CS 452 Project

Eric Woroshow (20165466)
Daniel Zapallow (20203316)

August 1, 2008

1 Executables and Instructions

Our kernel and module sources are contained in /u8/eworoshow/cs452/p. The provided Makefile compiles and posts the "dekernel" module to the EOS systems.

At boot time the operating system automatically starts the train control process. Before controlling the trains, however, the track must be specified by pressing the A or B key at the track prompt. The "init", "tr", "rev", and "sw" commands then control the trains and track as previously specified. In addition, the "rand" command provides a stream of random destinations for a train; entering this command a second time ends the stream.

2 Objectives

The overall objective for the project was to create a game in which one user-controlled train is pursued by two (or more) computer-controlled aggressors who attempt to trap the user train. Achieving this goal involved tracking, routing, and preventing collisions between trains. More specifically, the project required that we be able to cooperatively route multiple trains to independent destinations while avoiding collisions and detecting entrapment.

Of these goals, we successfully achieved multiple train independent routing. This functionality is built on excellent tracking, good routing and functional (but sub-optimal) collision detection and avoidance features. We lacked the time, unfortunately, to implement both cooperative routing for opponents and the game meta-structure.

3 Implementation

Our implementation comprises three semi-independent components: tracking, routing, and collision avoidance. Train tracking is the foundation, providing the track location knowledge necessary to accomplish routing and collision detection.

This section first presents an overview of the process structure and component interaction. It then delves into the specific implementation details of each component.

3.1 Structure

Figure 3.1 shows the process structure with a single engineer initialised on the track. (Additional engineers simply duplicate the *Engineer*/*Heartbeat*/*TrainDisplay* structure.) The purpose of each process is explained below.

- *Sensors*: Requests sensor data from the track then sends it, via the *SensorsCourier* process, to the track manager.
- *TrackManager*: Sets switches. Batches, then broadcasts to observers via the *EventBroadcaster*, switch change and sensor hit events. Provides an interface to the track model, exposing track connectivity and distance queries and a path finder.
- *TrackDisplay*: Listens for track events and displays them on the terminal.
- *TimeDisplay*: Displays the real-time clock.
- *TrainDisplay*: Queries an engineer (at 10 Hz) for its train data then displays the train's position, speed, and time to next sensor on the terminal.
- *Engineer*: Issues commands to a train. Tracks and routes a train. Avoids collisions with other trains by interacting with the engineer coordinator to reserve sections of the track. A *Heartbeat* process causes the engineer to update its speed and position estimates at 40 Hz.
- *EngineerCoordinator*: Provides a track reservation system for collision avoidance. Provides a sensor bidding mechanism to facilitate multiple train tracking. Communication with engineers during the bidding process is managed via the *CoordinatorCourier* process to avoid send blocking.
- *DestinationGenerator*: Provides a stream of random destinations to an engineer.
- *TerminalInterface*: Reads, parses, and dispatches user commands from the terminal. Train commands are sent to the engineer controlling the train. Track commands are sent to the track manager.

3.2 Tracking

Tracking a train is the responsibility of the train's engineer, in conjunction with the engineer coordinator. Tracking is divided into two phases: initialisation and movement.

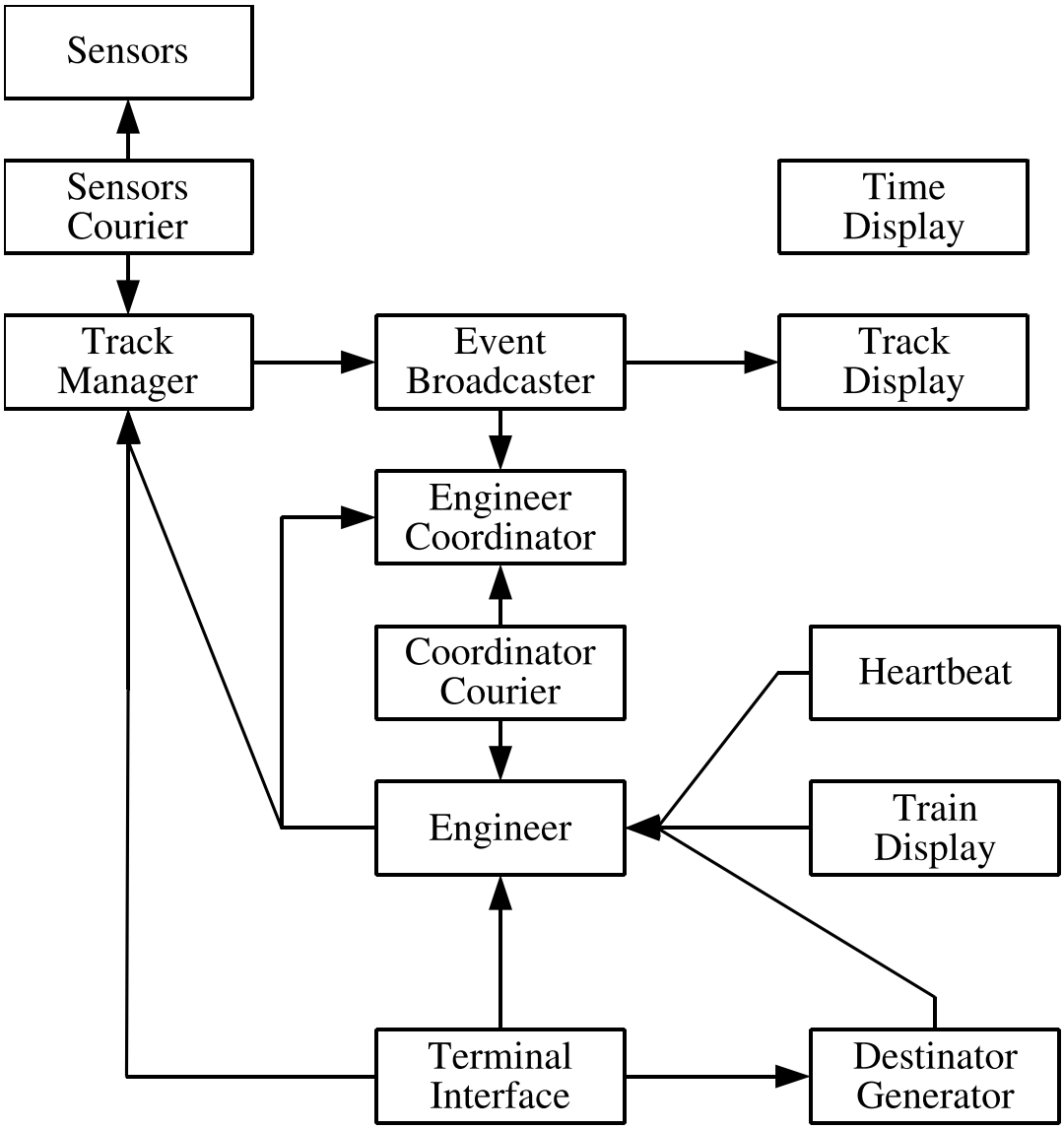

Figure 1: Process structure. Outgoing arrows indicate sending messages.

3.2.1 Initialisation

Before tracking can begin, the initial position of the train must be discovered; the initialisation sequence performs this task. The train moves forward[1] until a sensor fires, at which point its position is known. If no sensor fires inside an n-second window the train reverses direction. The window, which starts at $n = 2$, increases exponentially as initialisation progresses. This behaviour ensures that, even if a train is placed at a dead end or beside another train, it will (eventually) discover its position.

Since the train stops after a single sensor fires, no other trains may be moving on the track during initialisation. This limitation could, in theory, be overcome with judicious sensor filtering by moving trains but that approach is both error-prone and difficult to implement.

3.2.2 Movement

Once the train's initial location is known, its position must be tracked as it moves around the track. This monitoring has two components: tracking speed/distance and tracking sensor hits.

Precise knowledge of the train's speed is the foundation of tracking distance traveled and also sensors hit. Monitoring speed is a sisyphean task, however, given the continually varying performance characteristics of each train; speed varies as a function of time, track position, train condition, and moon phase. Given these difficulties, the engineers use a dynamic speed model that is calibrated at run-time. An initial speed estimate is updated with actual speeds as the train moves around the track. The engineer stores speeds for each section of the track (sensor to sensor) separately to account for varying speeds across curved and straight sections. Every speed update also feeds into an estimate of the average speed to handle uncalibrated track sections. This model is maintained for every speed number, one through fourteen. The entire model is stored in a massive multidimensional array.

Distance is calculated as a function of speed and acceleration. Each engineer uses a static non-linear acceleration model (discovered through extensive experimentation) to maintain an accurate estimate of intermediate speed during speed changes.

Sensor hits provide knowledge of absolute location on the track to complement the relative positioning provided by speed and distance. The tracker considers a sensor to be "hit" only when its state transitions from off to on, which filters most misfiring sensors. Allocating sensor hits to trains is handled by the engineer coordinator using a bidding system. After receiving a sensor event from the track manager, the coordinator asks each engineer to "evaluate" the sensor. Each engineer ranks the sensor as "expected", "potential" or "unexpected". An expected sensor is the next in line on the train's path (as determined by querying the track manager). A potential sensor is the next next (or next next next) sensor in line or a sensor along an alternative branch. An unexpected sensor is any that does not fall into the previous two categories. Each engineer also provides an estimate of how far it believes its train to be from the sensor being evaluated. A distance window is used to

[1]When a train is said to perform an action it is assumed that the train's engineer issued the command. The terms "train" and "engineer" are used interchangeably henceforth.

reclassify sensors as unexpected when the distance exceeds a certain threshold. The sensor hit is assigned to the engineer providing the best rank and closest distance.

3.2.3 Lost Trains

Occasionally trains become irrevocably lost after missing all the expected and potential sensors. This can happen, for example, when a train gets stuck on the track and is subsequently restarted. When an engineer discovers it is lost (having travelled some large distance without seeing a sensor) it informs the engineer coordinator. The coordinator then assigns the next unexpected sensor to the lost engineer.

This system is effective for a single lost train but is both prone to abuse and fails when multiple trains are lost simultaneously. However, the enormous benefit in recovering lost trains and extreme rarity of multiple lost trains outweigh the problems.

3.3 Routing

Path generation is handled by the track manager. Internally, the pathfinder only supports routing between sensors. It uses the Floyd-Warshall algorithm to generate all the shortest paths. Though the algorithm is $O(n^3)$ where n is the number of sensors, we perform this expensive computation at initialisation time. Path construction then becomes linear in the length of the path.

Path following is handled by the engineers. The engineer is responsible for setting the speed, setting the necessary switches, and reversing through switches. It receives from the track manager a path consisting of a series of sensors, paired with the distance remaining at each step. As the train follows its path, the engineer requests the track manager to set the switches between the upcoming two sensors. The engineer sets the switches early because, otherwise, the train could be on top of a switch when it is set thus derailing the train. If the engineer detects two co-located and opposite sensors in its path (e.g., A1 and A2) it interprets them as a signal reverse through a switch. Once the train has hit the first of the two sensors it waits for the train to fully clear the switch (again to prevent derailment) then sets the switch and reverses the train. When the train's stopping distance is greater than or equal to the distance remaining, the train decelerates to a stop. Routing to offsets, switches and dead-ends is accomplished by manipulation in the path finder of the distance remaining.

Should a train ever deviate from its path (by traversing an unexpected branch, for example) the engineer simply requests a new path to follow.

3.4 Collision Avoidance

Collision avoidance is the shared responsibility of each engineer and the engineer coordinator. The coordinator implements a track reservation system to ensure multiple trains do not enter the same section of track simultaneously. An engineer must successfully reserve the section of track it is about to enter to continue moving forward; if its reservation is denied (i.e., the section is currently owned by another engineer) it must stop.

The coordinator grants (or denies) reservations on a per-sensor basis, then stores the reservations in an array. An engineer reserves its rear sensor then all the sensors through its stopping distance plus a buffer of 15 cm. The reserved sensors cover all potential paths to prevent collisions through unexpected branches. These reservations, which are updated every heartbeat to ensure early collision detection, are designed to prevent both forward and rear collisions. In the case a full reservation is denied, the coordinator still reserves all the sensors up to the point of conflict to prevent other trains from colliding with the stopping train.

When a train stops because its reservations were denied it continually reattempts the reservation. When the reservation is granted (because the blocking train moved) the train starts again at its previous speed. This system obviates the need for a complicated notification system between engineers to resolve collisions. It is limited, however, insofar as a train must come to a full stop before restarting; constant acceleration and deceleration fatally confuses the tracker.

Collision avoidance complicates path finding. One train cannot follow a path through another hence the path finder must, at some level, be aware of train positions. Each engineer, after every sensor hit, informs the path finder of its position. The path finder then disconnects the section of track occupied by the train (sensor to sensor) from the model and recomputes the shortest paths[2]. In addition, each engineer following a path requests a new path after every sensor hit. In this way, trains never route through each other; head-on collisions are automatically resolved with new paths.

This collision avoidance system, though functional, has several deficiencies. First, the reservations are too aggressive, leaving little room to manoeuvre. Trains travelling in a loop may deadlock one another so that none may move. Second, the continuous path finding of each train occasionally leads to suboptimal paths. In the case of a head-on collision, likely only one train should calculate a new path whereas in this implementation both trains back off from the point of contention.

4 Postmortem

This section provides a brief overview of what did and did not work in the implementation of our project.

4.1 What Worked

The systems described in the proceeding sections worked well, except where indicated. The tracking was especially robust which provided a solid foundation on which to build more complex functionality. Using one engineer process per train, rather than a single monolithic process to control every train, was advantageous. This design simplified the code and allowed us to consider each train in isolation.

[2]This frequent recomputation involves rerunning Floyd-Warshall which is massively inefficient. Using Dijkstra's algorithm would have been, in hindsight, a better choice.

4.2 What Failed

Implementing the project involved numerous failed ventures. This section describes several notable failures.

One, we avoided implementing an acceleration model. This decision proved untenable as our speed and distance estimates became unusably crude when the train changed speed. We initially expended considerable effort, however, working around this poor tracking. Eventually we adopted an acceleration model.

Two, we initially lacked a sensor bidding mechanism, instead relying on each engineer individually to accept or reject sensor hits. While effective for a single train, this model quickly degenerates with multiple trains. Again, we spent far too much time working around the limitations of this system before implementing the current, superior bidding system.

Three, we initially implemented routing as node-to-node (where a node is a sensor, switch, or dead-end) rather than sensor-to-sensor. This design complicated both path finding and path following. Constructing the adjacency matrix was difficult. Traversing a heterogeneous path proved to be even more difficult. Ultimately, the marginal benefit of slightly more efficient paths was greatly outweighed by the complexity of the implementation and the design was scrapped.

Four, we were never able to resolve the intermittent sensor failures. Occasionally the sensor module requests sensor data from the track and never receives a response. We tried, but could not find, the cause of the failures. With additional time we could have implemented a sensor keep-alive process to destroy and recreate the sensors process should it become unresponsive.