STRESS:

Stational T-Stop Railing Emergency Sirening System

Jessica Chen, Jared Ni, Gary Wu, Bryan Han

May 2023

Introduction

Dangerous subway environments and a lack of safety procedures plague the subway stations of the major American cities and the commuters who rely on them for transportation. In 2021, the New York Metropolitan Transportation Authority (MTA) reported 1,267 reported track intrusions in 2021 in New York City alone, resulting in 200 collisions and 68 fatalities. Of the 200 collisions, 25 of which are victims shoved in front of incoming New York subway cars. Across the US in 2021, there were 5781 nonfatal train-collision related injuries and 892 deaths. Most if not all of these could have been prevented with the proper installment of safety infrastructures that prevent or detect schemed or accidental track intrusions.

Our group sought to mitigate this urgent issue through our Stational T-Stop Railing Emergency Sirening System (STRESS). Our distributed system consists of physical sensors with LED lights placed on or near the train tracks, a train simulation API that mimics a subway system, and a centralized server to handle communication and all relevant logic. We used gRPC to implement server and client communication and used both unary response RPCs and stream RPCs to build our system. We now describe our implementation in more detail.

System Implementation

Sensors

For our sensors, we implement two types: warning sensors placed on the platform, and alarm sensors placed inside the track tracks themselves. Each physical sensor is connected to a breadboard, an Arduino, and LED warning lights. We built three replicas of the alarm sensor clients and one replica of the warning sensor client. The warning clients are installed near-track, and they are installed with passive infrared sensors to detect close-by motion. The alarm clients are installed in-track, and they are paired with both the passive infrared and the ultrasonic sensors for increased accuracy and fault tolerance.

The heart of our operation are the Arduino boards. They are micro-controllers that we can use to program circuits and pass data to our main program. Each Arduino powers a client by connecting a branch of our sensor circuitry to the server. We use a combination of sensors to detect different types of incidences. The Ultrasonic Distance Sensor, model HC-SR04, detects how far away the nearest object in the line of detection is. The ultrasonic high frequency sound emitted by the trigger ping reflects off the object and is received by the echoing ping, and using the velocity of sound and the time it took the echoing ping to receive the triggered sound is used to detect the distance of the said object. The Trig and Echo pings can be connected to any spot on the Arduino slots (we chose slot 13 for Trig, 12 for Echo), so that the triggering and echoing of the sensor can be programmed in the Arduino stub. In addition to Trig and Echo, the GND ping connects to one of the GND slot on the Arduino, and VCC connects to 5V, which creates a circuit to power the ultrasonic sensor.

The Infrared Sensor captures motion by detecting temperature change from the emitted infrared radiation of nearby objects within its field of radius. It has three pings: VCC and GND to establish circuit, and an output ping to output booleans of whether it detects motion. We connected the output ping to Arduino slot 8.

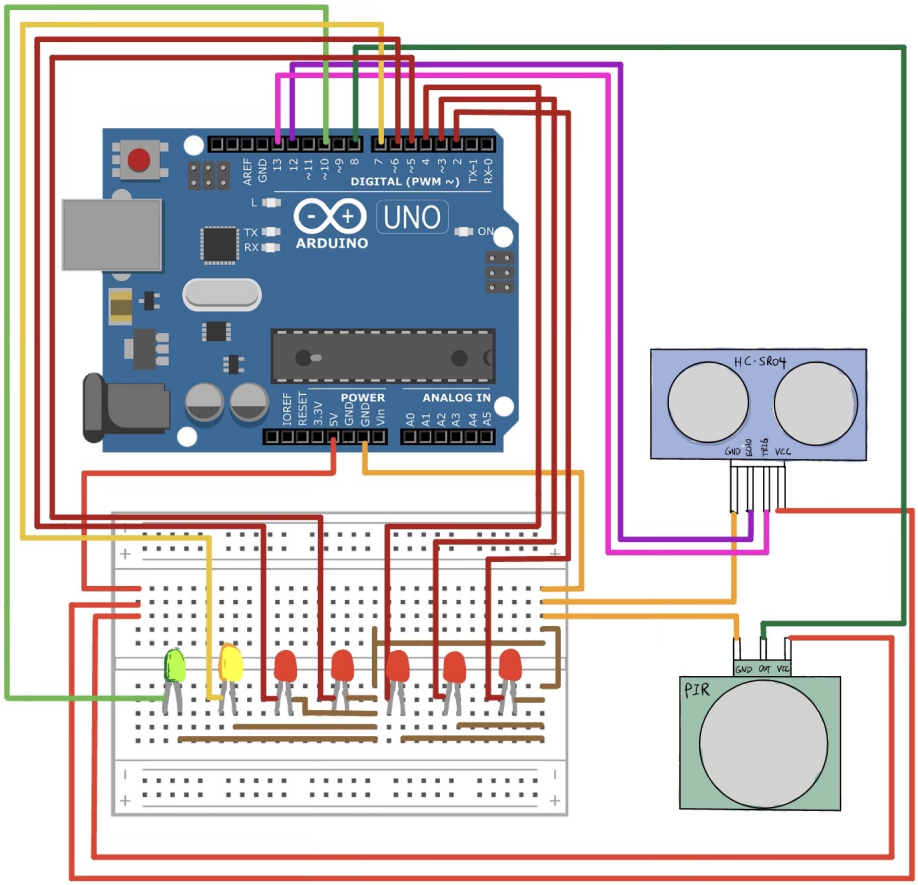
For warning near the train track, we use infrared rangefinders (PIR) and yellow LED signals installed parallel to the train track to detect proximity of passengers. These PIR sensors detect

abnormally close passenger motions when the train is not at the train stop, which notify the central server and activate yellow lights and yellow warnings to its distributed clients, immediately requesting the passenger to back off to safety via an audio file.

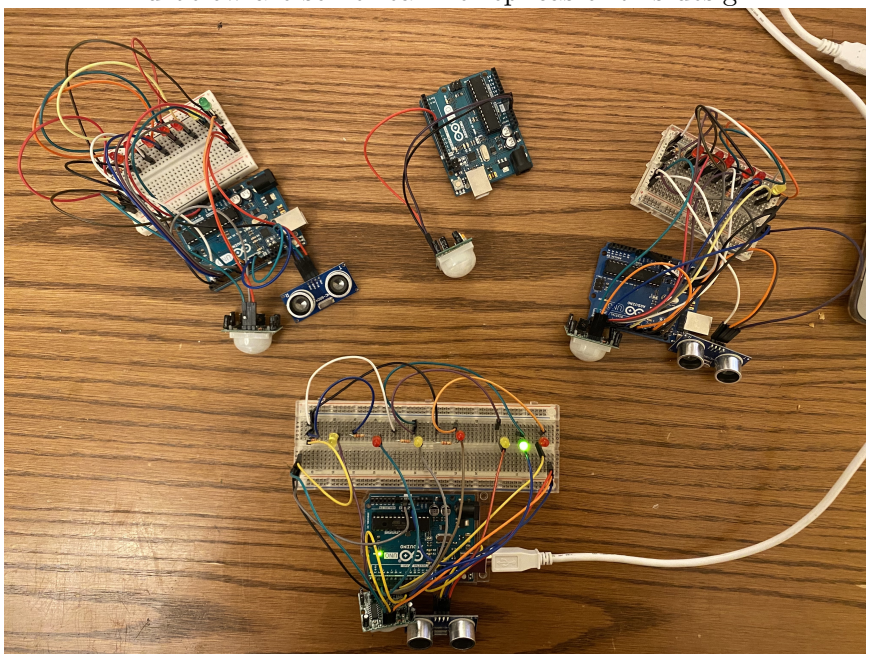
Alarm sensors are a bit more complicated. For alarm signals and protocol in the event of track intrusion, we use both infrared rangefinders and ultrasonic sensors in addition to green, red, and yellow LED lights installed vertically parallel to the wall of the track to detect potential track intrusions below the level of the subway tracks. If the alarm signal is activated, a signal is sent to the server, where a consensus algorithm is run to determine if the threat is real. If it is, the central server notifies the train API client so that the train conductor can enforce emergency stops immediately, before any possible collisions. The ultrasonic sensor would differentiate the difference between an arriving train or other moving objects via consecutive proximity sensing along the tracks, reducing the probability of false positive emergency triggers. We programmed our LED lights specifically to handle all three possible cases: if both the PIR and ultrasonic sensors go off, the yellow and red LED lights would light up in an incremental manner from left to right such that it appears as if the lights are moving from one LED to the next, if only the PIR goes off, all yellow and red LED lights blink simultaneously and continuously, and if no sensors are going off, the green LED is kept on.

Sensor Clients

Both the physical warning and alarm sensors are connected via USB to the computers running the sensor clients. Each sensor is methodically connected using jumper wires to the correct corresponding ports on each Arduino. For Arduino's with more than 1 sensor connection, we use a breadboard to effectively organize the device by creating parallel circuits of sensors and LEDs. Below is the design diagram we created for alarm client to demonstrate the correct wire flow and connections for our Arduino, sensors, and LEDs. It maps directly to the "PIR_and_ultrasonic.ino" arduino embedded stub code stored in ./arduino_sensors sub-repo.



And below are some real life replicas of this design:



In the Arduino IDE, we can easily view and read the data that our sensors give us, after uploading an Arduino embedded program into the Arduino itself (stored in the ./arduino_sensors sub-repo). In the case of the infrared rangefinder, the data comes in the form of a distance that we convert to centimeters, so we can expect to see a stream of data giving the closest distance to the path of the sensor. The motion sensor gives us whether there is or isn't motion. We consolidate this data into the form of a string with 0 or 1 for no movement and yes movement, followed by a separator "|" and then the distance value in centimeters. We then serialize this in the Arduino code and stream it to the port that the USB is connected to.

From here, the Python sensor clients are able to constantly read in the data from this port and de-serialize it. In both client files, we get that data but don't stream that data again to the servers to prevent unnecessary redundancy. Instead, we only update the server when there is meaningful information. For the warning sensors, we only update the server when there is someone crossing the line when there is no train, and for the alarm sensors, we only update the sensor when someone has fallen in given no train. This practice helps decrease the load on the system and abstracts the communication to what each participant needs to know and act upon.

Trains

Our trains client serves two main functions: first, to provide important information to our sensor system on the whereabouts of each train; second, to simulate trains on a track. We view the first function as something akin to a Train API. It assumes that each train is equipped with the ability to keep track of their own whereabouts within the subway system, and that each train is able to communicate its status to the central server, which manages all trains. As such, each train client calculates its own location based on speed and displacement (in real life, this would be replaced by another set of sensors installed on each train) and sends its own updated status to the server. The server keeps track of all statuses, which are then used to coordinate actions between trains.

Notably, train clients may request for the status of other clients' locations, and as a result, are able to instantiate safely. When we start up a new train client, the train will always wait until

it is safe to instantiate at the stop. It does so by following a minimum distance parameter; it queries for the status of other trains until it knows no other trains are within this minimum distance, then instantiates.

Another note on our API design: for simplicity, we assume a circular track with a singular stop on which on trains run. We also hardcode a maximum of three trains; we do not have a more complex algorithm to safely allocate a potentially unlimited number of trains on the track. Each train has a unique ID (akin to how the Red Line would want to know which train is coming through). We also set all train speeds to zero upon receiving an alarm; in other words, when it's been detected that someone has fallen onto the tracks, all trains immediately stop so that rescue personnel can address the crisis without further safety concerns.

Server

For our central server implementation, it facilitates the following business logic. While a train hasn't docked into the station, if the rangefinder is crossed, we simply sound the alarm as warning via an audio file, and if the motion sensor detects a person, we alert train conductor to stop the train (done on a case-by-case basis) and alert the station via an audio file and call 911. Now, while a train is docked we turn off rangefinders to allow flow of traffic in/out the train without false-positives and turn off motion sensors to negate false-positives as the train arrives. This is implemented by simply ignoring any calls while the train is physically present at the station.

The server is also replicated across two backup servers to ensure fault tolerance. Should one server go down, another backup server will, through a simple lowest-port election process, assume the role of the leader. This helps ensure that our system will be persistent through potential server outages.

Distributed System Problems Addressed

One of the first problems we faced was thinking about what information should be communicated between each of the computers at different levels. We decided to convert from Arduino data to outputting "stream" in Python client and then thought about the tradeoff

between the client sending a constant stream to the server compared to only sending a message when something is triggered as it relates to scalability and accuracy. At the first level, we have the communication between the physical sensors and the python sensor clients. Since this doesn't exactly impact the load on the network, we chose to stream the Arduino data completely to the client. At the next level though, we choose to reduce the toll on the network to only update the servers when there is something to act upon or know about. Not only does this help in reducing load, it also helps us better determine consensus among the clients, which we will talk about next.

In terms of consensus, we found that in a system like ours, the clients are actually more susceptible to failures than servers. Since the system is also designed such that there isn't much data in total being sent around, the likeliest ways it can experience failure is when clients go down, or more specifically, the physical sensors. Physical sensors are prone to often unpredictable errors, such as faulty wires, wind, tampering by other humans, and more. We even experienced a few of these in our controlled experimental testing. Since sensors have a higher chance of going down due to the various factors we discussed, we implemented a way for the server to get consensus among the clients so that it has a clear idea of what is happening in the physical world. As of now, we have implemented a simple majority consensus algorithm that compares the most recent statuses of the active sensors on the network. This is why we felt it was better to only update the server when it needs to know something - so that instead of the server having to interpret the data while also performing consensus, it can easily get the status of each of the sensors and simply compare their statuses.

For the train clients, we distribute the server communications by functionality so that there are separate threads for updating statuses with the scheduler API and the alarm system. The thread for the scheduler API helps ensure that a train will always been able to ping the serer with its current status; it also ensures that we can command any train movements, such as stopping, starting, or slowing down. The thread for the alarm system allows for communication with the sensor system. If there is a warning, the train conductors would know, in case the warnings are followed by subsequent alarms and crises. For alarms, the train conductors will immediately know to stop the train.

We also simulate this distributed train system by modelling various train clients on a circular track with one station. For a train to "enter" the track, we set its entry point on a station. Just like how a real-world train would wait until it is a safe distance away from all other trains before entering the track, each train client will continuously query all other active trains to see if any are within the minimum safe distances from the stop; only when all other trains are adequately far away will the train client instantiate on the track.

The sensor clients and train clients also communicate back and forth by through the server. By having each train client know its own status, the system is able to cross-reference alarms with whether or not a train is at the station. This prevents false positive situations–in other words, we make sure that the alarm will not go off when a train is at the stop.

Because such a system requires a large array of sensors, we have designed the sensors to be "plug and play." In other words, we can easily replicate the physical sensor designs and the sensor clients so that our system is easily scalable. While small adjustments may be implemented to create efficiencies in detection geometry, we have made the sensors well distributed.

Furthermore, we have implemented the servers to be replicative and two-fault tolerant. Should one server go down, one of the follower servers will be instated as the main server. Because the current server does not necessarily store vital information (all the train clients have their own statuses that will be continuously communicated via the RPC), this was a relatively simple process that does not require additional work in maintaining updated storage.

Although this feature has yet to be implemented, we should also consider implementing fault tolerance for the case that a train client fails or experiences delays. This problem is particularly salient for any public transportation commuter–after all, we have all come across the problem where the vehicle's reported location or speed is not fully accurate. We can predict the same thing may happen to the trains, especially since the trains are in the tunnels (and probably underfunded). As such, the server-side logic should not simply store the last known status of each train; instead, it should have an alternative method for calculating gaps in client updates when the pinging becomes noisy or irregular.

Things We've Learned

Byzantine failures occur more frequently than expected and are difficult to control based on what we've seen in our debugging. Especially due to the high likelihood of sensor inaccuracy and general problems when dealing with hardware, we saw that even with the same code across multiple devices, sometimes the distributed system didn't function properly. These problems were really difficult to deal with and required careful consideration of how to best build our system to mitigate the effects of byzantine failures.

Sensors are inherently distributed. Each sensor acts as a client, which communicates with the server in order for the server to understand the state of the system, then relaying it to the other sensor clients.

A system's logic should be mostly performed on the server-side with fault-tolerance. Again, sensors are inherently faulty and have a high individual chance to crash often. We saw that if each sensor was given the same consensus data and had the server communicate between every sensor, the system becomes not only hard to scale, but faulty when some sensors go down. Thus, most of the logic should be performed on the server side

Synchronization is difficult to tackle and hard to implement but is critical, especially for our project which works with train times and a stream of sensor data. Thus, if we design system internalizing delays that are inevitable, we can produce a better, more stable distributed system.

Further Steps

Looking ahead to how we can continue to work on the project, a lot of effort can still be made into turning out proof-of-concept implementation into one that can be physically placed into train stations.

For one, we would want to improve the scalability of the subway system. We want to make sure our system works when there are multiple stations and when tracks are not circular.

We can also add more scheduling API capability to dictate train movements such as

restarting trains after rescue has taken place and have the train movement physics correspond with local stops and slowdowns. This would involve adding physics into our simulation. At the moment, all trains immediately stop and start, without any calculation for acceleration. We could also implement more complex train-to-train mechanisms so that we would not have to stop all train at any given alarm; should the metro system become more complex and have multiple lines, our system should be able to localize stoppages as to maximize safety while minimizing commuter inconvenience.

Moreover, we can add geometric efficiency in sensor arrays with or without better sensors; namely, we can find a better placement of the sensors, especially alarm sensors, such that we minimize the number of sensors while maximizing the train track surface area covered for cost-efficient reasons.