César Robinson

CS6310 – Assignment 6

October 28, 2018

New Requirement changes

Class Diagram Updates

Bus Changes

For this requirement, I intend to allow the user to enter these changes via the UI. Visualizing the available buses in the system, as well as their current routes and possible new route/stop options. I have declared new methods in the TransitSystem class that will call on additional methods on each individual bus and set the route, nextLocation, passenger capacity and/or speed as needed. These changes will only take effect after the bus has arrived at its next destination.

Passenger Exchange

This new requirement considers how passengers utilize and navigate through the system. In order to implement it, I have added additional instance variables (ridersArriveHigh, ridersArriveLow, ridersDepartHigh, ridersDepartLow, ridersOnHigh, ridersOnLow) to the stop class, as well as additional variables (ridersOffHigh, ridersOffLow) to the bus class. The expectation is that all of those values will be provided at runtime (via separate file or as additional parameters in the add_stop/add_bus commands). Once the objects have been instantiated the system will call on some new methods to handle the new passenger exchange requirements. These methods will derive the communication

between the activeBus and the activeStop and perform the steps described by the client to simulate rider activity.

System Efficiency

I will enable functionality in the UI to update each constant, and to request the System Efficiency on demand. I plan on doing this by adding 3 new methods (return type Double) to the Simulation System (waitingPassengers, busCost and systemEfficiency). The system already contains a hashmap of all the route, stop and bus objects on the simulation; upon request, the software will iterate through all the members of the stops hashmap and store the sum of waiting passengers, it will also iterate through the members of the buses hashmap to get the sum of all bus speeds and bus capacities to calculate the bus cost. Both results will then be used to calculate and display the system efficiency. All constants will be initialized to 1, and the customer will be able to set() different values as well.

Replays

In order to satisfy the replay requirement, I have created a new class called SystemState. This class will have 3 instance variables of type object (bus, route, stop) and will be instantiated by the TransitSystem class. The SystemState instances, called states, act as a snapshot of the activeBus (includes passenger count, current location, etc), activeRoute (includes the stops and location within the route, etc) and activeStop (contains number of passengers waiting, etc) during a move_next_bus event. Once one move_next_bus event has been triggered, the state will be created by invoking the transitSimulation's saveState method from within the eventQueue. This will populate the states hashmap<Integer,SystemState> which will be updated as needed every time a new

move_next_bus takes place. The states will be stored based on the rewind number (1-3) and will keep at most 3 consecutive previous states at all times.

Now, for the user to invoke a rewind, a button will be enabled in the UI, upon action, the rewind method in the TransitSystem class will be invoked and it will first confirm that the user has not requested more than 3 consecutive rewinds. If that is not the case, it will then perform a set of operations to restore the attributes of bus/stop based on the data from one of the saved states (state selected based on the rewindCount, 1-3) and it will update the rewindCount=+1. If the customer has reached the 3 consecutive rewind max, then a move_next_bus will trigger the transitSimulations' resetRewindCount(), which would allow the user to once again rewind up to 3 consecutive times.

Sequence Diagram Updates

As part of the requirements, we need to make some modifications to the move_next_bus() flow. In my case, some of these changes are derived from the newly received passenger management requirement. The exchange begins after the system has found what the current stop (activeStop) on the route the bus (activeBus) is currently at. Below is a brief/superficial definition of what the methods are supposed to do:

Step 1.

activeStop.addWaiting() → based on the values of RidersArriveHigh and RidersArriveLow, return Integer and stores value in Integer ridersArrive.

activeStop.setWaiting(ridersArrive) → updates the value of waitingGroup with the random value of ridersArrive.

Step 2.

activeBus.departingRiders() → based on the values of RidersOffHigh and RidersOffLow, return Integer and stores it in Integer ridersOff.

activeBus.updateRiders(ridersOff, false) → important here, the Boolean parameter indicates whether the passengers are getting on or off, in this case false means they are getting off. This method has built in logic to ensure the value of riders is physically logical (>= 0) after ridersOff have been removed from the bus.

activeStop.updateTransfers(ridersOff, true) → similar to the previous method, a Boolean allows for the logical implementation to follow the correct path. In this case, true means that passengers are being added to the transfersGroup.

Step 3.

activeStop.catchingBus() → based on the values of RidersOnHigh and RidersOnLow, return Integer and stores it in Integer ridersOn.

activeBus.updateRiders(ridersOn,true) → true here means passengers are now getting on the bus, so the method ensures there is enough capacity for all of them and handles the case where there is not enough space.

activeStop.updateWaiting(ridersOn) → updates the size of the waitingGroup by decreasing the number of riders that got on the bus.

Step 4.

activeStop.leavingStop() → based on the values of RidersDepartHigh and RidersDepartLow, return Integer and stores it in Integer ridersDepart.

activeStop.updateTransfers(ridersDepart, false) → this time, the updateTransfers method reacts to passengers having arrived at their destination or beign added to the waitingGroup as they wait for their next bus.

After all steps have been completed, the system calculates the next stop and travel time in the route as it did previously.

System Efficiency

For the system efficiency requirement, we are using some methods internal to the mts:TransitSystem object that will allow us to get the waitingGroup size for each stop in the system by looping through the stops hashmap, as well the averageSpeed and passengerCapacity for each bus by similarly looping through the buses hashmap. In combination, those two methods will provide the input to the systemEfficiency() operation which will provide the results that will be returned to the user.