CS 452 - Final Project

By: Raymond Wan (20507029) and Tim Pham (20511526)

Running

Please follow the instructions in README.md for building and running, but **please restart the box and train controller before running**.

Place two trains down at entries in **reverse orientation**, either triggering the sensors or near them. Call the **set** command to initialize the trains at those sensors (if the trains are not triggering them already, please trigger them).

Call **go <train_num> c14** and let the train get halfway into the path before calling **go <train_num> e8** to start the next one. The trains will move to random destinations after.

Commands:

ut < a | b >: Use track **a** or **b**.

set <train_number> <sensor>: Initialize a train's first sensor (for locating purposes). Once the train hits that sensor, velocities and deltas will start computing.

tr <train_number> <train_speed>: Change the train speed of a given train

rv <train_number>: Reverse a train_number and reaccelerate it back

sw <switch_number> <switch_direction>: Changes switch with number switch_number to switch_direction

res <train_number> <track_node_string>: Reserve a segment of the track for the given train e.g. res 24 c11

fr <train_number> <track_node_string>: Free a segment of the track for the given train e.g. res 24 c11

go <train_number> <sensor>: Make the train go to a specified sensor
e.g. go 24 c11

g: Sends the GO command

x: Sends the STOP command

i: Initializes the switches

q: Quit the program

Project Description

Our project aims to have three trains running at once performing deliveries to sensors and returning back to home sensors. This is a simple extension of TC2 with slightly different behaviour and having one more train. Although this was the goal of our project, we ran into unexpected roadblocks that put us very far behind the proper pace to complete the project. Please see the Bugs/Unimplemented section for details.

Task Structure

Inspired by a popular state management architecture for called Redux, we rearchiteched the tasks to follow a similar pattern that can be described as a pub-sub pattern. There now exists a Dispatcher task that handles all event publishes and subscriptions. For example, the Sensor server will now register itself with the Dispatch server and make announcements when a sensor has been triggered. Any task that is subscribed to this event will receive this information and will be able to act upon it. We use a pub-sub pattern to simplify the management of changing state across our system. For example, when we send commands that affect the train speed and switch positions,

Track Reservations

To reserve a segment of the track, one must specify a node in the track. What is reserved is the edge that connects the specified node and the next node ahead. If the specified node is a branch, then both parts of the branch are reserved to avoid collisions at a branch.
It may be the case that a train will try to reserve the track from the opposite end of the edge. Although the track has one physical edge between two nodes, there are two logical edges that are used in the track graph. This is not a problem for our implementation because when we check if an edge is free, we check whether the edge coming from both sides are free. If one of them is taken, then we know the segment of the track is not free.

To store this information, we simply have a fixed sized array of size TRACK_MAX, where the values in each represent the train that has reserved the edge (starting from that node).

Our model for reservations is quite conservative. We reserve enough distance for the stopping distance of the train and then a buffer length. The buffer length is the distance between the sensor that the train is about to hit and the next sensor in the path. Since we unreserve segments of the track between sensor hits, the buffer distance is required to always have enough stopping distance reserved. Therefore, the train will never overdrive with this reservation policy. However, distances between sensors can become quite large, and so it could be the case that there is a lot of redundant track reserved that could have been safely reserved by another train.

Pathing and Navigation

Building off the Dijkstra's algorithm from TC1, we added logic for the algorithm to take “reverse” edges (that requires trains to reverse in order to take them) and included reservations to the weighting of edges. Furthermore, to resolve around reserved segments of the track that aren’t owned by the train that is requesting for a new path, we also check whether an edge is free during the computation of our version of Dijkstra’s algorithm, if an edge is not free, it is simply ignored in the algorithm.

With these additions, trains will now take the shortest path using as much unreserved track as possible. It should be noted that this path may not be the fastest (as it is possible for a longer path to be faster if it has significantly fewer stop-and-gos than its shortest-path counterpart).

Train Driving

After computing a path for a train, the path is parsed to determine the necessary switches and reverses that must be made. Points in the path where we require a reverse are called “checkpoints”. We divide up the train driving to handle one checkpoint at a time. The distance to the checkpoint is obtained (from the computed path) to determine whether to use a long move or a short move. Long moves are performed in the same way as TC1. Short moves, however, are computed by first calibrating short move distances at various delays and performing linear regression to approximate the function. A desired distance is passed to the function to obtain the necessary delay to move that distance.

Initial reservations will then be made before moving. If any initial reservation fails, the train will transition into an idle state and use a random backoff before retrying to reserve. If there is no path available i.e. all reservations around the train are taken, then the train will remain in an idle state and use random backoff to retry computing a path.

Collisions and Deadlocks

Trains will attempt to reserve track to cover its current stopping distance. While traveling, if the train fails to acquire a piece of track, the train will immediately stop, which can be done safely since the train has already acquired enough track to stop. The train will then sleep for a random amount of time before recomputing its path to its destination. The result of this behavior prevents trains from colliding since they will only travel onto track that they own and will acquire enough space to stop before potentially entering another train’s path.

Because trains only acquire just enough track to move forward as well as stop and give up track during track reservation failures, deadlocks are not possible. However, it is possible for trains to livelock if they are constantly running into each other’s path. These situations are avoided by the random sleep time that occurs after the train stops. Trains will recompute

their paths at different times and eventually a train will navigate their way out of the situation before its opposing train can move and block it again.

Bugs/Unimplemented

While our goal was to have three trains running, this was unfortunately not accomplished. We fell quite far behind when the TC2 milestone was due and spent some time trying to complete that before moving onto three trains. The biggest issue that we faced was debugging deadlocks in our task structure. As explained in a previous section, we used a non-conventional task structure that wasn't described in class. We spent approximately 3 days trying to debug task deadlocks that would occur during certain configurations of our project running.

Some things that we wished we could have done:

- Have better train positioning. Our train position is not as good as it could be, due to inaccuracies in calibration and using a model that is not complex enough
- Handling failures better i.e. lost trains and failed switches. Although we have a structure that can handle switch failure, we could not implement it fully. Similarity for lost trains, we could not get around to handling the cases that could occur.
- Continuous reservation model. Our reservations were very conservative, so we couldn't get the trains as close as possible.