ADDIS ABABA INSTITUTE OF TECHNOLOGY

DEPARTMENT OF INFORMATION TECHNOLOGY AND ENGINEERING (SITE)

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

Distributed Systems Mini-Project

Project Title: Tic-TacToe

GROUP MEMBERS:

1.	Dagmawi Elias	UGR/2465/14
2.	Hanna Nigussie	UGR/9180/14
3.	Lidet Tadesse	UGR/8824/14
4.	Nobel Tibebe	UGR/5954/14

Submitted to: Wondimagegn Desta

Date of Submission: Feb 5, 2024

Multiplayer Tic Tac Toe Game Using WebSockets in Golang

1. Introduction

The game of Tic Tac Toe is a multiplayer game on a distributed system that uses Golang with the implementation of WebSockets for real-time session-based multiplayer gaming. This project serves as a practical example of how concepts related to a distributed system, such as **fault tolerance**, **synchronization**, and **replication**, can be implemented in an interactive environment.

Real-time interaction allows multiple players to join, compete, and interact using WebSockets. It features **low latency** with strong error handling, achieving session persistence so that if any player disconnects, the game can continue with the remaining players. It also offers a scalable architecture capable of handling multiple independent games running simultaneously.

2. Key Components of System

The main components of the proposed project include:

- **Golang WebSocket Server:** Enables real-time communication between clients, session-based multiplayer gaming, and implements distributed synchronization, replication, and fault tolerance.
- **Web Interface:** Provides a user-friendly interface for players to start, join, and interact with games, utilizing HTML, CSS, and JavaScript for an appealing design.
- **Session-Based Game Management:** Each game receives a session ID, allowing multiple games to be played concurrently, with reconnection logic for handling unexpected player disconnections.
- **Real-Time Synchronization Mechanism:** Updates game states in real-time across all connected clients, ensuring consistency in game logic execution across multiple nodes.

3. Features and Functionality

3.1. Real-Time Multiplayer Game Playing

WebSockets are implemented for two-way communication between the client and the server.

Players can engage in a traditional game of Tic Tac Toe, with changes updated in real-time.

3.2. Session-Based Game Management

- **Session ID:** Each game has a unique session ID.
- **Concurrent Instances:** Multiple instances of the same game can run independently.
- **Joining Ongoing Games:** Players can join any ongoing game using the session ID.

3.3. Options to Host & Join Game

- **Hosting a Game:** A player can create a new session and share the session ID with an opponent.
- **Joining a Game:** The second player can join the current session using the provided session ID.

3.4. User-Friendly Interface

A simple and responsive design is used for the following reasons:

- **Real-time Visualization:** Displays the game board in real-time.
- **Dynamically Track Status:** Keeps track of the game status.
- **Smooth Restart or Exit:** Allows for easy game restarts or exits.

3.5. Distributed Synchronization & Replication

The game data is maintained by the server and synchronized to all clients linked to a session.

- **State Replication:** Ensures game data remains intact even in case of failure.
- **Server Processing:** Every game move is processed on the server and relayed to clients.

3.6. Robust and Fault-Tolerant System

- **Crash Recovery:** The system continues running even if a client crashes.
- **Session Persistence:** Allows players to reconnect and resume their game.
- **Graceful Error Handling:** Manages invalid moves without disrupting game flow.

4. System Requirements

4.1. Functional Requirements

- **Multi-client Support:** The system supports multiple autonomous clients interacting simultaneously.
- **Distributed State Management:** Maintains game state across a set of nodes.
- **Fault Tolerance:** The system continues running if one of the clients crashes.

4.2. Non-Functional Requirements

- **Programming Language:** Implemented in Golang.
- **Synchronization & Replication:** Ensures consistency in game state across distributed sessions.
- **Fault Tolerance:** The system remains operational during client crashes.
- **Systematic Testing:** Effectively tested at unit and system levels for reliability.
- **High Performance:** Operates efficiently under high loads.

5. System Architecture

5.1. High-Level Architecture

The architecture of the system will be a **client-server model** using **WebSockets** for real-time data exchange:

WebSocket Server (Backend - Golang):

- Session management of all game sessions.
- Client connection, disconnection, and message exchange.
- Distributed state management to maintain data consistency.

Web Interface (Frontend - JavaScript, HTML, CSS):

- Provides an interactive platform for the players.
- Presents game updates in real time.
- Input validation and enhances UI.

Clients (Players - Web Browser):

- Communicate over WebSockets.
- Interact with the server to make moves and get updates.

5.2. Communication Workflow

- Player Makes a Move: The client sends a WebSocket message to the server with the move data.
- Server Processes the Move: Performs the movement.
- Updates game state.
- Server Broadcast Update: Sends the updated board state to all connected clients.
- Game Completion Handling: Checks for win conditions or draw.
- Informs the players and, if wanted, allows for a replay (play again).

6. Fault Tolerance and Recovery Mechanisms

6.1. Session Persistence

- If a player disconnects, the session does not lose its state.
- Players reconnecting are able to resume their game at the same point.

6.2. Distributed Synchronization

- For any session, all clients see the same set of game updates.
- Avoids race conditions, guarantees state change uniform.

6.3. Client Unexpected Crash Handling

- The system can handle disconnection detection on the client side.
- It would not make a difference even when one player has gone out.
- A new player can be added if needed.

7. Installation and Setup

7.1. Clone Repository

Run the following command

- git clone https://github.com/Donelongo/distributed-system-game-tic-tac-toe.git
- cd ./distributed-system-game-tic-tac-toe
- Code .

7.2. Run WebSocket Server

Open project terminal at root directory.
Run command

- go run cmd/main.go

Server is listening at port 8080.

7.3. Open the Web Interface

Open website and search - http://localhost:8080/

8. Testing

8.1. Unit Testing

- Handling WebSocket messages.

- Validation of move.
- Game state synchronization.

8.2. System Testing

- Consistency across clients.
- Disconnections and reconnections of clients.
- Stress test with many game sessions in parallel.

9. Known Issues and Future Enhancements

9.1. Future Improvements

1. AI Opponent: Implement single-player with AI.
2. Game Stats: Track players' wins, losses, and draws. Streak feature implementation is also possible.
3. Improved UI: Enhance the animations plus add sound effects.
4. Server-side failure handling

10. Conclusion

This project develops an interactive, real-time, multiplayer game that uses WebSockets in Golang. The system is designed with:

- Distributed state
- Fault tolerance
- Session persistence

This structure makes the system strong and scalable.

Future Enhancements

The game can be made more interesting by:

- Adding AI opponents
- Implementing session recovery
- Enhancing UI components

This project may serve as a valuable learning exercise on real-time communication, WebSockets, and distributed system design in Golang.