Symbols Game: A Decentralized Multiplayer Tic-Tac-Toe

DS Project Report, Group 15

Xinyang Chen, Runjie Fan, Sergei Panarin, Axel Wester

Table of Contents

Introduction

This project aims to implement ‘Symbols Game’, a decentralized multiplayer tic-tac-toe, where multiple (2-5+) players take turns trying to mark their symbols on a shared board, and win by having multiple symbols in a row.

In this implementation of the game, multiple nodes run the same software, each node maintains its own game state, takes turns coordinating the game, and progresses the game by passing state changes with messages through sockets.

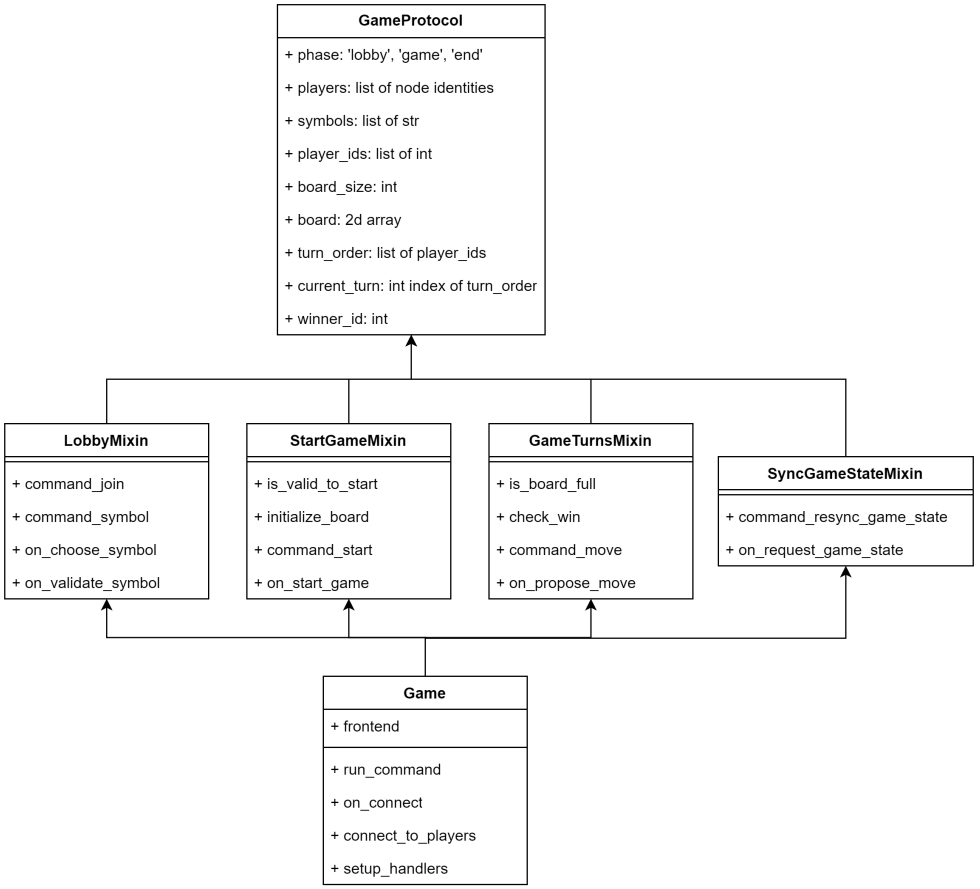
Within the core of this implementation is implementing turn-based coordination between players. This type of architecture can also be used to easily host other types of turn-based games, like poker.

Design

Architecture

Each node runs an identical version of ‘symbols game’, a Python program. The program uses Pydantic and JSON to serialize and deserialize message objects, sends and receives them over the network using the native socket library.

Game logic is implemented as a base protocol that contains the crucial game states, various mixin classes that implement different phases of the game, and the final Game class that handles frontend interactions.



Implementation using Mixins

Game has two frontends, one being GUI and one being CLI. Both of these frontends respond to user inputs by calling various game commands, and refreshes the displayed state when game logic has demanded a redraw (usually at the end of each command).

We also collected sent messages between nodes, for local development we just used local log files, but for demo and deployment we deployed a quick Filebeat - ElasticSearch - Kibana pipeline to ingest and visualize these logs in real time.

Messaging

Within each node, a server socket is spawned to accept incoming connections from other nodes. The node may also connect to other nodes with socket clients. In game, a node would have one connection to each node on the network (except itself), and that connection can be an incoming or an outgoing socket connection, in terms of the implementation this does not matter. The program keeps track of all connections using a ConnectionStore, which indexes all the socket objects using the name of the node it connects to.

To send a message, the program simply serializes the message into JSON, and then sends it to the other end using the socket.send method. To receive a message however is less straightforward, we ended up using an on-message callback architecture, where upon receiving a message, the connection would dispatch this message based on its type to the registered callback function, and within the callback function, relevant state updates can be carried out.

Each message is represented by a Pydantic model, with a mandatory field “method” denoting which kind of message it is, and other relevant fields for each message. We also maintain a central register of all methods and their corresponding message types, so we can automatically detect incoming message types and deserialize them before handing them over to message handlers.

Process

The game is divided into three phases: lobby, in-game, and end-of-game.

In the lobby phase, one node is the ‘host’ while other nodes connect to the host. This phase also sets up user preferences, like usernames, symbols, etc.

Then, when the ‘host’ selects ‘start’, the game transitions into the in-game phase, where all the game states are initialized based on host messages, and the first turn starts. Within a turn, the player making the turn coordinates this turn and other players observe the turn. The coordinator uses a two-phase commit protocol to update the board state and progress the game.

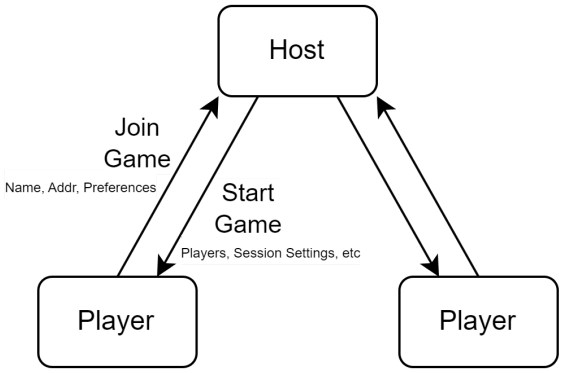
Once a win or tie condition has been achieved, the game then moves to the end-of-game phase, where results are displayed and connections are properly terminated.

All phases and phase transitions are implemented as separate mixins that handle the relevant behaviours and messages, therefore we’ll be following the same structures in the report.

Phase: Lobby (Discovery)

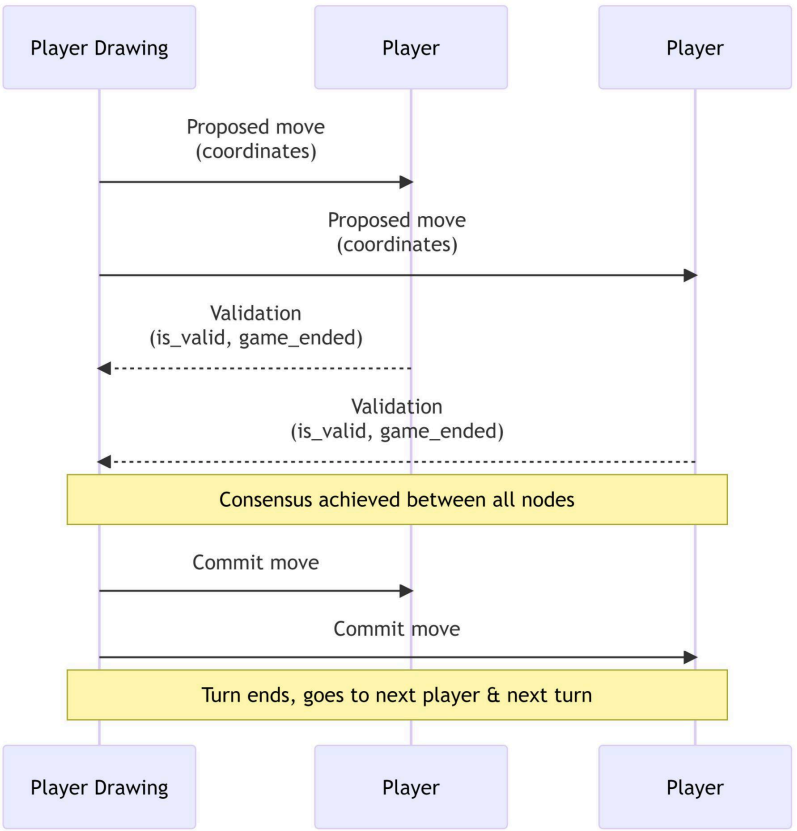
Lobby is created by the host. The host sets up the state. Other nodes connect using IP address and port. Each connecting node exchanges a "Hello" message with the host. Other nodes configure their preferences, in this case choosing symbols, which is validated by the host. The list of nodes is then established and prepared for phase transition.

Phase Transition: Start Game (Lobby - In-game)



The host verifies that all players have chosen their symbols and are ready to proceed. Then a StartGame message is broadcast by the host, it initializes the game state on all connected nodes. It assigns player ID, sets up the board based on the number of players, and defines the turn order randomly. Nodes connect only to peers with IDs smaller than theirs for better game coordination and less redundancy. Finally, the first player is notified to start their turn in the CLI.

Phase: In-game (Coordinator)



After the first player has been notified of their turn, the in-game phase begins. The game uses the randomly selected turn order to notify players when their turn is active. The player, whose turn it is currently, becomes the game coordinator for that turn. The coordinator is

responsible for proposing a move, validating it with other nodes, and committing it to the game state.

Once a player wants to make a move, they send a ProposeMove message to all other players with information about the move. All players must check and validate the move on the subject of its legality and respond with a ValidateMove message with either acceptance or rejection information. If all players validate the move, the coordinator sends a CommitMove message. The game board is then updated for all players.

Every time a coordinator commits a move, the coordinator makes a winning condition check. If the check is successful, the game transitions to the next phase. If not, the game continues and the next player is selected as coordinator. Each node updates their board based on the committed move. If the coordinator or any node fails, other nodes can recover the game state by reconnecting to active nodes and resynchronizing.

Phase Transition: Win or Tie (In-game - End-of-game)

The game transitions to the endgame state when two of the following occur:

- A player achieves a winning pattern. All symbols in a row, column or diagonal are marked as this person's symbol. Condition is validated through check-win method, which examines the board state after every move.
- The game state becomes a tie, which means that all cells on the board are filled, and no player meets the winning condition.

After this happens, a message is sent to all players which indicates the outcome of the game. A message is then displayed on the winner's screen. In case of a tie, a tie box is shown instead to all. After that connection is terminated between nodes.

Functionality

Naming

In the game, nodes are uniquely identified by the (ip_address, port) tuple. This port is the listening port of each node, and the actual connecting port might be different (for example, could be an unnamed client port, if this node is connecting to another node). Because of this, to make sure the naming stays consistent, upon connection the nodes exchange Hello messages, containing their listening port and address.

This setup also allows any node to be able to reconnect to all other nodes if needed, because all the node names are directly the reachable addresses of those nodes.

Fault Tolerance

Game implements fault tolerance during the in-game phase.

Firstly, as per recommendations given out to our project plan, each move proposal is retried by default, if another node does not respond within a specified time limit the coordinator will resend another proposal message and wait for a response again. This guards against temporary network issues, like packet drops.

Secondly, if a player drops out entirely during the in-game phase, that player can recover its state by using the –reconnect flag and reconnecting to any other player. This is accomplished by this node requesting other nodes for a copy of their game state, and then reestablishing connection to every player. In this way, this player can continue playing, whether he's the coordinator or the watcher nodes in this particular turn.

Scaling & Performance

The intention of this game is to be played by a few players at this time, so we had some choices that might impact the overall scalability of this implementation. Assuming n players are playing:

First, every node directly connects to every other node, that is every node maintains n-1 connections. This might present challenges in complex network environments or very large networks (we might even run out of sockets!). Then, during every turn, 3*(n-1) messages are sent between the coordinator and other nodes. This puts a lot of network IO pressure on the node that is coordinating, since all of these messages either originate or are sent to the coordinator, with a large number of players this might present performance challenges.

However, with a low number of players (2-5), we've showcased that our current implementation will be sufficient. We've measured the average overall turn propose-commit time to be 117ms, and that is with some generous waiting builtin to the coordinator's side to save CPU cycles.

We've paid some attention to not blocking the main thread during network operations and transitioning the tasks into background threads if possible. This means users can still interact with the game using the CLI (and of course GUI, since that is handled with another separate thread) when some action is being done in the background, like waiting for the coordinator to make a move this round.

Lessons Learned

- Knowing the basic principles of messaging and consensus helped get the project started, but we should probably have gone for a re-architect and a re-scope to add more complexity and give this project more flourish in terms of DS features. We have a lot more ideas now, periodic state consistency checks, consensus and state resyncs on consistency failures, etc.
- The team didn't have great experience with async programming, so we chose the native socket implementation route. Despite our team members trying their best to keep operations from blocking, in hindsight using async probably would have made it easier.
- We didn't have the best idea of how to divide up the game logic between team members, and only came upon the mixin architecture afterwards. In hindsight we should have probably looked into this earlier, so the workloads aren't blocking between each other and perhaps have a even better distribution of workloads.

Contributions

Everyone contributed to system design, ppl took up different parts of implementation
Xinyang: Code structure, messaging and connection management, lobby logic
Axel: Infrastructure (VMs, ELK, deployment), GUI and CLI, testing
Runjie: Logic for starting the game, broadcast game configs, state initialization
Sergei: Coordinating game logic in a turn, two-phase commit for moves, win states