Project Final Design

Team Name: the-a-team-but-not-in-the-ed-sheeran-way
Group: Chris Hinstorff
Derek Benson

For our final project we built a distributed messaging platform on Erlang. As part of it we also built a ‘demo’ application that takes advantage of the message servers to allow clients to chat with each other and challenge each other to games of count-to-fifteen/tic-tac-toe.

The main challenge of this project was building the concurrent messaging platform and making sure that it was scalable. We considered two possible choices for the design of this component:

- This first was a multithreaded python server that communicated with clients over a socket connection. We decided against this model because we would have ended up implementing a number of things that erlangs distributed communicated model gives us for free. That lead us to the second option.

- Erlang gives us a great communication model for message based concurrency between clients and different servers. The clients can easily make network calls to registered servers and wait for responses without impacting other clients. Inactive servers are also very lightweight as they are just Erlang processes. Erlang pattern matching also provide a great way for us to easily distinguish between different types of messages.

We were not able to hit our stretch goals involving friends and leaderboards, however we were able to build out the functioning game UI as well as a robust backend that scales across multiple erlang nodes. Our main inability to scale to these features was ironically the way that we designed our message platform, it’s great at passing messages between a whole slew of clients that all want to see every message. It fails where you would want to restrict the receivers of a message. To accommodate for this change in scope we decided to demonstrate our ability to scale the message system horizontally across multiple erlang nodes by demoing our system on several AWS instances.

The best decision that we made was choosing erlang for communication. The ability to spin up servers across multiple nodes would have been incredibly difficult without having Erlang's Gen Servers. Erlport also proved to be an effective tool for communicating with a Python client from Erlang. If we had the opportunity to do this project over again, we would have decided to experiment with Elixir and Python3. We ran into several annoying issues due to the way that datatypes are converted between Python2 and Erlang using Erlport, especially with strings. We also found it difficult to find what we needed to in the Erlang documentation, Alex's presentation about Elixir made us wish we had learned about Elixir before starting work on the project.

The most annoying bug that we found was related to Erlport and communicating back to Erlang from Python. We solved the communication from Erlang to Python by just making the game state global and creating several 'setter' functions in Python for ErlPort to call. We then created a function for Erlport to call that starts the main game loop inside it's own thread so that the Erlport shell remains responsive. When we started trying to call Erlang functions from inside the game loop it the Erlport shell kept crashing because it was receiving an unexpected message. It took us a while to realize that the Erlport shell was receiving an unexpected message because the call from the game loop thread was being intercepted by the Erlport shell and not the thread that made the call. We solved this issue by giving each game client a dedicated Erlang process that just listens for a cast of {Module, Function, Args}, basically allowing the threaded game loop to 'cast' to a function call without receiving the response.

game.py - handles launching the python GUI as well as maintaining the gamestate and facilitating the communication between Erlang and Python

text_input.py - handles inputting of text into pygame screen. This is only a slight modification of a permissively licensed open source module as attributed in its file header comment.

The src directory contains all of the Erlang modules

userserver - this module defines the main server that clients connect and talk to upon login. It does authentication (ensuring no duplicate usernames) and name resolution for chat rooms.

clientserver - this module defines the server that every client has to run to connect to the network. It authenticates with the userserver and stores the rooms that a user has

connected to. It also provides a way for the user to broadcast messages or listen to messages from specific rooms.

nodemanager - this module defines the server that every node capable of spawning msgservers must run. When it boots it registers itself with a userserver and then starts listening for requests to spawn nw msgservers.

msgserver - this module defines the server that broadcasts messages out to clients. It allows for subscribing, unsubscribing, messaging, and listing the members of a room.

tictactoegame - this module defines a specific set of methods that are used to plug the python gui and tictactoe game into the Erlang communication system. This file is the example file for how someone else could use our libraries to dynamically create rooms, pass state between clients by sending messages and listening for new messages.