Initial Design

- car knows how fast it's going
- car's path is a one-dimensional interval [0, 1]
    - position is number in that range
    - just a single road
    - car calculates path to take upon initialization
    - cars should be able to pass each other when on road
- road layout is specified in TOML file that erlang and python both read

- libraries (and similar things)
    - erlang's digraph module
    - https://www.erlang.org/doc/reference_manual/ports.html - for communicating between erlang and python
    - https://github.com/bdcht/grandalf/ - for calculating graph layout in visualization
    - pygame for visualizing the roads and cars
    - https://github.com/dozzie/toml - parser for road configuration

Notable changes:

We decided that, instead of using something like TCP or UDP to communicate between Python and Erlang, we would use Erlang's model of "port drivers", which are processes (in the operating systems sense) that can be spawned by an Erlang program, and are subsequently communicated with by using the usual message-passing paradigm. We plan to have the Python program send the Erlang program a message when it's ready for the traffic state to be updated.

Data Structures and Algorithms

- hash table of FIFO queues to model different lines of stopped cars in an intersection
- FIFO queue to model a cyclic list (for a primitive way of alternating traffic signals)
- built-in pathfinding algorithm in erlang's digraph module
- a graph layout algorithm from python's grandalf library

Development Plan

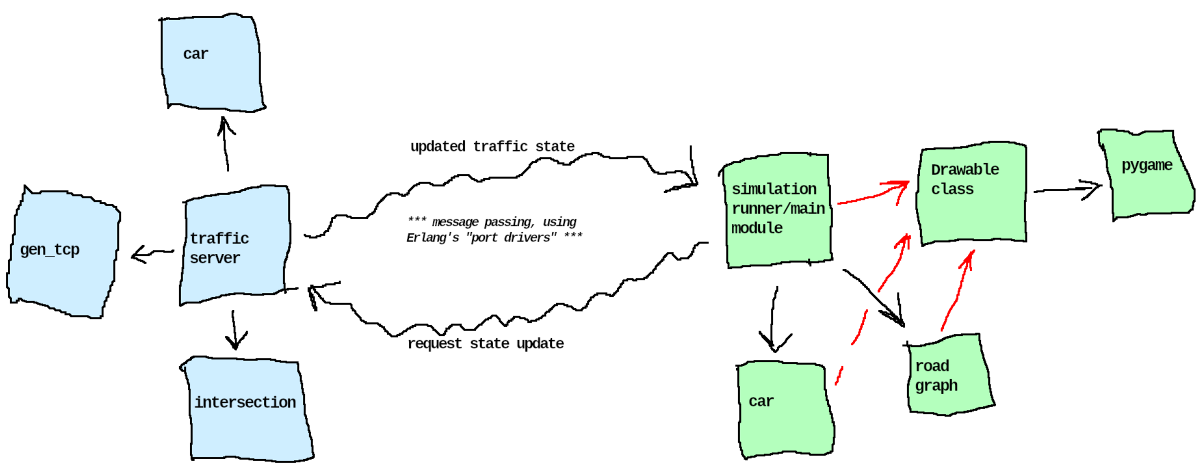
- ~~Week 0 [28 Mar]: figure out overall design; run simple scenario with 1 car, 1 intersection, and 1 road, without Python~~
- ~~Week 1 [04 Apr]: visualize simple scenario using pygame~~
- Implemented so far:
    - simple port-based communication between python and erlang
    - traffic simulation (just the erlang parts)
- Still needed:
    - loading road graph from a config file
    - python visualization

- Week 2 [11 Apr]: read graph in a TOML file; communicate car states between python and erlang
- Week 3 [18 Apr]: visualize everything with pygame
- Week 4 [25 Apr]: presentation day; everything should be finished

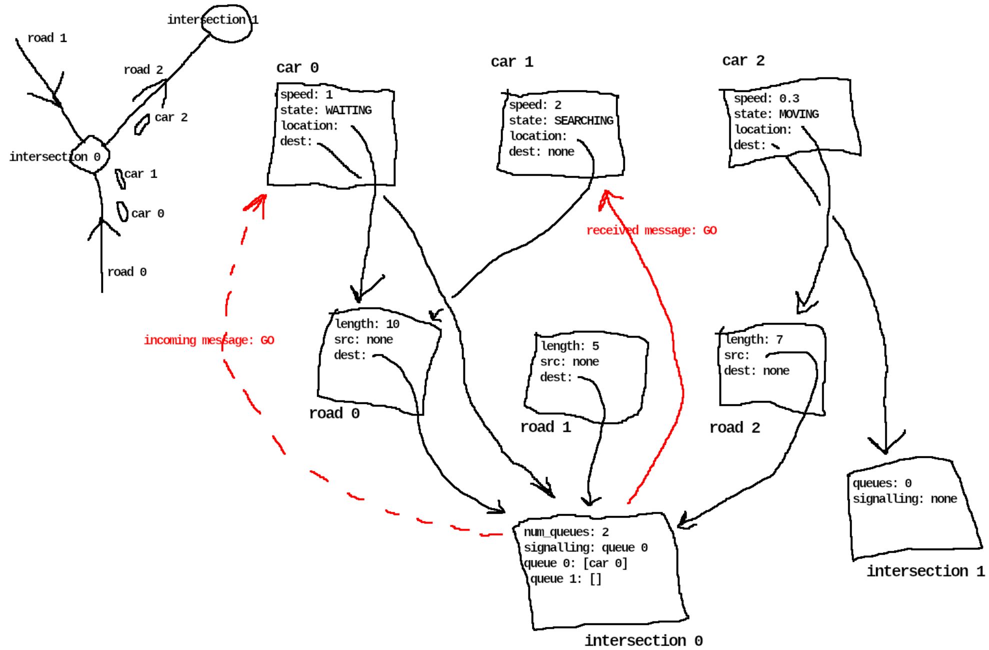
Roles:

- Ben: Erlang parts
- Liam: TCP communication between Erlang and Python, communication protocols
- Andrew: visualization

<u>Diagrams</u>



^ class diagram; blue modules are in Erlang, green ones are in Python; black arrows just represent module inclusion, while red arrows represent a subclass relationship



^ Cars and intersections are represented as processes; roads are just tuples or records. References to cars and roads are PIDs. When a car arrives at an intersection, it sends the intersection its PID. The intersection enqueues it, and will eventually let it through by dequeueing its PID and sending a 'GO' message. An intersection has a queue for each incoming road, but this could be simplified by consolidating all incoming traffic into a single queue, though that would be less realistic. After a car receives a 'GO' message, it uses a pathfinding algorithm to determine which node to visit next, in order to reach its destination in the fastest time.

Here's how the illustrated scenario came about: car 0 and car 1 started on road 0, and car 2 started on road 1. The following messages were then exchanged, before the cars reached their current configuration:

- `car_2 arrives at intersection_0: {arrived, self()} ! intersection_0`
- `intersection_0 lets car_2 through: go ! car_2`
- `car_1 arrives at intersection_0: {arrived, self()} ! intersection_0`
- `car_0 arrives at intersection_0: {arrived, self()} ! intersection_0`
- `intersection_0 lets car_1 through: go ! car_1`