Project Final Report

By Adnan Jalal, Jackson Warhover, Haijun Si

May 1, 2024

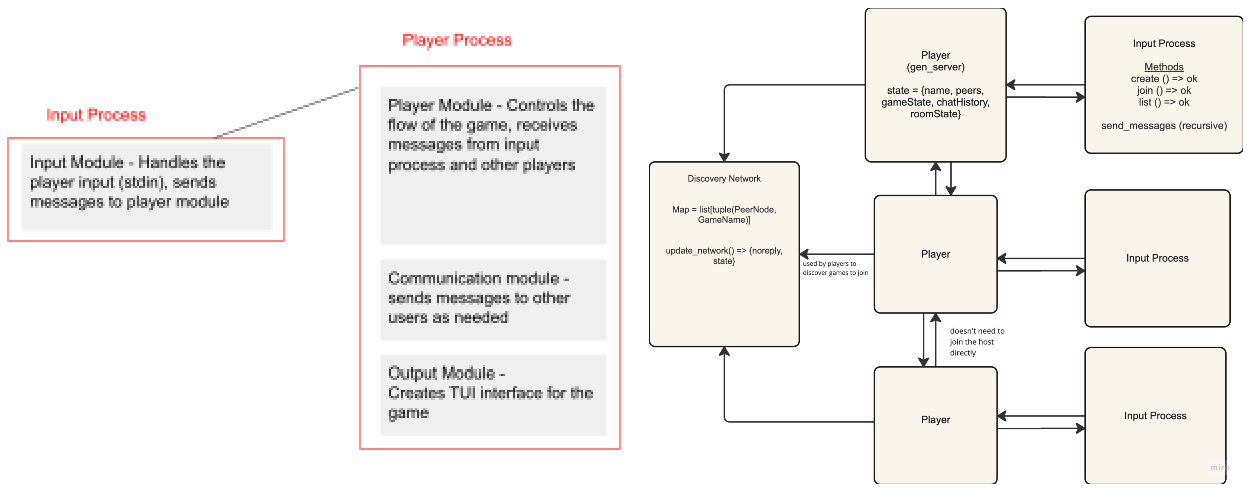
1. Our Story

Designing a multiplayer game is inherently a concurrency problem. We want to build a fun game and think about interesting concurrency problems along the way. Our first idea for addressing concurrency was to build the game in a similar way as the chat server was built. There would be some centralized server responsible for maintaining the game state, and each client (potentially composed of multiple processes) would act independently and communicate only with the server.

Since this is a very simple approach and would only require slight modifications to the chat server in order to produce something distinct, we decided to take on a more challenging task: implement the game using a peer-to-peer model instead of a client-server model. Implementing peer-to-peer would mean that each individual player of the game would be responsible for maintaining the game state independently, and instead of communicating with a centralized server, each player would communicate with all of the other players.

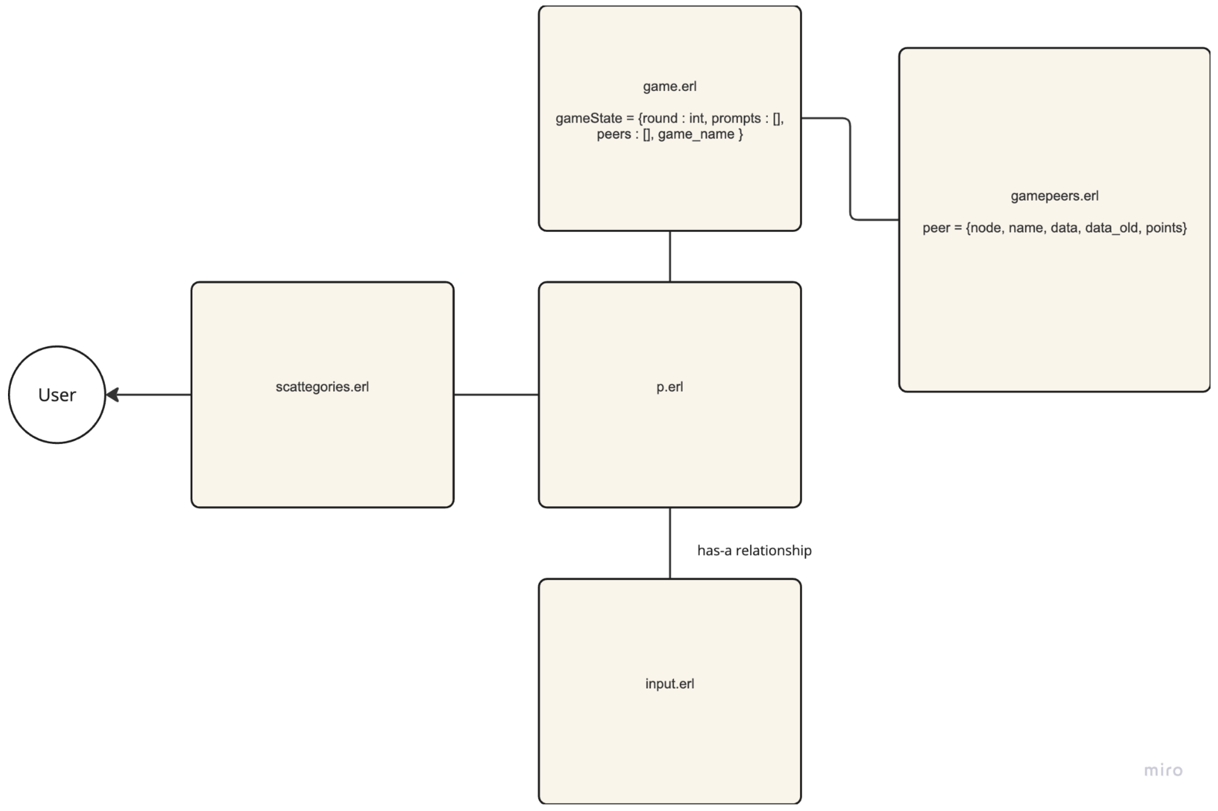
This will pose unique challenges regarding synchronization. There will be a high likelihood that scenarios will arise where each user's individual game state is not consistent with the one held by the other users. We will need to address this and design a protocol that works around this potential failure mode and other potential issues that come into play from not having a centralized server.

2. Overall Design


Input Process
Input Module - Handles the player input (stdin), sends messages to player module
Player Process
Player Module - Controls the flow of the game, receives messages from input process and other players
Communication module - sends messages to other users as needed
Output Module - Creates TUI interface for the game
Player
(gen_server)
state = {name, peers, gameState, chatHistory, roomState}
Input Process
Methods
create () => ok
join () => ok
list () => ok
send_messages (recursive)
Discovery Network
Map = list[tuple(PeerNode, GameName)]
update_network() => {noreply, state}
used by players to discover games to join
Player
Input Process
doesn't need to join the host directly
Player
Input Process

In our original design, we planned on having a discovery network where players could advertise their games and other players could join. However, due to time constraints, we didn't implement this system and our final program requires players to know the nodes of the peers that they would like to join.

Final Design


game.erl
gameState = {round : int, prompts : [], peers : [], game_name }
gamepeers.erl
peer = {node, name, data, data_old, points}
User
scattegories.erl
p.erl
has-a relationship
input.erl

3. Code Overview

3.1 runme.sh

runme.sh contains the general compilation for our erlang files. It includes obtaining a user's node/username, and running the program for scattegories.

3.2 scattegories.erl

This file contains the erlang interpreter for creating and joining rooms. This subsequently launches p.erl to obtain the information of the game being created. We created this file so that a user can seamlessly join and create multiple games, and it serves as a wrapper for the p.erl module.

3.3 p.erl

This module serves as a gen server that each individual peer runs on. It supports creating and joining rooms, as well as casts and calls. Joining and creating rooms creates a listener process, which is discussed more in the input.erl module. The casts and calls are utilized within creating and joining rooms, along with the game module to update the individual peers state.

3.4 input.erl

This file runs on a linked process responsible for parsing the user input. Generally, it sends the message to the gen server through a cast. Typing "–leave" stops the loop and invokes a client leave cast to the gen server.

3.5 game.erl

This file handles the mechanics of the game logic. The gameState record keeps track of the overall gamestate for an individual peer, which includes the rounds, the prompts for the game, a peer record list, and the game name. This includes functionalities like adding players, adding a submission, adding a vote, and displaying the leaderboard. This module is interesting due to our design of the peer-peer system, as we have functions for updating our own peer state and other peers in the game. Everything that involves calling the gen server uses casts, except for add_players, which is a call.

Originally, everything was a call, so that we could guarantee that peers receive and process peer-to-peer messages, but that turned out to cause a deadlock. If two peers sent messages to each other at exactly the same time, they would become blocked on receiving a reply from each other, essentially one of the simplest forms of deadlock. Join is safe because join only occurs when joining a game, so two peers could never join on each other at the same time.

3.6 gamepeers.erl

This file is a module that contains helper functions for handling peer records. We decided to abstract this from the game such that this module purely contains helper functions for handling lists of peers and peer records.

3.7 generate_prompts.erl

This file is a module which handles choosing random words and letters for our game. The file contains two main functions. One which chooses categories based on a given file and the number of categories. The other function simply randomly chooses different letters within the alphabet given a number of letters to choose.

4. Analysis of our outcome

4.1 Design Reflection

Overall, We think our design worked well. Our organization of how messages are sent to the gen_server was concise and led to some easy abstractions. For example, every peer-to-peer message contains the atom 'peer_input', which allows p.erl to only contain minimal boilerplate code for transferring work onto game.erl for state management. Likewise, the same happens for client_input, which handles typed input. This design allowed us to maintain a concise p.erl design free of many specific actions. This theme continued through to our other files. The designs of each file are concise, and there is very little overlap for responsibilities. Knowing the purpose of each file, it is easy to determine where certain functions are or where new functions should be placed.

We achieved our minimum deliverable. We have made a playable game with a TUI that reflects the current state. There are a few problems with the TUI, but fixing them would require major re-engineering. The project was definitely more difficult than we thought it would be. It took a lot of refactoring to make our code and modules concise but in the end we are really happy we went with Erlang. A big stretch goal for us was to create a GUI, but due to the complexity of learning NCurses or building a GUI in python, we were unable to complete this deliverable.

4.2 Division of Labor

We met many times over the course of the project period. Each time we met, we usually worked through parts of the code that were blocking our individual work, such that we could continue to work at home on certain features.

Haijun: The parts of the project I was responsible for independently were creating our bash script and scattegories.erl. Jackson and I worked pair-programming style on building most of p.erl, input.erl, game.erl, and gamepeer.erl

Jackson: I made a lot of the initial draft outlines of the gen_server and its connectivity to other files, worked with Haijun a lot on the main game logic, and worked on concurrency bug analysis

Adnan: I was responsible for creating generate_prompts.erl and generating the different categories which we chose from. I also worked with Haijun and Jackson when we would meet up over the project period.

5. Bug Report

5.1 Sending Messages:

Based on our design of the program, everytime a peer updated other peers in the game the other peers would print out the state of the newly updated game in the TUI. This caused an issue with the user input during parts of the game, where a user1 could be typing an answer or voting and user2 would submit an answer, causing the user1's terminal to print a newly updated state and erasing their answer. To solve this, we added a condition to check that a user has already submitted before printing out the updated game condition.

While there have been many concurrency problems that we have addressed and built solutions for, there are still some significant concurrency bugs in our code that we have been unable to completely solve at the time of this report.

5.2 Improperly formatted inputs:

Many of our functions that ask for user inputs, particularly at the beginning of the program when setting up or joining lobbies, fail to account for unexpected inputs. For example, entering a non-numerical value as the number of rounds when creating a game crashes the game, and inputting a node that is not running when attempting to join a lobby also causes a crash. However, these problems have been solved for joining a lobby that has already started, typing something other than ready in the lobby, and typing unexpected usernames while playing the game.

5.3 Race Condition:

Assume 3 players, A, B, and C, all in the lobby, where A and B are currently ready. If C becomes ready, C needs to inform A and B that it is ready. Assume B hears this ready message first, and then submits an answer before A hears from C that C is ready. Theoretically, this means that A could receive the answer from B before ever hearing that C is ready. Therefore A can receive an input for round X while it still believes that it is in round X-1. This will cause out-of-date information on A's behalf and will desync the lobby state, possibly causing A or other peers to be unable to advance to the next round. This bug is less likely to occur than the joining race condition (which we solved) since this one relies on fast successive inputs (which are unlikely in our game) instead of coincidentally synchronized inputs (which are always possible) which is why we devoted our time to solving the other problem.

6. Running Scattegories

To play our game, first clone the repository:

```
git clone https://github.com/haijun12/ErlangTheMovie.git
```

```
cd ErlangTheMovie/scattegories
```

Run the bash script using the following command:

```
./runme.sh
```

Enjoy!