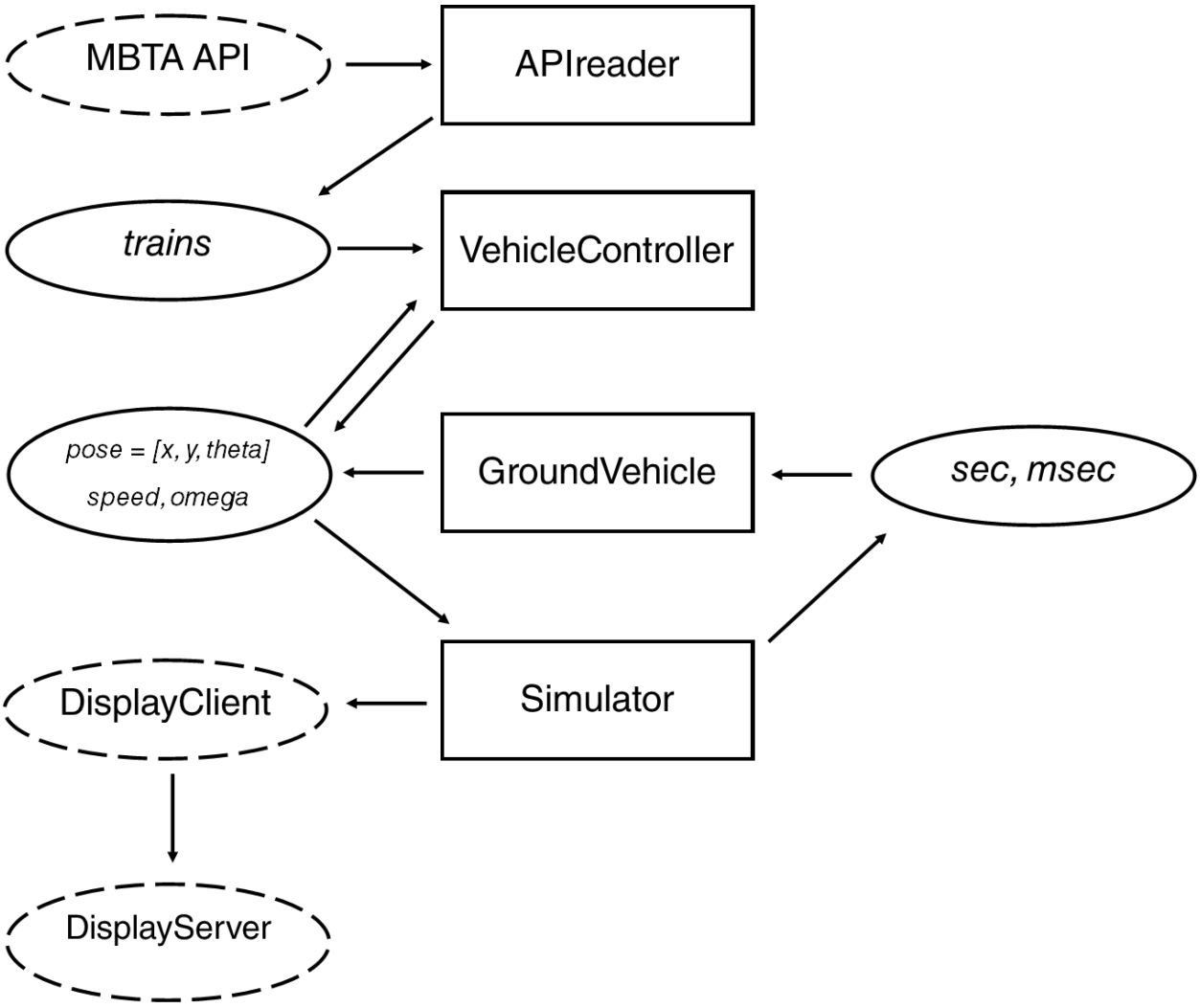
16.35 SPRING 2016 – MBTA RED LINE VISUALIZER – ALLAN KO

Overview

I describe below the software requirements for a multithreaded program which visualizes the motion of trains on the Massachusetts Bay Transportation Authority (MBTA) Red Line track, using real-time location data retrieved from the MBTA API.

The general architecture of the system is depicted below. Rectangular boxes denote threads, solid ovals denote shared variables, and dashed ovals denote other objects. An arrow pointing from an oval to a rectangle denotes that the thread reads from the shared variable or object. An arrow pointing from a rectangle to an oval denotes that the thread writes to the shared variable or object. Notice that threads never communicate directly with each other; they only communicate via shared variables. There is a cycle between GroundVehicle and Simulator, but Simulator never accesses both shared resources simultaneously; so there is no hold and wait condition and no possibility for deadlock.



In brief, the APIreader thread repeatedly queries the MBTA API and updates a *trains* dictionary, which contains train IDs and position information for every train currently on the Red Line. The VehicleController thread reads from the *trains* variable and updates the position and speed of the GroundVehicle to match the incoming data. Simulator reads the GroundVehicle positions, sends the GroundVehicle information to the DisplayClient (which forwards it for display on the DisplayServer), and increments an internal clock. GroundVehicle reads from that internal clock in order to extrapolate the GroundVehicle's motion between API calls.

PS: this will look incredibly boring and slow until you realize that each little GroundVehicle triangle represents a massive subway train, trundling through Boston in real-time.

Run Instructions:

1. Load all files into your directory of choice
2. Open two command prompts or terminals, and navigate to your directory in both.
3. In the first terminal, type `python DisplayServer.py`. An empty DisplayServer window will pop up.
4. In the second terminal, type `python Simulator.py` to automatically run the program. The DisplayServer window will populate with GroundVehicles representing trains on the MBTA Red Line.

Software Requirements:

1. GroundVehicle class
    1.1. GroundVehicle shall be a thread with mutually exclusive access to the internal variables defined in Section 1.2.
    1.2. Variables
        1.2.1. GroundVehicle shall contain the following variables, describing the GroundVehicle's longitude, latitude, orientation, speed, and angular velocity, subject to a standard Cartesian coordinate system.
        1.2.2. *x* shall represent the longitude of the GroundVehicle's position
            1.2.2.1. *x* shall be a float
        1.2.3. *y* shall represent the latitude of the GroundVehicle's position
            1.2.3.1. *y* shall be a float
        1.2.4. *theta* shall represent the orientation of the GroundVehicle's position.
            1.2.4.1. *theta* shall be a float
            1.2.4.2. *theta* shall be in the range [-pi, pi)
        1.2.5. *s* shall represent the linear speed of GroundVehicle
            1.2.5.1. *s* shall be a float
        1.2.6. *omega* shall represent the angular velocity of GroundVehicle
            1.2.6.1. *omega* shall be a float
        1.2.7. *sim* shall be a reference to the Simulator object in which GroundVehicle is running.

1.3. Constructor: GroundVehicle(*pose*, *s*, *omega*)
  1.3.1. the GroundVehicle constructor shall take three arguments: *pose*, *s*, and *omega*.
    1.3.1.1. *pose* shall be a list of three values [*x, y*, *theta*] representing the initial x-position, y-position, and angular orientation of the GroundVehicle, where *x*, *y*, and *theta* are subject to the requirements detailed in Section 1.2.
    1.3.1.2. *s* shall be a float representing the initial speed of the GroundVehicle, subject to the requirements in 1.2.5.
    1.3.1.3. *omega* shall be a float representing the initial angular velocity of GroundVehicle, subject to the requirements in 1.2.6.
1.4. getPosition()
  1.4.1. the getPosition method shall take no arguments.
  1.4.2. the getPosition method shall return a list of three values [*x*, *y*, *theta*] representing the GroundVehicle's longitude, latitude, and angular orientation, subject to the requirements designated those variables in Section 1.2.
1.5. getVelocity()
  1.5.1. the getVelocity method shall take no arguments.
  1.5.2. the getVelocity method shall return a list *vel* of three values [*sx, sy, omega*] representing the GroundVehicle's x-velocity, y-velocity, and angular velocity.
    1.5.2.1. *omega* shall be subject to the requirements in 1.2.6.
    1.5.2.2. *sx* shall be a float representing the x-component of GroundVehicle's linear velocity, calculated according to **Equation 1**:

Equation 1: $sx = s*\cos(theta)$

    1.5.2.3. *sy* shall be a float representing the y-component of GroundVehicle's linear velocity, calculated according to **Equation 2**:

Equation 2: $sy = s*\sin(theta)$

1.6. setPosition(*pose*)
  1.6.1. the setPosition method shall take a single argument, *pose*, subject to the requirements in 1.3.1.1, and set GroundVehicle's corresponding internal variables to the values in *pose*.
    1.6.1.1. if *pose* does not meet requirements 1.3.1.1, setPosition shall raise a IllegalArgumentException.
  1.6.2. if *theta* falls outside the constraints defined in 1.2.4.2, *theta* is set to the value within constraints that represents the same angle as the given *theta* argument.
1.7. setVelocity(*s*, *omega*)
  1.7.1. the setVelocity method shall take two arguments *s* and *omega* subject to requirements 1.2.5 and 1.2.6 respectively.

1.7.2. the setVelocity method shall set GroundVehicle's internal variables *s* and *omega* to the given arguments *s* and *omega*

1.8. controlVehicle(*c*)

1.8.1. the controlVehicle method shall take a single argument *c* representing a Control object.

1.8.1.1. If *c* is None, controlVehicle shall do nothing.

1.8.1.2. if controlVehicle is given a wrong argument type that is not None, controlVehicle shall raise an AttributeError.

1.8.2. the controlVehicle method shall set GroundVehicle *x, y, s* and *theta* equal to the *lon, lat, s,* and *theta* (respectively) of the given Control object *c*.

1.9. advance(*sec*, *msec*)

1.9.1. the advance method shall take two arguments *sec* and *msec* and, based on the GroundVehicle's current state, update the GroundVehicle's internal state after an elapsed time *sec* + *msec*/1000 based on a Newtonian kinematic model, described by **Equation 3**:

1.9.1.1. *sec* shall be an integer representing an elapsed time in seconds since the last time advance was called.

1.9.1.2. *msec* shall be an integer representing an elapsed time in milliseconds since the last time advance was called.

1.9.1.3. GroundVehicle's internal state shall be updated according to **Equation 3**, for $t = sec + msec/1000$, and where *x*, *y*, and *theta* represent the x-position, y-position, and angular orientation in Cartesian coordinates, and *x0*, *y0*, and *theta0* represent the initial x-position, y-position, and angular orientation, and *s* and *omega* represent the linear speed and angular velocity.

Equation 3

If *omega* is nonzero:

$$x = s/omega * (\sin(omega*t + theta0) - \sin(theta0)) + x0$$
$$y = -s/omega * (\cos(omega*t + theta0) - \cos(theta0)) + y0$$
$$theta = omega*t + theta0$$

If *omega* = 0 :

$$x = st*\cos(theta0) + x0$$
$$y = st*\sin(theta0) + y0$$

1.10. run()

1.10.1. the run method shall start the thread and take no arguments.

1.10.2. the run method shall access the GroundVehicle's Simulator (*sim*) and retrieve the current time, then advance the GroundVehicle's position using the advance method described in 1.9.

2. Simulator class

2.1. Simulator shall be a thread with mutually exclusive access to the internal variables defined in 2.2.

2.2. Variables

2.2.1. Simulator shall contain the following variables, representing the time, the GroundVehicles being simulated, and the APIreader.
2.2.2. *currentSec* shall represent the seconds component of the current Simulator time.
2.2.2.1. *currentSec* shall be an integer
2.2.3. *currentMSec* shall represent the milliseconds component of the current Simulator time.
2.2.3.1. *currentMSec* shall be an integer
2.2.3.2. *currentMSec* shall be in the range [0, 999]
2.2.4. *gvList* shall represent the list of GroundVehicle objects in the Simulator.

2.3. Constructor: Simulator()
2.3.1. The Simulator constructor shall take no arguments.
2.3.2. The Simulator constructor shall initialize *currentSec* and *currentMSec* as 0, and *gvList* as the empty list.

2.4. getCurrentSec()
2.4.1. The getCurrentSec method shall take no arguments.
2.4.2. The getCurrentSec method shall return *currentSec*.

2.5. getCurrentMSec()
2.5.1. The getCurrentMSec method shall take no arguments.
2.5.2. The getCurrentMSec method shall return *currentMSec*.

2.6. advanceClock()
2.6.1. The advanceClock method shall take no arguments.
2.6.2. The advanceClock method shall increment the Simulator time by 10 milliseconds every 10 milliseconds, where time is defined in **Equation 4**.

Equation 4: $time = currentSec + currentMSec / 1000$

2.7. addGroundVehicle(*gv*)
2.7.1. The addGroundVehicle method shall take a single argument, GroundVehicle object *gv*.
2.7.2. The addGroundVehicle method shall append *gv* to the end of *gvList*.

2.8. removeGroundVehicle(*gv*)
2.8.1. the removeGroundVehicle method shall take a single argument, a GroundVehicle object *gv*.
2.8.2. The removeGroundVehicle method shall remove *gv* from *gvList*.
2.8.2.1. If *gv* is not in *gvList*, the removeGroundVehicle method shall throw a ValueError.

2.9. inList(*gv*)
2.9.1. the inList method shall take a single argument, a GroundVehicle object *gv*
2.9.2. the inList method shall return True if *gv* is in *gvList* and False otherwise.

2.10. run()
2.10.1. The run method shall start the thread and take no arguments.
2.10.2. The run method shall send the position of every GroundVehicle in *gvList* to the display client, advance the clock according to advanceClock(), and repeat indefinitely.

3. DisplayClient and DisplayServer classes
   3.1. The DisplayClient and DisplayServer classes shall be provided by the 16.35 staff.
   3.2. The DisplayServer shall be started by running “python DisplayServer.py” in the console.
   3.3. The x-axis of the DisplayServer shall represent longitude, and the y-axis of the DisplayServer shall represent latitude.
   3.4. The DisplayServer shall correspond to a equirectangular map projection of the Boston area.
      3.4.1. The DisplayServer x-axis (longitude) shall be bound by [-71.18, -70.98]
      3.4.2. The DisplayServer y-axis (latitude) shall be bound by [42.20, 42.45]
   3.5. The DisplayClient shall be created when Simulator.py is executed in __main__, detect the DisplayServer, and automatically connect.
4. Control class
   4.1. Control shall be an object representing a GroundVehicle control signal (position, speed, and angular orientation).
   4.2. Variables
      4.2.1. Control shall have the following variables, representing position, linear and angular orientation.
      4.2.2. *lon* shall represent the longitude.
         4.2.2.1. *lon* shall be a float.
      4.2.3. *lat* shall represent the latitude.
         4.2.3.1. *lat* shall be a float.
      4.2.4. *s* shall represent the linear velocity.
         4.2.4.1. *s* shall be a float.
      4.2.5. *theta* shall represent the angular orientation.
         4.2.5.1. *theta* shall be a float.
         4.2.5.2. *theta* shall be in the range [-pi, pi)
   4.3. Constructor: Control(*lon, lat, s, theta*)
      4.3.1. The constructor shall take four arguments, *lon*, *lat*, *s*, and *theta*, subject to requirements 4.2.2, 4.2.3, 4.2.4, and 4.2.5.
         4.3.1.1. If *lon*, *lat*, *s*, or *theta* do not meet requirements 4.2.2, 4.2.3, 4.2.4, and 4.2.5, the constructor shall raise an IllegalArgumentException.
      4.3.2. The constructor shall initialize *lon*, *lat*, *s*, and *theta* equal to the given arguments.
   4.4. getLoc()
      4.4.1. the getLoc method shall take no arguments and return a tuple *lon, lat*.
   4.5. getSpeed()
      4.5.1. the getSpeed method shall take no arguments and return *s*.
   4.6. getTheta()
      4.6.1. the getTheta method shall take no arguments and return *theta*.
5. VehicleController class

5.1. VehicleController shall be a thread that references a Simulator and a GroundVehicle, and sends Control objects to the GroundVehicle based on the desired trajectory of the GroundVehicle.

5.2. Variables

5.2.1. VehicleController shall have the following variables, representing references to the associated Simulator, GroundVehicle, and APIreader.

5.2.2. *gv* shall be the GroundVehicle associated with VehicleController.

5.2.3. *sim* shall be the Simulator associated with *gv*.

5.2.4. *api* shall be the APIreader associated with *sim*.

5.2.5. *id* shall be the train id associated with *gv*.

5.2.6. *timelastsynced* shall be the epoch time of the most recent update to VehicleController

5.3. Constructor: VehicleController(*sim*, *gv, api, id*, *timelastsynced*)

5.3.1. The constructor shall take five arguments, *sim, gv*, *api*, *id*, and *timelastsynced*, and initialize the corresponding internal variables to the given arguments.

5.4. getControl(*lon*, *lat*, *bearing*, *time*)

5.4.1. getControl() shall take as arguments *lon*, *lat*, *bearing*, *time* as defined in 6.3.3.

5.4.2. if *time* is equal to *timelastsynced*, getControl shall return None.

5.4.3. otherwise:

5.4.3.1. if *lon, lat* is equal to *gv*'s current position = [*x, y*], getControl shall return Control(*lon*, *lat*, 0, *theta*) (where *theta* is calculated by **Equation 5**).

Equation 5: $theta = \text{clamp}(\pi^* (90 - bearing) / 180)$

where the clamp() function calculates the equivalent angle in the range [-pi, pi)

5.4.3.2. otherwise, getControl shall return Control(*lon, lat, s*, *theta*) where *s* and *theta* are calculated by **Equation 6** and **Equation 5** respectively.

Equation 6: $s = \text{sqrt}((lon - x)^2 + (lat - y)^2) / (time - timelastsynced)$

5.4.3.3. getControl shall then set *timelastsynced* equal to *time*.

5.5. run()

5.5.1. the run method shall take no arguments and start the VehicleController thread.

5.5.2. the run method shall access *api,* retrieve the data ([*lon, lat, bearing, time*]) for train *id*, calculate a Control using getControl (5.4), and apply that Control to the corresponding GroundVehicle (*gv*) according to 1.8.

6. APIreader class

6.1. APIreader shall be a thread that periodically accesses the MBTA API to retrieve the current train positions on the Red Line.

6.2. The complete MBTA API documentation is available here: http://realtime.mbta.com/Portal/Content/Documents/MBTA-realtime_APIDocumentation_v2_1_1_2016-04-12.pdf

6.3. *trains* variable

6.3.1. *trains* shall be a dictionary of all trains on the Red Line, keyed by *id* (corresponding to *vehicle_id* in the API).

6.3.2. *trains* shall have mutually exclusive access.

6.3.3. For each *id*, *trains*[*id*] shall be a list [*lon, lat, bearing*, *time*], where:

6.3.3.1. *lon* denotes the longitude of train *id* (corresponding to *vehicle_lon* in the API)

6.3.3.2. *lat* denotes the latitude of train *id* (corresponding to *vehicle_lat* in the API)

6.3.3.3. *bearing* denotes the bearing of train *id* in degrees clockwise from true north (corresponding to *vehicle_bearing* in the API)

6.3.3.4. *time* denotes the time at which the *lon* and *lat* were updated, in epoch time (corresponding to *vehicle_timestamp* in the API)

6.4. Constructor: APIreader()

6.4.1. The APIreader constructor shall take no arguments and initialize *trains* as an empty dictionary.

6.5. getTrains()

6.5.1. the getTrains method shall take no arguments and return the *trains* dictionary.

6.6. APIget()

6.6.1. The APIget method shall take no arguments

6.6.2. The APIget method shall call the vehiclesbyroutes query in the MBTA API to retrieve a json data file, containing the trains, train positions, and train metadata for all trains on the Red Line

6.6.3. The json data file retrieved from the API shall meet specifications defined in section 4.4.4 of the API documentation (see requirement 6.2).

6.6.4. If the API query fails, APIget shall raise an IllegalArgumentException

6.7. APIupdate(*resp*)

6.7.1. The APIupdate method shall take a single argument, *resp*.

6.7.1.1. *resp* shall be a json data structure specified by requirement 6.6.3.

6.7.2. The APIupdate method shall use the vehiclesbyroutes query in the MBTA API to retrieve a json data file, containing the trains, train positions, and train metadata for all trains on the Red Line.

6.7.3. The APIupdate method shall update the data in the *trains* variable to match the json file retrieved from the MBTA API.

6.8. run()

6.8.1. the run method shall take no arguments and start the APIupdate thread.

6.8.2. the run method shall call APIupdate repeatedly, sleeping for 10 seconds between calls.