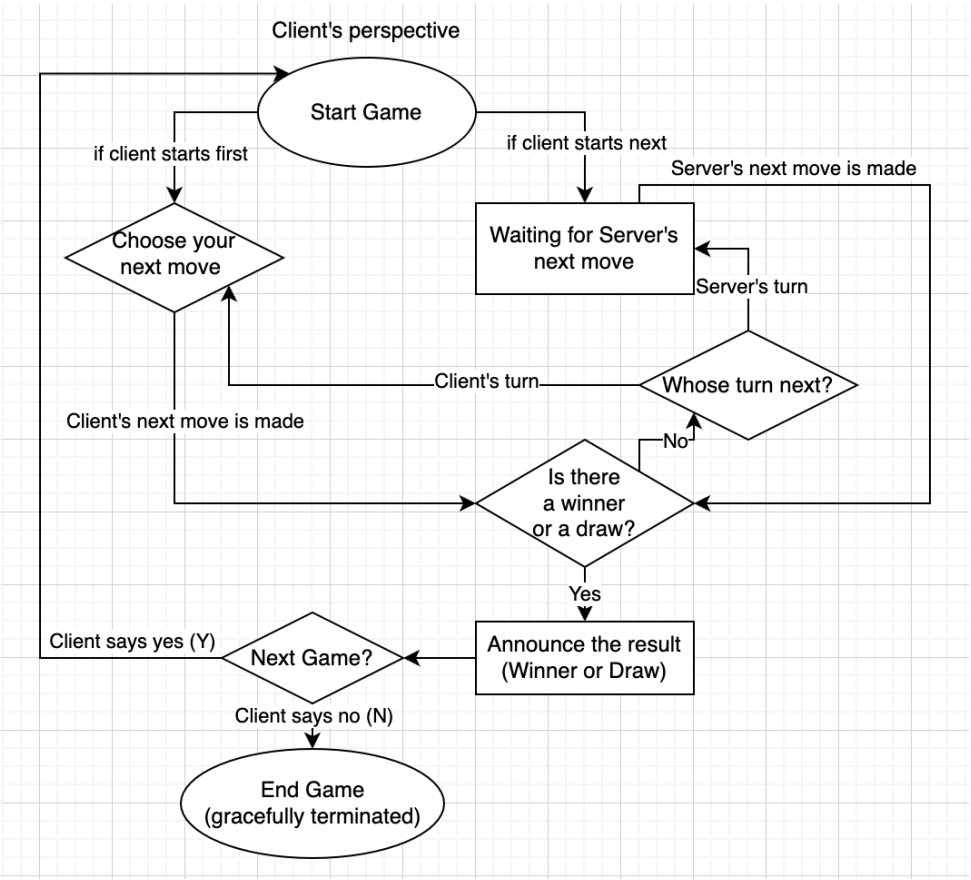

Figure 2: A flowchart showing the process of communicating and updating the game board

3.5. Rewriting and Aligning the Game Board

The server takes care of rewriting the game board based on its own input and the client messages received. For aligning the game board state on both side, the server sends a message string with the board information that can be parsed into a 3*3 grid at client side. When the client receives data from the server, the program uses helper functions to check if it is a board or some messages. If it is a string of board information, the board state at client side updates with the 3*3 grid parsed with the new data each turn.

3.6. Determining the Winner

The game checks for a winner after each move. If a player achieves a winning condition (a horizontal, vertical, or diagonal line of their symbol), the game announces the winner and ends the session. The check_winner function finds the winner by iterating through the board's rows, columns, and diagonals. If no winner is found and the board is full, the game declares a draw.

3.7. Replaying and Resetting the Game

After a game concludes, players can choose to start a new game or exit the program. This replay functionality is implemented through the server_request_new_game and client_reply_new_game helper functions. These functions are called when a winning player or draw is detected in server_mode and client_mode. The server prompts the client to decide whether to play another game. If they agree, both sides clear the board, reset the game state, and begin a new game. If not, the server sends a closing message, and both sides exit gracefully. This design enhances user-friendliness, allowing for seamless gameplay.

3.8. Ending and Closing the Program Gracefully

The program is designed to close gracefully upon completion or upon receiving a termination request from the user. A proper termination sequence is initiated when the players decide not to start a new game, or when they enter a specific command (e.g., Ctrl+C). The server sends a closing message to the client, acknowledging the end of the game session. Both server and client implement error-handling mechanisms to account for unexpected disconnections or errors during gameplay. If either side encounters an issue, they print an error message to inform the user about the issue and exit the program, ensuring a clean termination.

To further enhance the program's stability, resources such as sockets are properly closed before the application exits. This prevents potential resource leaks and ensures the operating system can reclaim the resources immediately upon termination. By incorporating these features, the tic-tac-toe program can exit smoothly without leaving any unresolved issues, thus providing a reliable and user-friendly experience.

Conclusion

The networked Tic-Tac-Toe game presented in this report exemplifies a thoughtful design that emphasizes modularity, reusability, synchronization, error handling, and an engaging gaming experience. The game accommodates a range of actions, including initiating a new game, managing player turns, communicating and updating the game board, determining the winner, replaying and resetting games, and gracefully closing the connection. This implementation highlights a well-organized, maintainable, and efficient approach to the classic Tic-Tac-Toe game, allowing players to engage in a fun and competitive experience from different locations.