Final Report – Banana-Tag:

Creators: Theodore Schneider, Matt Zhou, Noah Tervalon

a) **Description**: A game of tag where chaos reigns.

b) **Rules**: Normal tag. However, when you are tagged you are temporarily suspended from the game until the person that tagged you gets tagged. Thus, the only way to win is to tag everyone without being tagged once. Visit this link for more rulebook information, note that the video is not related to the rules on the page: https://ultimatecampresource.com/camp-games/tag-games/banana-tag/

c) **Our game**: We plan to make an interactive game that allows many people to play at once. Our area of play will be a finite map with multiple player characters all striving to become the champion.

Min Deliverable:

Our minimum goal is to create a visual game that is hosted on the Halligan servers. This game will be multiplayer (minimum 4). Additionally, we want to have graphics for the game that allows all players to see what is currently happening in real-time. This includes player movement, players tagging each other, and other events happening on the map. For the game to work, we also plan to have the ability to accept input accurately from all players and avoid concurrency issues such as deadlocks, etc. On the whole, we aim to provide a smooth-gaming experience for the users through our input processing.

Max Deliverable:

Our maximum deliverable is a fully polished game that supports large-scale player bases (15+). We aim to have multiple maps that each have unique layouts (obstacles, dark mode, etc).

Abstraction and Language

Abstraction Discussion:

1. Our different modules can be seen in the class diagram below.
2. Our server python file consists of multiple classes. One is a gamestate class, and one is the player class. These classes allow us to group useful functions, and abstract information that others don't need to know. The gamestate class will call the relevant functions from the player class to reflect the player's status, position, and status. The gamestate class will keep track of the progression of the whole game and handle any interactions between the players as well.
3. The Erlang modules handle all the communication between the server and players and are pretty straightforward. There will be Erlang nodes situated on the client side (one for sending player input, and one for receiving info from the server) and one Erlang node on the server side that communicate with each other and pass the relevant information onto the Python modules. The most important decision was figuring out the format we wanted to pass the information about the current game state to the clients.
4. The Python modules and the Erlang modules will talk with each through erlang ports to pass information.

Language Discussion:

There are multiple things we refer to when talking about our program and its functionality:

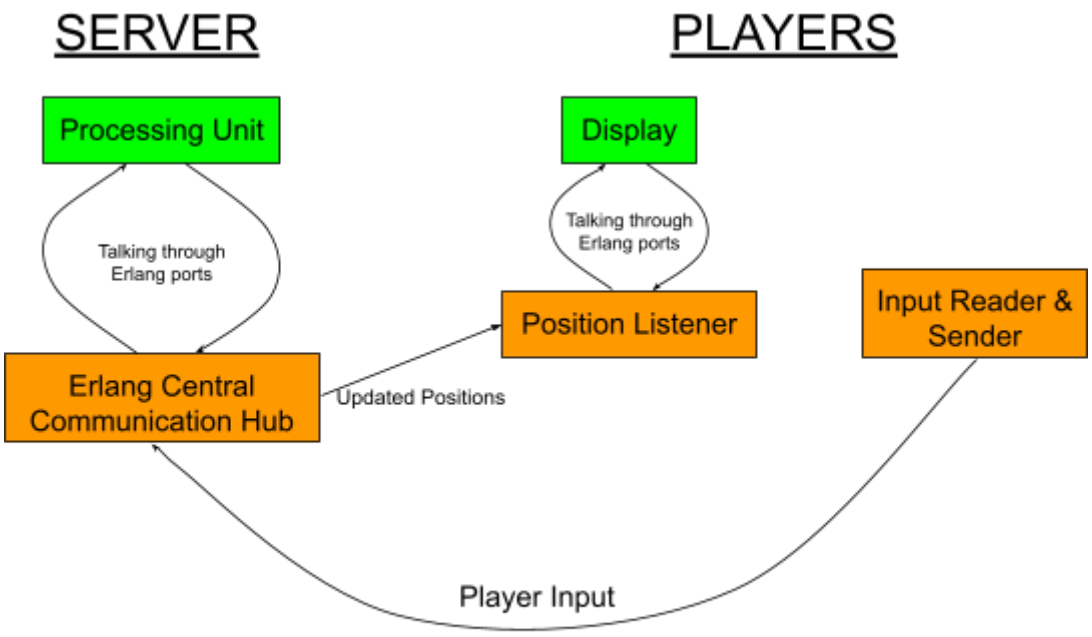
1. Player input: This is just the key presses of the real-life player. This is reflected in the design of the Erlang module that collects this information.
2. A Tag: This is another name for a collision of two players on the screen. In our proposal, this was not fully discussed. Below this section will be a more in-depth discussion of this process and which modules will handle it.
3. Updating player info: This is our language for the process in which the server receives player information and updates the game state by sending it out to all the display modules.

A Tag:

A tag will be accounted for using multiple modules.

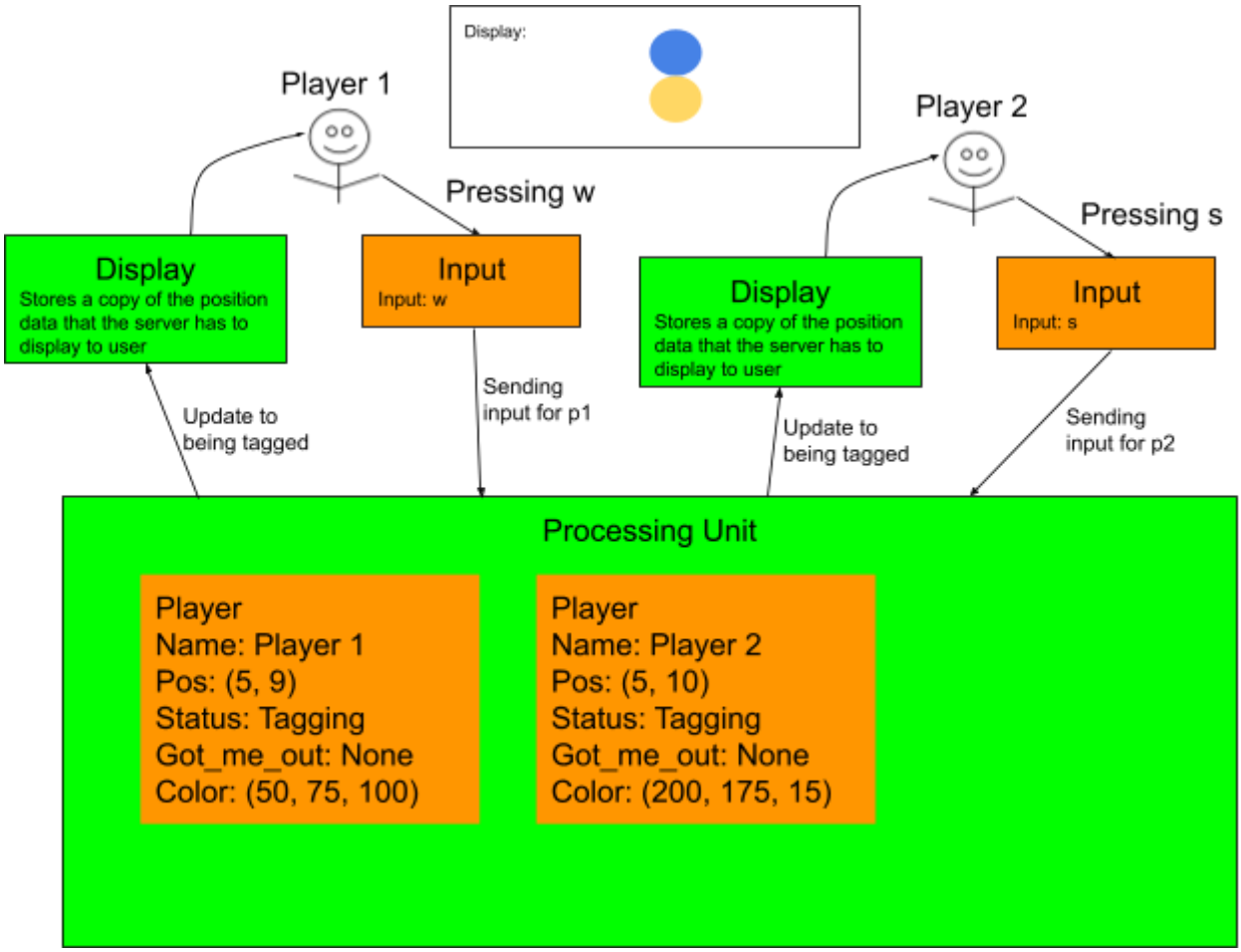
1. First, all the player input is sent to the server's Erlang module which then relays it to the Python server module.
2. Each time player locations are updated, we want to check if any two players are in a location.
    a. NOTE: we know this could be very slow. However, we just want to get something working first. This is subject to change in the final design.

Class Diagram:



Note that green boxes are python and orange boxes are erlang

Object Diagram:

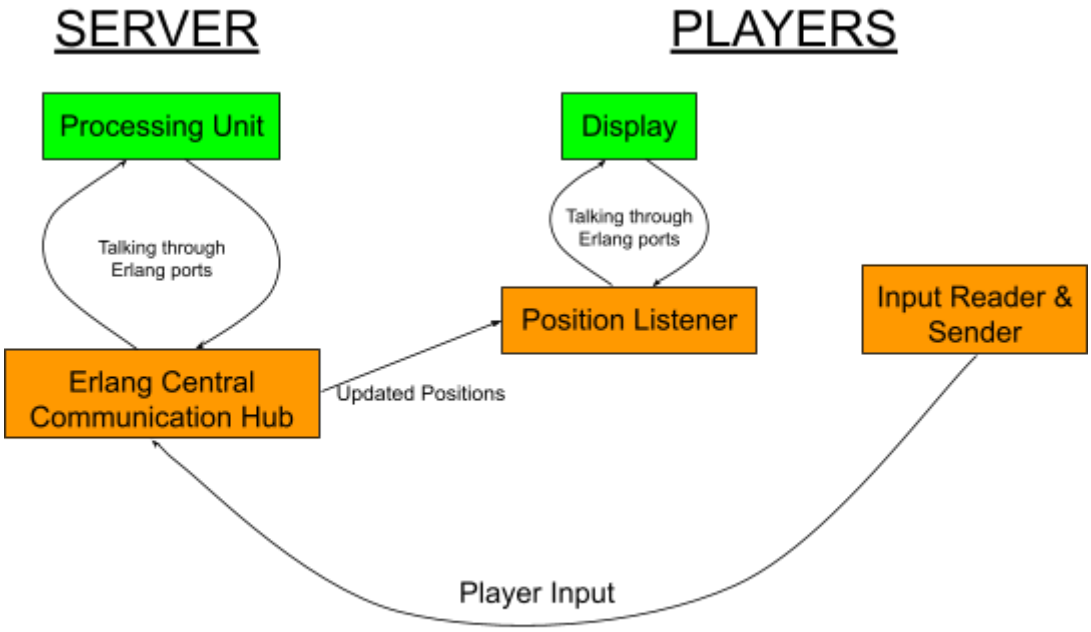


The display at the top shows the game state before the players input their movements.
Player 1 started at position (5, 9).
Player 2 started at position (5, 10).
Player 1 sees a displayed game board and presses "W", this information is relayed to the server.
Player 2 sees a displayed game board and presses "S", this information is relayed to the server.
The server receives the messages, calculates the positions of the players, and recognizes that Player 1 and Player 2 can tag each other.
The server prompts the displays of the two players to show them that they can tag each other.

DEVELOPMENT PLAN:

1) Week 0 (March 11):
    a) Familiarize ourselves with useful packages and come up with a more concrete design
    b) This includes writing small programs to test the usage of these packages
2) Week 1 (March 25):
    a) Finish object and class diagram for the initial design proposal
    b) Create a basic working model for message passing and work out the overall architecture. This includes creating a central node that's able to communicate with client nodes. Also, nail down the way to take inputs and display basic info on the client side.
3) Week 2 (April 01)
    a) Connect the different components. Mesh the client input and display code with the overall architecture. Finalize the communication protocol between the client and server.
    b) Work on the refined design
4) Week 3 (April 08)
    a) Implement the communication protocol and have basic functionality
    b) Finalize the tagging protocol
    c) Test the connected features and communication protocol and work on bugs.
5) Week 4 (April 15)
    a) Implement Additional features:
        i) Multiple games being able to be played at once.
        ii) Fog of war.
        iii) Multiple maps.
        iv) Objects in maps.
6) Week 5 (April 22)
    a) Test advanced features.
    b) Test features even more. And even more.
    c) Write final report
7) Due Date: Apr 29, 2024

Final Design:



Note that green boxes are python and orange boxes are erlang

Our final design has largely remained the same as before. There were no changes made to the class diagram. There were some changes made to the Python modules at both the players and the server side. Both of the Python modules will spawn a new thread for listening for input coming from Erlang. The input will then be placed on a queue and processed by a separate thread. The server no longer has an explicit gamestate class, the game state is simply reflected through global variables.

➢ Outcomes:

a. We fully achieved our minimum deliverable. At present, we have a fully functioning game that has the ability to support multiple players (4+). Input delay and lag are kept at a minimum, and each player's displays are the same.
b. As for our maximum deliverable, we fell slightly short. While our game has the ability to support a larger player base, we have not yet implemented additional features such as new maps, powerups, etc.
c. On the whole, we fell sort of in the middle between our min and max deliverables.

➢ Design Reflection:

a. The best decision that we made was to use Erlang to handle all the communication between the clients and the server. The message-passing abstraction saved us from looking at network protocols to set up communication, and this saved us a lot of time. Also, it was very easy to format the messages as there was no need to design our own serialization protocol to pack data, etc.
One thing that we would do differently next time/would like to try is to try and set up all the display/game logic in erlang. We were not too familiar with Erlang packages that supported GUIs and graphics and chose to use python. But we did run into a decent amount of bugs trying to ensure that the communication between Python and Erlang is correct. Next time, we might try to do the whole project in Erlang.

➢ Division of Labor:

a. Our division of labor was effective. We split into these broad categories: Matt - message passing and node architecture; Teddy - keyboard/player input, Noah - graphical display and central server/game state.
b. Doing this allowed us all to work independently on different areas of the project and then integrate our work during weekly meetings.

➢ Bug Report:

a. The worst bug we encountered was excessive latency on user-display. In other words, each player's screen was extremely delayed compared to their input, and their screens were not synced at times.
b. Originally, we thought the bug was caused by a slow message passing on our end. However, after logging all messages sent to all players and using the time library, we were able to deduce that all players were receiving the same messages within a few thousandths of a second. So we determined that the latency was not occurring due to slow message passing.
c. We deduced that it was actually the tool we were using for the displays (X-11 forwarding) that was causing this issue rather than our code itself. This took us multiple days of meetings plus help from Neil together to figure out. When we tried the game on our own network, the game ran smoothly.

d. In retrospect, if we had done more in-depth research about X-11 forwarding to determine if it was suitable for real-time updates, we could have saved some time by choosing not to use it. Also shoutout to Neil (one of the course TAs) who helped us find a workaround to this issue by having everyone clone the repo and run the code locally.

➢ Code Overview:

a. client_display.py: Spins up the pygame window that players will see the game running on. This Python process will be spawned by the listen.erl module when the listen module opens an Erlang port. The process will then take incoming messages from the port and update the client display.
b. listen.erl: Erlang node that listens for communication from the server and then forwards it to the Python display on the client side. It will spawn the Python process client_display.py through a port opening.
c. input.erl: The Erlang node that collects player input from the terminal and sends it to the server.
d. gameserver.erl: The Erlang node that sits underneath the Python server and handles message passing to and from clients. This file will spawn the server.py process by opening an Erlang port. It will pass incoming messages from clients to the Python server and also take messages from the Python server and send them to the clients.
e. server.py: Python server, handles all the processing in the game. Will communicate with gameserver.erl to pass messages to players and take updates from players as well.
f. joingame.sh: Script for clients to run to connect to the game. Will start the listen node and input node on the client side. This spawned listen node will in turn start client_display.py.
g. startserver.sh: Script to start the Erlang server node, which in turn, starts the server.py file

➢ How to Run the Program:

a. **NOTE: For the lowest input delay, clone the repo locally and run the game on your own network: Noah-Terve/banana-tag (github.com), so that displays can pop up without using X-11 forwarding. If on Halligan, try to use joingame.sh on the same vm where the server is located to minimize delay.**
b. First, you will need to install erpy and pygame on a python version that is able to run them (>= 3.11)
c. In order to run the game you will have to fire up a server node, you can do this by running ‘sh startserver.sh’
d. Once the server is running, on the same network any number of players can join by running ‘sh joingame.sh’

e. This join script will have the players enter the node name of the server and their name. It is important to have all players have unique names.
f. Once you have done both these steps the players will have 2 windows open, 1 is a pygame window which is purely for visual, and the other will be their terminal that they ran the script from, this will be where you put input. Bring the terminal into focus, and then each player just has to press r to start the game.
g. Once the game is started you will move around with WASD, and when you tag another player you will click r p, or z to choose rock paper or scissors, whoever wins will stay in, and the other person will be out until the person they lost to is back in.