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
    - finds next road at intersection using shortest path to dest
- ~~max speed of a road is min(speed of fastest car on road, speed limit)~~
    - cars should be able to pass each other when on road

- tentative ideas
    - each car (and pedestrians? maybe add those later) is an actor
    - each intersection is also an actor
        - has multiple 'queues' (list of car pids)
        - to let car through, take head of queue and send 'go' signal
    - car movement just steps increments position

- libraries
    - erlang's digraph module; python’s pyerlang module

<u>A short story about a design decision:</u>

We decided to represent a network of roads and intersections as a directed graph (roads are the edges, intersections are the nodes). Initially, we thought that a car's physical position on a road would be too difficult to simulate, so we decided it would be better if cars themselves just "jumped" between intersections after certain periods of time (which would correspond to speed and road length). Using this model, we also decided to make pedestrian traffic block an entire road, rather than a section of the road--again, in order to simplify things.

Then, we thought about what part of the simulation would need to be concurrent, and figured out that it would be easiest for the cars to be concurrent, which would (practically speaking) require them to track their actual position on the road. We figured out that position tracking wouldn't be so difficult if we chose to model position as a value between 0 and 1, representing how far a car is along a given road. This seems like a good compromise between simplicity and reality.

<u>Data Structures and Algorithms</u>

The only significant data structure will probably be a FIFO queue and a directed graph; Erlang has modules for both. Because each car wants to reach its destination as fast as possible, it would use a pathfinding algorithm to determine which road to take next, updating its path at each intersection. We would probably use Dijkstra's algorithm for this.

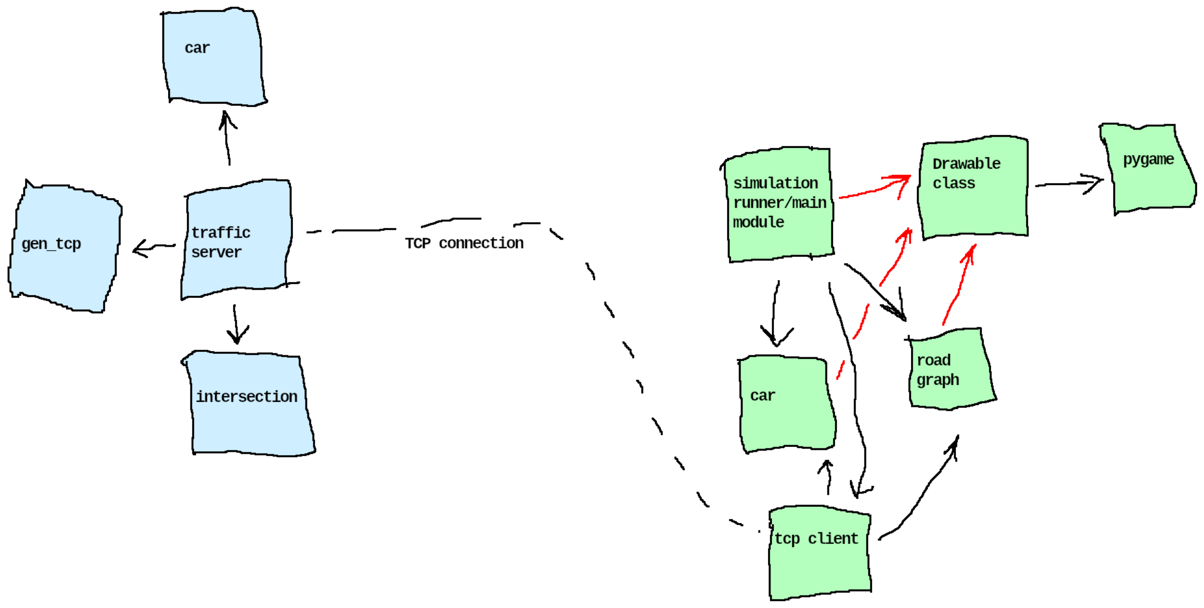
Development Plan

- 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
- Week 2 [11 Apr]: scale scenario to multiple cars, intersections, roads
- Week 3 [18 Apr]: extra time for testing and debugging
- Week 4 [25 Apr]: presentation day; everything should be finished

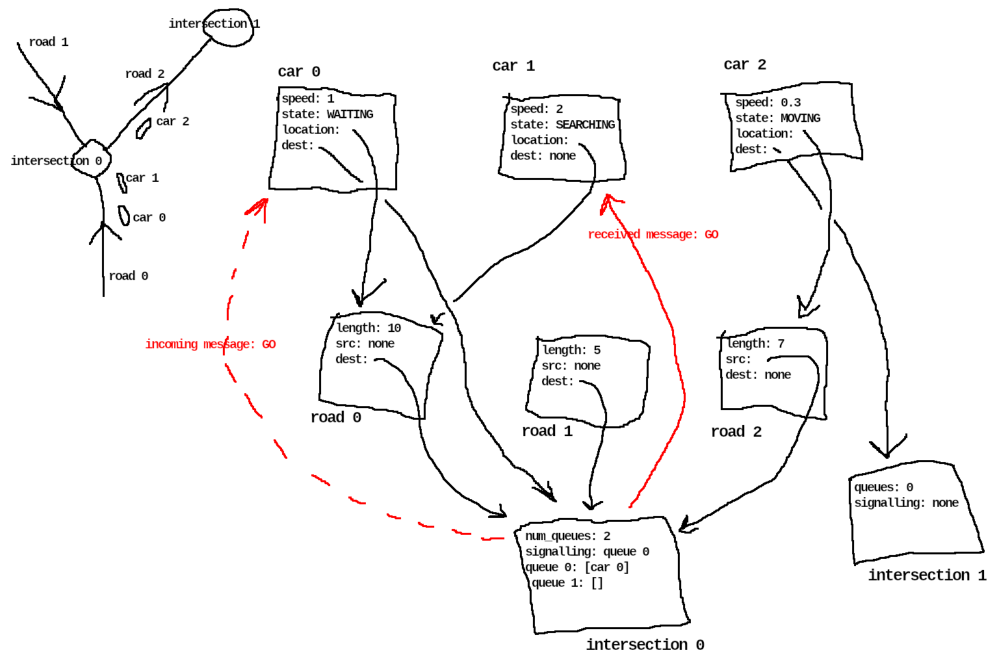
Roles:

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

Diagrams



^ 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`

Code demonstrating our understanding of the libraries we intend to use can be found in our github repo, subdirectories `pyerlang_demo` and `digraph_demo`.