SiMBTA

A simulation of the subway lines of the MBTA.

By: Hannah Clark, Raewyn Duvall, Andrew Stephens

Project Description

The project is to run a simulation of the subway lines of the MBTA to allow observation of patterns during busy hours and during service disruptions.

The MBTA often experiences events that affect people's travel times. As many people rely on the service to get them where they need to go in a timely manner, it is the responsibility of the system's managers to ensure that service disruptions do not inconvenience their riders an excessive amount. At the same time, they also have to work with a limited budget, and so have to figure out a way to deal with these disruptions in a cost-effective manner, whether it be running more trains or changing schedules to allocate trains to busier times. The project is intended to be demonstrate events such as delays, perhaps due to breakdowns, or the effects of increased traffic on various schedules, such as during commutes or after sports events.

We will use the MBTA's data for average time between stations, which we will use to calibrate our simulation. There will be one clock governing the travel times in order to produce reasonably accurate time estimates. Passengers will be created as specified by the input and will record the duration of their journey for use in calculating averages.

Concurrency will be used in this project by making each traveler, train, and station a separate process. The purpose of this is to make the simulation as true to life as possible, where various travelers in the system also may move concurrently, except when being physically blocked, which our simulation will take into account. Each train will also be its own thread, and will allow certain passenger threads to be linked to it via message passing.

Minimum and Maximum Deliverable

The absolute minimum planned deliverable is a simulation of the Red Line, the results of which will be output to a text file. The maximum planned deliverable is a visualized simulation of all lines for which the MBTA provides accurate information.

Because there is a large range of deliverables between those two points, we have decided on a progression that we feel will give us the best chance at producing useful results. After implementing the Red Line simulation, we plan to implement a the most basic visualization possible that still presents the information in a useful manner. We will then add a simulation of the Orange Line, including transfers between the lines. We will then add the remaining lines for which we have data, then improve the visualization.

Design Overview

We will be programming the simulation in Erlang. Initially, we were tied between Erlang and Python. Some parts of our design, such as the clock and how to keep the order of trains on a track lended themselves to concurrent programming design patterns for multi-threaded programs that we had previously encountered. For the clock, because our timing is event driven rather than system clock based, a condition variable would have been very useful for sending out ticks. For maintaining the train order, we could have had a module with many FIFO queues. However, Python would greatly limit the scale of the simulation because of the limits on the number of threads. Erlang was then considered because of the large number of processes. Also, passing messages is closer to how the components of our simulation behave in the real world. After laying out possible ways to create each component in Erlang, we determined that it would be feasible to do, and so we decided on Erlang so that we could have the most true to life an large scaled simulation.

We will have modules for trains, passengers, stations, the clock, the cartograph, and the main module for program startup. The cartograph is a set of functions that deliver information about stations, including which are reachable from a given direction and travel time between them. A train keeps track of the passengers currently on it; how long before it will arrive at the next station, if not delayed by a previous train or currently in a station; how long before it will leave the current station, when in a station; its capacity; and its direction. It also keeps track of the number of passengers that still wish to disembark while in a station and the number that may still do so in the current minute, the values of which are only relevant while the train is in the station. A passenger keeps track of its starting point current location, its endpoint, and its desired train to board. A station keeps track of the current trains, if any, on its inbound and outbound platforms; the trains approaching or waiting for a platform for both inbound and outbound directions; and the passengers in it waiting for trains in each direction. An “instance” of each of these modules is a process begun by the module’s start function that runs the module’s loop function.

The clock module was also the focus of a major design decision. The initial thought was that the clock simply send a tick to all trains and passengers that have yet to begin their journey after a predetermined amount of time chosen to represent a minute within the simulation. That would have been very easy to implement. We realized however that there was another possible method, and event driven clock. Because of the possible scale of the simulation, a great deal of things need to be able to happen in a minute to be accurate. While events can happen very quickly, there still could be the possibility of having to make the simulation minute longer and slow the simulation down. Also, we realized that if no passengers start a journey and there is no train in the station in a given minute, then waiting seems pointless because nothing needs to happen. Therefore, we thought of using an event driven clock. A simulation minute would then last exactly as long as needed, which is long enough for passengers to enter the station and for a reasonable number of passengers to board and disembark trains that are in the station, if there are passengers needing to do so. This would be able to handle cases where a large amount of things need to happen in a simulation minute without slowing down simulation minutes where few things need to happen. This method, while a little less true to life at first glance, should be just as true to life because of the limits on boarding and disembarking, which will be based on what is feasible in the real world, if not more so because of the prevention of unrealistic limiting due to a simulation minute not being long enough to execute all needed behaviors. While the event based clock requires more complexity, it was the method chosen because of the possible speed advantages. The clock will keep track of all trains, the number of trains done with boarding and disembarkation for the current minute, the passengers who haven't entered stations yet, and the current time. At the beginning of each clock tick, the stations and trains will send their state to the output module. Passengers will send the output module their start and end stations and the time taken for the journey upon its completion.

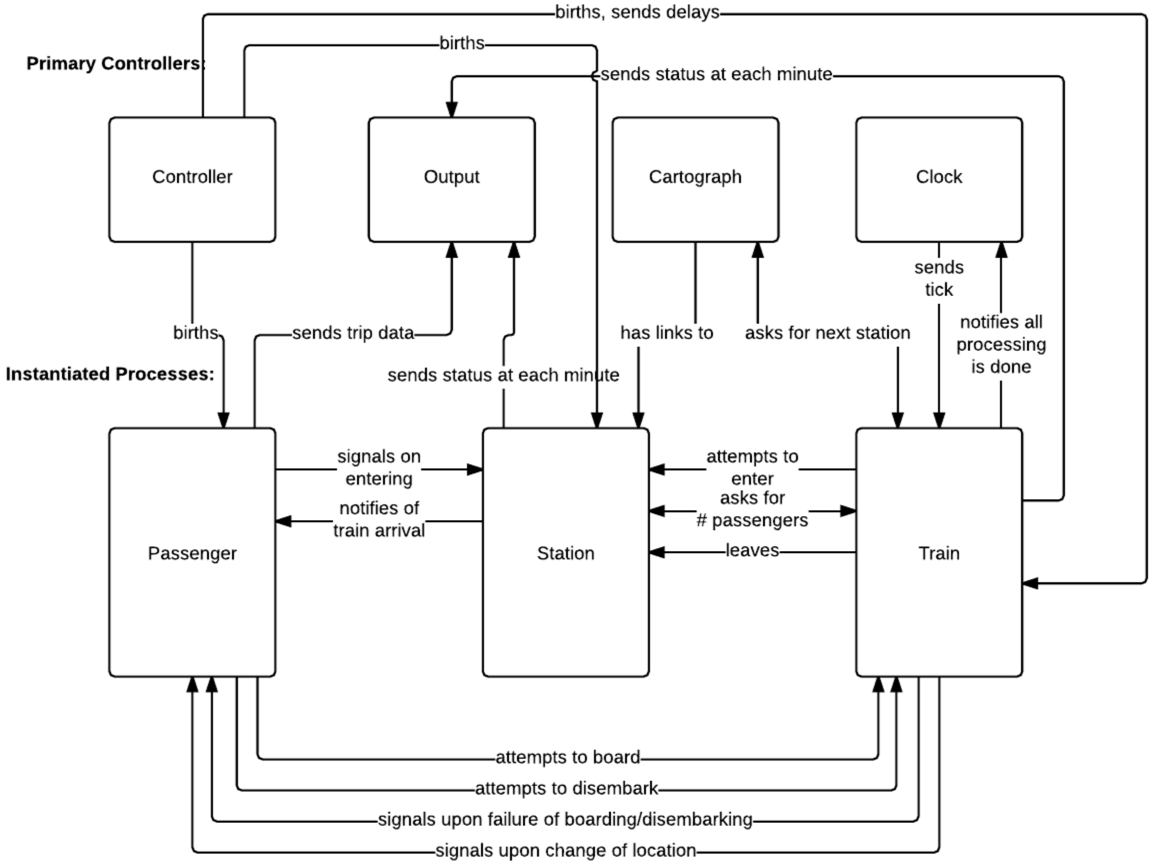
The main module will be used to start the simulation and clean up remaining processes upon completion. It will read in the simulation specifications from a specified text file, start up the clock, and create instances of the other modules as needed for the simulation, then kills remaining processes upon completion. It also communicates information about these modules to the clock and output modules. Main also contains a loop that causes train delays as indicated by the input file. Delays are specified in the input file to happen at given times for the corresponding train.

The clock also serves as a rendezvous point for the entire program. The next minute is not triggered until all operations in the previous minute are complete. Before this happens, we take this opportunity to record all data about the current state of the program using an output module. This module exports information about all of the trains, stations, and passengers that completed in the minute to a text file, which aggregates this data over the course of the simulation. At the end of the simulation this data is loaded into a web page for a simple visualization. We choose to output data at the end of every minute, rather than as it happens, in order to make sure that messages do not get output in an incorrect order, for example, data from train 5 at minute 3 being output after data from train 2 at minute 4. The output module will receive messages from trains and stations each minute and from passengers at the end of their journeys and handle all output to prevent the slowing of the simulation. It's sole purpose is to provide consistent and organized output. It only keeps track of the number of stations and trains in the simulation and the backlog of its output.

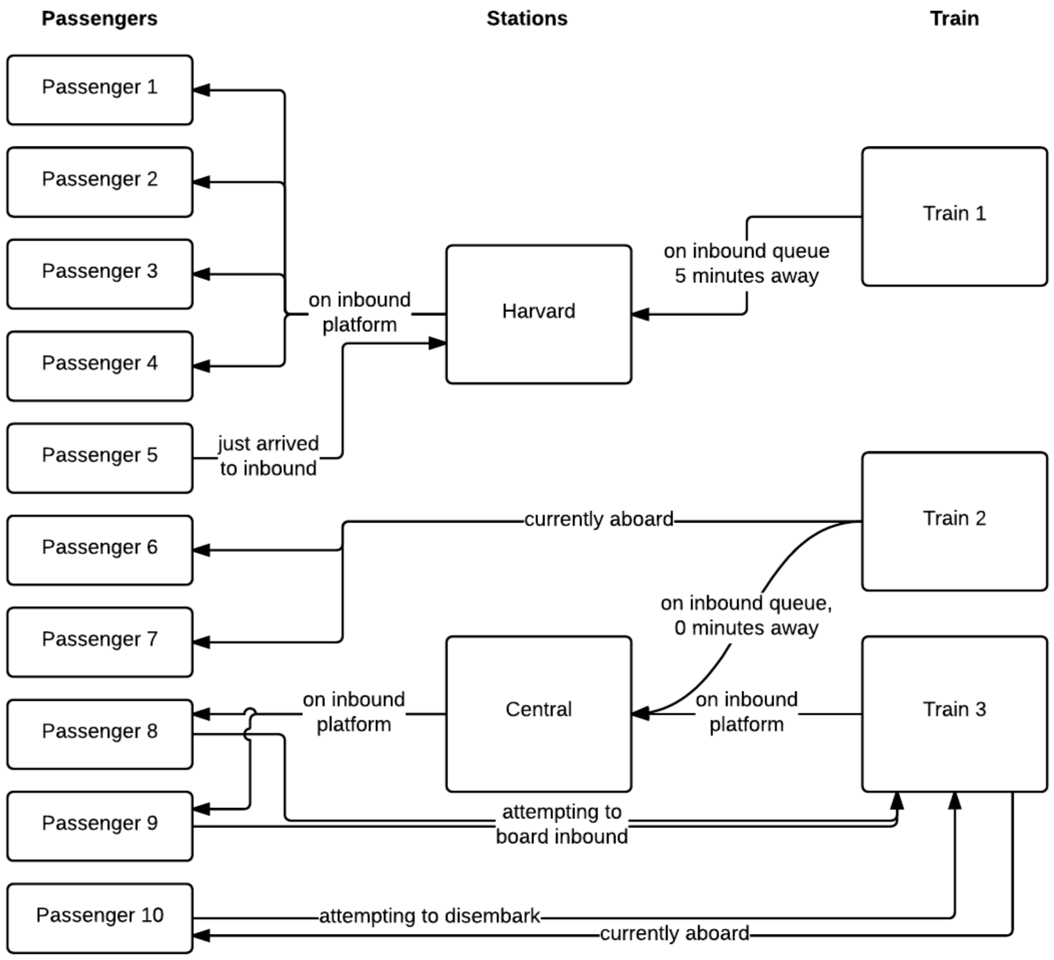
The visualization will be built in HTML using jQuery. The file will be parsed when the web page loads, and the page will contain a view of the red line and a slider for the user to change the current minute being viewed. Each train will be displayed as a graphic with a number, indicating the number of passengers on board, located either at a station or between two stations. Each station will also be displayed with the number of passengers currently waiting. Each passenger that finishes his/her journey at a particular minute will be displayed as such, along with the time it took him or her to travel from their origin station.

This design covers all of the components of the original minimum deliverable, with a simple visualization for the data following the simulation. The simulation only uses the Ashmont branch of the red line, so the path is linear.

Module Diagram:



Object Diagram:



At this particular moment in time, we examine the relationship between 10 passengers, 2 stations, and 3 trains. Of course, in the actual simulation there would be many more of all of these, but for the sake of simplifying the diagram only a few are shown. Passengers 1-4 were all birthed into existence by the master controller at various points in the last few minutes. Each one has a definite destination in mind, and by querying the cartograph they were each able to determine that inbound was the correct direction to go to get to their destination. The station is aware of their existence, and has them in a list that it plans to signal once an inbound train arrives. Passenger 5 has just arrived to the station, and so it notifies the station accordingly. It will then be added to the list of people to be notified when an inbound train arrives. Meanwhile, Train 1 has left Porter and is in the queue for Harvard, but still needs to wait for 5 ticks before it can begin boarding (as it will not have arrived at the station before then).

While these passengers are busy arriving and waiting, Passenger 6 and 7 are traveled to Park St for a romantic dinner in the North End. They are currently passengers on Train 2, which means that no station has reference to their PIds, but Train 2 does. Train 2 is currently on the track between Harvard and Central, but cannot enter the station because Train 3 is currently on the platform, and therefore Train 2 is waiting on a queue. As soon as Train 3 leaves, Train 2 will enter and notify its passengers that it has arrived in Harvard. As Passenger 6 and 7 are going to Park St, they will disregard this message.

At the same time, Train 3 is on the platform and boarding passengers. Central station has already sent a message that an inbound train is available to all the passengers on the inbound platform, allowing the passengers to request the ability to board the train. Depending on how crowded the train is and how many people are trying to disembark, the train may deny this request. For example, Passenger 10 is currently on Train 3 but wants to get off so he can get to his stand-up comedy show on time. As long as he is not blocked by the other passengers attempting to disembark, he will be successful. Otherwise he will have to wait a minute before trying again. Passengers 8 and 9 will only be able to board once all of the disembarking passengers have successfully done so.

There were no delays added to this simulation, so the controller has no delays to send to the train and is no longer in existence.

File Formats

Input

```
train directionAtom startTimeInt capacityInt numDelaysInt
delay timeInt delayLengthInt
passenger countInt startTimeInt startStationAtom endStationAtom
```

Output

```
Minute num
train Direction:dir Station:station Passengers:num
train Direction:dir Approaching:station Passengers:num
station Name:station Passengers:num AshTrain:bool AleTrain:bool
passenger Start:station End:station Began:timeNum Duration:num
```

Delays apply to the train immediately preceding. A `numDelaysInt` of zero for a train indicates there are no delays following. The number of delays after a train must match `numDelaysInt` exactly. Any misspellings will cause the simulation to fail either because it cannot parse the input file or because some passengers will not be able to exit trains. Trains must begin at minute 2 or later.

Outcome Analysis

Our minimum deliverable was met with the Alewife-Ashmont Red Line Train being simulated accurately based the rules we pre-defined in terms of time and passenger flow on and off trains. Our maximum deliverable was partially met with a nice visualization of the data; however getting all trains simulated was not a practical possibility. The visualization was completed not because of ease of getting the minimum deliverable done but because it was necessary to have a comprehensible output and previous experience with creating a simple webpage made it easy to create relatively quickly. An interesting point of our simulation is that variation may occur based on a race condition of passengers and trains arriving in a station at the same minute. We feel this is accurate to life because a person may arrive in the station in the same minute as a train and not arrive on the platform in time. This can cause longer loading times for later trains and therefore unscheduled delays. The benefit of using an event based clock was confirmed, in that minutes in the beginning of the simulation run much longer because more passengers and trains must communicate with the clock, but speed up later. The decision prevented our simulation being limited in number of passengers and trains while slowing down the simulation for unnecessarily long minutes.

Reflection on Design and Labor Division

The best decision in the design was the way we divided the modules and documented interfaces before programming. While the modules needed to be very thoroughly tested when integrated, it was still possible to do basic testing of each module before integration. Next time, however, we would look for ways to incorporate Erlang OTP behaviors that may have helped with clarity or reusability of our code overall, and we would also try and find cleaner ways of communication than what is currently implemented. The division of modules made it very easy to divide labor, and theoretically made putting it all together very simple; however because of various conflicts and unforeseen circumstances, it was very difficult to get together to work out small issues together. In the future it would be helpful to plan more scheduled meetings in case of cancellation.

Bug Report

Our hardest bug was that we had a Heisenbug presenting itself as not all processes ending at the end of the simulation occasionally. We created a lot of testing messages to be outputted to try and figure out what might still be running and found that procs_alive was not terminating. Then we made more testing messages and realized that there were passengers still alive that were waiting in stations when no more trains were coming. The total process of finding this cause took a few days, but there was a quick fix in adding a kill all remaining passengers in stations when simulation ends when all trains are done. In retrospect, if we had had better test outputs and a greater variety of test inputs originally and thought sooner to separate all types of processes instead of thinking just all our processes were still running, we would have found the bug significantly faster.

File Overview

Our code and its revision history may be browsed at:
https://github.com/hannahcclark/SiMBTA

SiMBTA folder

- carto.erl
  Presents a “map” of the T-lines (only Red line currently) for the other modules to use.
- clock.erl
  Maintains the simulation’s concept of time and notifies objects watching when it changes.
- main.erl
  Parses input specifications for and runs simulation.
- output.erl
  Handles output to send to results file for simulation.
- passenger.erl
  Imitates a passenger taking the T. It is either in a station or on a train while in existence.
- station.erl
  Imitates a station of the T. It notifies passengers in itself when trains arrive and manages trains passing through.
- train.erl
  Acts as a train on the T. It notifies passengers on itself and the stations when arriving at the stations.
- Viz folder
  - index.html
    HTML file to display the visualization. Contains the canvas, average form, and other static elements.
  - main.css
    Stylesheet to set page layout. Used very minimally as most display code is done using the canvas API.
  - main.js
    Contains all code pertaining to loading the file, parsing it, and then displaying the visualization itself.

  Other files contained in the directory are libraries used to support this code.

How to Run the Program

1. Compile the program using
   ```
   erlc *.erl
   ```
   This ensures that all files are up to date before running.
2. Open the erlang shell using `erl`.
3. Run the following command:
   ```
   main:run(“IN.txt”).
   ```
   The argument is a string of the file to read for input, which may be changed.
4. Allow the program to run. Once it terminates, the output is stored in outFile.txt.
   In the event that the program does not terminate, remain calm; you have encountered a Heisenbug. Terminate the simulation with Ctrl+C, and abort. Run the simulation again and you are likely to encounter success.
5. Run a webserver in the main directory of the project. Python 2 makes this easy using the command:
   ```
   python -m SimpleHTTPServer
   ```
   Note: The output file will not be able to be read via Javascript unless the file is run on a web server. Do not simply open index.html. And be sure not to run the server in the Viz/ directory.
6. Open the Viz folder in a browser (most likely at `http://localhost:8000/Viz/`). You may now view the visualization and its associated data.

Sample Files Provided:
test.txt - small sample input file
IN.txt - larger sample input file