# CS 452 Train Control Two

Louis A. Burke (laburke) and Taras Kolomatski (tkolomat)

July 18th, 2017

## The Program

Our kernel development defined the possibilities of interactions of tasks in the real-time system. The blocks of our system, either tasks or constituent functions, are short elementary pieces of code, themselves non-monolithic. The first train control milestone produced the fundamental structural unit of the system - a tangled logic across tasks that accomplished the goal of controlling the movement of a train. The second train control milestone implemented protocols for the interaction of two of these fundamental units. This was accomplished by setting straightforward rules, lacking intricacy and far from optimal, to ensure the units interacted correctly - that the trains controlled did not collide. Thence the control of a single train is fundamental in the sense that its parts are not units of function, and that interaction among fundamental units is far more simple then their internal complexity.

## Its Overview

We have anthropomorphised the results of each previous assignment. Assignment zero was about the difficulty of communication. Kernel one was about how the context of human evolution, both biologically and sociologically, resulted in the equilibrium of society and control of its trajectory. Kernel two and three expended on the structure of this society and on the forces that must be counteracted to maintain it. Kernel four observed that the most successful communication was concise and simple.

Having produced the fundamental structural unit in the first train control milestone, we shifted away from the perspective of a human in the context of their society, to the perspective that society is just a byproduct of the inherent ingenuity of human thought. All interactions among people convey only an approximation of the complexity of the thoughts of their participants. The net effects of these sub-optimal manoeuvres could not possibly be the etiology of an individual's complexity.

## The Theme

Russian mathematician Israel Gelfand delivered a lecture for his receipt of the Kyoto prize, which honours the holistic contribution to humanity of the recipient's technical work. Titled, *Two Archetypes in the Psychology of Man*, this lecture was on the the two conflicting human functions of wisdom and intellect. Wisdom is derived from experience, delivered to the individual by society. Intellect is ingenuity in fundamental opposition to the genetic and social conditioning of an individual, as observed from the patterns in society.

The outline of the lecture is as follows: Gelfand provides examples of the serious social conflicts that arise from the discrepancies of these views. This is the conflict of technocracy to the view of technology as evil, the conflict of illustrating the concept of a point to a student in saying that it is *that which has no part*, appealing to intuition, and of taking a formal axiomatic approach that enforces rigour and eschews loose intuition, the conflict of a model to the reality that it is modeling - in all technical disciples (he speaks of his experience in biology, medicine, and computer science). He postulates that the cause of these conflicts is lack of an adequate language that can express both archetypes. For example, Hilbert showed that Euclidean geometry could be placed on fully rigorous ground. If one abandons the classic definition of a point, focusing only on axiomatic relations between a point and other geometric objects, one can purify the theory. Indeed one could take points to be classical planes, and planes to be classical points - the duality of projectile geometry does not care as three generic instances of one defines a unique instance of the other. A mathematician must have both intuitive and axiomatic understanding to conceive new results, hence, although the language differs, there is no fundamental divide between the views. Gelfand explains that mathematicians are capable to and responsible for producing such a language, which must describe the abstract notion of a fundamental unit: indivisible into complex systems, and interacting mutually in a far less complicated manner.

CS 452 is a course which clashes precise theoretical models with imperfect reality, the ability to perform any single task well with the requirement that necessarily less complicated protocols govern their interaction, and the creation of elegant, suddenly inspired, solutions with the inelegant and time consuming act of debugging. Computer science is a human field of study, and, in the course of these introductions, we have explained how the field's problems are reflections of a universal psychological dilemma fundamental to the human condition.

## Calibration

We know that steady state (no command related acceleration) velocity over a segment correlates well with distance traveled when stopping over the segment. In our first demo, we found these values by making a calibration loop every time we requested a train movement. We would record that the stop command be issued a certain number of ticks after a certain sensor activation. Further, we determined the progress of the train in the path it followed by counting sensor activations. Any sensor failures would be fatal to this approach, and it is not optimal to calibrate over the same segments multiple times.

We re-wrote the logic that determines progress in a path, identifying if an unusual sensor activation is the result of skipping a dead sensor. We further introduced a structure to store track calibration data. This structure records which sensors are dead and inter-sensor times. This structure allocates indexed space for storing records for all adjacent sensor pairs, and a constant number of entries for multiple segment times (with dead sensors in the middle). When running the program with no stored calibration data, we process a movement request as follows:

i. We determine if the calibration data has times for every adjacent pair in the list of live sensors on the path (all of our paths are computed as circular, which eliminates the question of what to do in the case of dead endpoints).
ii. If there is insufficient data, then run the train on a calibration route.
iii. If there is sufficient data, calculate the stopping distance and find the last live sensor on the path that can serve as a trigger to stop after a delay (if the path is short and you think that, with few dead sensors, this would be a counterexample of the existence of such a sensor, then recall that all paths are circular).

Because the map of path index to sensor ids is not necessarily injective on a circular path, the attribution logic mentioned above becomes relevant. Running over a calibration route fills in only records that do not already exist, dropping newer duplicate data. We don't loose generality by considering only circular routes without reversing, as this is a program only for calibration and maintaining steady state velocity over a track exit is impossible.

New calibration records were printed to the debug log, which was scp'ed over to our personal machines. We formatted the records to be the lines of C code that caused the records, and placed those code files in the `src/data` directory. There is a global macro to determine if the program is run in calibration mode. If it is set to false, then we run the code in `src/data`. If a calibration record does not exist at run-time, for example at the exits, then an average value is written into the structure.

## Attribution and Reservation

Our attribution and reservations systems are designed to be fairly simple, yet parameterizable. This allows them to be robust, yet effective.

Our attribution system maintains the current and possible next sensors for each train. If two trains are between two sensors then only the first one will attribute to the destination, the second will only attribute there once the first successfully passes it. Then when a sensor is flipped it iterates over all of the trains in the system and decides if it was the one that flipped it. At first it merely checks the immediately following sensors of each train. If after that it can't find any match it will begin searching forward along both sensor

paths to see if the activated sensor might be one of the sensors after a dead sensor. The distance ahead it looks is a specifiable constant.

Our reservation system is fairly straightforward, using a separate server to request to own sections of track and route given other train's reservations. It also provides the ability to steal sections of track to indicate that a train is misbehaving. This notifies the previous owners of those sections and lets them know that they may have to reroute. While this may not always prevent a collision, it helps to at least provide a possible response to unexpected behaviour.

An example of this is spurious sensor firings. The program can't determine which train caused it, but must assume that something did, so it should route the trains away from there - even though all trains were correctly told to avoid that location. Similarly if a train moves too slowly and gets stuck at a position on the track other trains will have to carefully route around it until it is confirmed to have been removed.

## Modularization

In order to test many of the complicated subsystems in this assignment, we have transferred most of our code to a separate directory and designing it to be completely independent of our kernel and its systems. This allows us to unit test this code on our own computers - helping to get tests done when there is a lot of track contention.

## Results

Unfortunately when integrating the many systems involved in this assignment, many kernel limitations were encountered which, coupled with heavy track contention in the lab, resulted in a very unstable demo. While on rare occassions the executable loaded would successfully dynamically avoid collisions and deal with up to 4 simultaneous points of failure, for the most part it did not perform as expected.

## Moving Forward

After adding extensive logs through the GUI to the project we have identified a number of the problem sources and are slowly re-integrating each module one at a time with extensive tests between each module's integration to ensure they work well with previous modules. With new log systems in place bugs are much easier to identify and development has become more streamlined.

## SHA of commits

This submission involved two separate executables, as well as a GUI executable. The basic routing and stopping code can be found on the `tc2_integration` branch with hash `752e5bf0a7bcb6ab4a66e7a1d08e9b3315e085a7`. The dynamic reservation and attribution code along with GUI integration and command parsing can be found on the `dev-tc2` branch with hash `3c8e1374e593d7ca131c74130ae871bf7542ff2b`. The heads of these branches may change over time, but these commits were the ones used for the demo.
The repository can be cloned from:

`gitlab@git.uwaterloo.ca:laburke/cs452_kernel.git` (SSH)
or
`https://git.uwaterloo.ca/laburke/cs452_kernel.git` (HTTPS)

**There is a README in the root directory of the repository which outlines the loading command.** To run the program once it is compiled just restart the machine and run

```
load -h 10.15.167.5 "ARM/user/kernel.elf"
```

to load it.