Final Report

sdmay18-18

Fleet Monitoring System

Client: Lotfi Ben-Othmane

Group Members: Venecia Alvarez, Kendall Berner, Matthew Fuhrmann, William Fuhrmann, Anthony Guss, Tyler Hartsock

Introduction

Currently, it is difficult for managers of fleets to track the current location and status of their vehicles without using expensive software and hardware. The goal of our project is to provide a system for fleet managers to more easily monitor their fleet. This system will help make a fleet manager's job more time efficient and cost efficient. The system involves a device to be placed inside each individual vehicle in the fleet. The device pulls data from the vehicles that is then presented to the fleet manager on the website.

The device we used was a Raspberry Pi 3 running a Python application, interfacing with a PiCAN2 and an Adafruit Ultimate GPS. The Pi sends live data from the vehicle to the server, where the server processes and stores the data. This data is then displayed on the website for the manager's use. A map on the website shows the current location of each vehicle in the fleet. There are several charts and graphs that display a vehicle's current status as well as historical data, such as speed and engine temperature. The fleet managers can access the website quickly and easily to track their vehicles.

Related Projects

We researched several commercial fleet management applications to get an idea of what kind of features people would want, as well as to see where there are opportunities in the market. There are two main categories that we feel most fleet management applications fit into: applications that had the driver have a mobile application, and applications that had the vehicle have a device that is installed into the OBD-II port.

Most of the applications where the drivers have a mobile application sent the mobile device's' GPS and data manually entered by the drivers to a web application used by the fleet manager. The web application the fleet managers used generally had ways to interact with the drivers in

the fleet, such as messaging and dispatching. Examples of this type of application in our market study were collectiveFleet, ManagerPlus, and OnFleet. Generally, our application is covering a different niche than applications from this category because we will be providing internal data from the vehicle that these applications do not have access to, and we do not plan to provide functionality for the driver interacting with their fleet managers. Our product will provide useful interpretations of vehicle internal data for fleet vehicles instead of functionality for administration of a fleet.

We also noticed that a lot of things needed to be entered manually by drivers for these applications. We intend to improve on these systems by automating as much as possible. For example, several apps required the drivers to enter how much fuel they got at a gas station.

Most of the applications where the vehicles have an installed OBD-II device have a similar goal to our product, which is to provide live tracking, statistics, and useful interpretations of vehicle data to fleet managers. Examples of this type of application in our market study were FleetComplete, Fleetmatics Now, and High Point GPS. One issue with these solutions is that they are expensive, and are sold as a service. By using common hardware, instead of designing our own, we can allow more people to be able to use our solution.

See the complete list of fleet management software we reviewed in the references section.

Design Overview

Requirements

The requirements for our project were created through collaboration with our faculty client, and had several iterations where requirements were changed and updated based on the priorities of our client. To see more about alternative designs that were changed or not realized, see Appendix 2. Our client has pursued this project as a proof-of-concept, meaning that our deliverable should be a product that meets all of the requirements, but does not need to be ready for immediate release.

Functional Requirements

The product shall:

- Gather data from a vehicle's OBD-II (On-Board Diagnostics) port
- Transmit data from the vehicle to the server
- Process raw data from the vehicle on the server
- Record vehicle data into a database
- Display a map with a location of all vehicles in the fleet
- Display live data for a certain vehicle (speed, engine temperature)
- Allow managers to register vehicles that belong to a particular fleet

- Allow managers to remove vehicles from a particular fleet
- Allow managers to customize the data being displayed to them
- Display vehicle information only to users who have that vehicle registered to their fleet

Non-functional Requirements

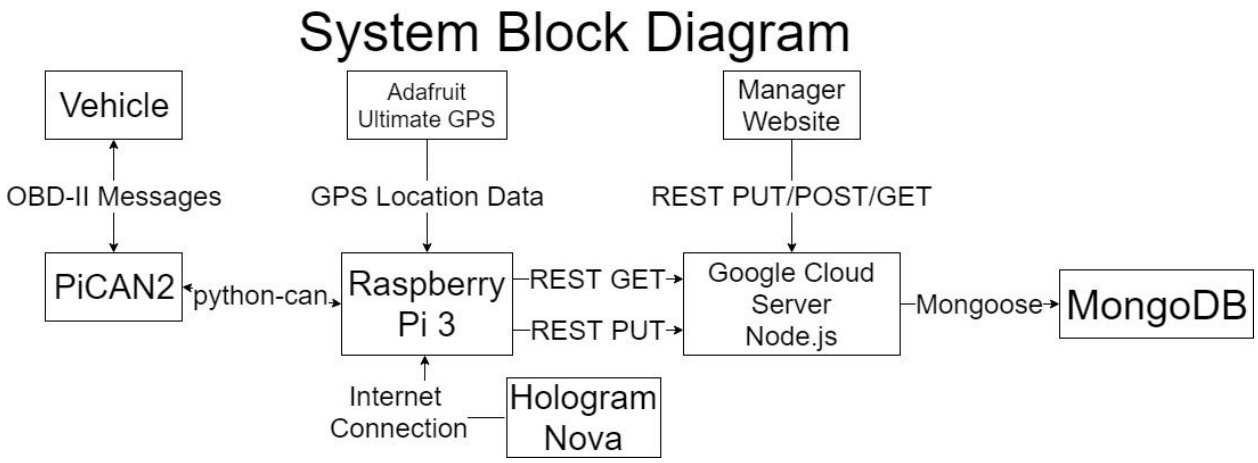
The product shall:

- Only allow managers to view fleet data on the website
- Only allow managers to view vehicles in their fleet

Constraints

- Have the server side code made in Node.js
- Utilize Google Cloud services

Component Overview



Our project will have three components. The first is the Python application for the Raspberry Pi that will get data from the OBD-II diagnostics port of a vehicle and forward it to the server for processing. The second will be a server that will take in data sent by the Raspberry Pi and store it, as well as provide necessary data to the website. The third is a website that will serve as a dashboard for fleet managers to be able to view specific data about their vehicles.

Raspberry Pi Overview

The Raspberry Pi collects the data needed and sends it to the server using a Python application. The Raspberry Pi uses a PiCAN2 device to interface with the OBD-II port, querying the OBD-II port for PIDs (Parameter IDs) that the server needs to calculate the live statistics. The Raspberry Pi gathers location data using an Adafruit Ultimate GPS. The data is sent to the server using a Hologram Nova for mobile connectivity. The Raspberry Pi polls configuration information, such as the PIDs to query and the bitrate of the vehicle's OBD-II port, from the server, allowing for front-end users to reconfigure the application while it is running.

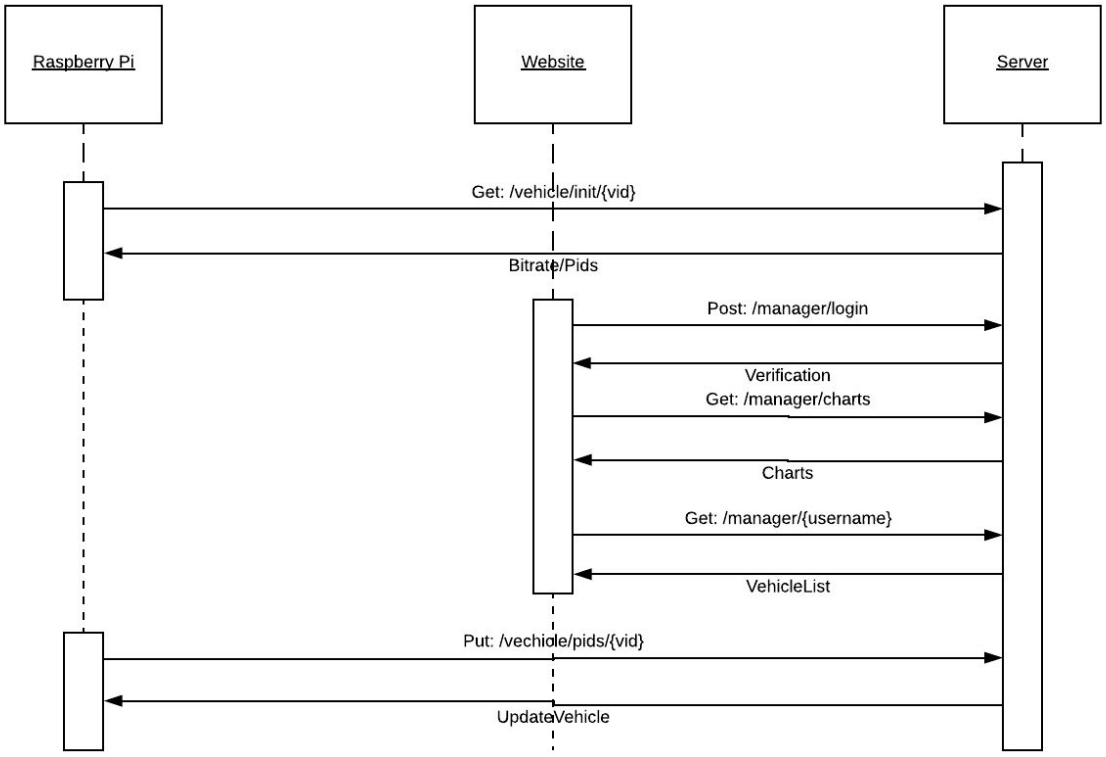
Front-End Overview

The webpage was made for use by fleet managers. It has a simple and intuitive design made for ease of use. The main web page displays a live map tracking the locations of all vehicles in the fleet in real time. Charts are displayed beneath the map, showing various statistics and information about the fleet as a whole. A single vehicle can be selected, causing the map to center and zoom in on that particular vehicle. While viewing an individual vehicle, the charts will change to show information about that vehicle. The types of charts available differ between the fleet-wide and individual levels. A separate page exists that allows the manager to select which charts they want to appear, so the manager's front-end is customizable to suit their particular needs. Another page exists to allow managers to add and remove vehicles from their fleet.

Server Overview

The server for our Fleet Monitoring system is implemented using Node.js. The server handles incoming and outgoing data through RESTful API calls. The server receives raw can data from the Raspberry Pi, and processes the data before putting it into a Mongo database. By using a REST API at the center of our server, it allows us to use the same protocol throughout our project. The API is implemented using GET, POST, PUT and DELETE API calls. This follows the suggested format for a RESTful service. In order to accurately document our API, we are using swagger which represents the API in a webpage. The server maintains a Mongo database through the use of Mongoose which allows us to define models. The models we have implemented are a Manager model and a Vehicle model. Vehicle are identified through a vehicle id and managers are identified by username and passwords. Passwords for managers are salted and hashed using bcrypt.js in order to securely store user information on our database. The server is hosted using a Google Cloud compute engine. By using Google Cloud we are able to scale our resources for our needs.

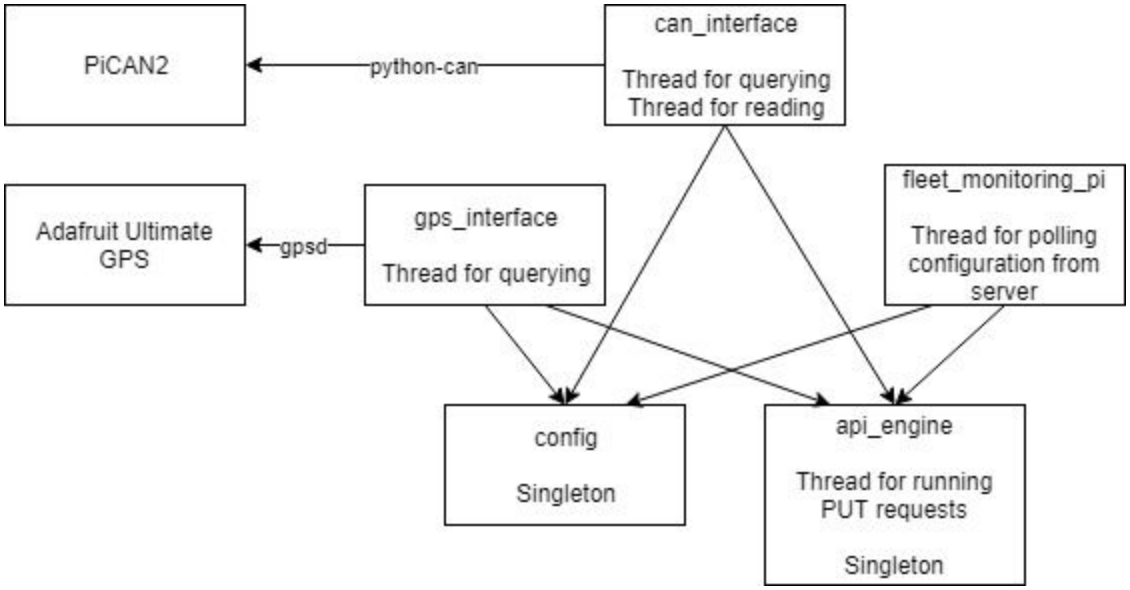
Server Interaction Diagram



Design Details

Raspberry Pi Design

The Raspberry Pi component consists of five software modules, two of which primarily interact with specific hardware elements and three which provide other functionality related to configuration, application management, and communication with the server.



can_interface

The can_interface module contains the interface to use the PiCAN2 with the python-can library to query and read messages from the OBD-II connector. It runs two threads: one that continuously queries the OBD-II network for the PIDs it is configured to query for, and one that reads the messages received on the connector. The way that the OBD-II port works, there are lots of messages on the network that are not needed to do the calculations for the front-end. Messages received are filtered to ensure that they are responses to a PID query and that the PID for the response matches one of the PIDs we are configured to use. The usage of this module usually consists of creating a CANHandler object, running startup() on it, and then running the thread using start(). This module uses the config module to get the configuration information it needs and the api_engine module to do the API calls to the server when it receives data. The interface has its own main method so that the hardware can be tested independently of the rest of the code.

gps_interface

The gps_interface module contains the interface to use the Adafruit Ultimate GPS, running using gpsd, with the python-gps module to pull reports the device gets from the GPS. It runs one thread that continuously pulls reports from the gpsd and filters them to only include reports that provide location data. The usage of this module usually consists of creating a GPSHandler object, running startup() on it, and then running the thread using start(). This module uses the api_engine module to do the API calls to the server when it receives data. The interface has its own main method so that the hardware can be tested independently of the rest of the code.

fleet_monitoring_pi

The fleet_monitoring_pi module runs the application. It is run with a set of arguments that allow it to connect to the server and know its own driver and vehicle ID. The application will startup all of the hardware components, populate the ConfigStore from the config module, and create and parameterize the API_Engine from the api_engine module, and will start all of the other threads in the application and begin polling the server for configuration updates. It contains a class FleetMonitorApp, which is used to run the application. It will use all other software modules so that it can start them up.

api_engine

The api_engine module runs the API calls to the server. This module contains the API_Engine singleton class, where all of the state of the device is accessed from and stored on a class variable. This allows the different classes to interact with an API_Engine without having to worry about whether the class has been configured yet. For the GET calls that are made in the fleet_monitoring_pi module, the requests are done synchronously because it is easy to run that call occasionally. For the PUT requests for the data, there is a thread that will only run once every 5 seconds with the most recent data. This is because the amount of data that comes from the OBD-II port is large, and the data usage on sending each part of PID or location data that comes in would be very large. The PUT requests also only send data that has been updated since the last API call to further save data usage. The usage of the api_engine module consists of creating the API_Engine class (which will be the singleton instance), and calling one of the API methods. The api_engine module depends on the config module to pull configuration information that goes into the URLs that requests are made on.

config

The config module holds the state needed to run the hardware and the api_engine modules. This module contains the ConfigStore singleton class, which allows the config updated in one module to update the configuration for other modules. The config module is loaded with default configuration, which is overwritten when the fleet_monitoring_pi application gets its first response and is overwritten by the arguments passed into the application. The reason for the defaults is mostly because when it is not run through the main application, it can still run the hardware using default values such as the standard 500kHz bitrate for the OBD-II port. The config module has no dependencies. The usage of the config module consists of creating the ConfigStore class (which will be the singleton instance), and calling its methods, which will set or access the values that are common for all modules in the application.

Technologies Used

Hardware:

- Raspberry Pi 3 - Runs application
- PiCAN2 - OBD-II interface
- Adafruit Ultimate GPS - GPS data
- Hologram Nova - mobile connectivity

Software:

- python-can - Creates messages for OBD-II, interfaces with PiCAN2
- python-gps - Gets data recorded from gpsd
- requests - Runs API calls
- gpsd, gpsd-clients - Gets data from Adafruit Ultimate GPS
- Python - Application language
- Bash - Startup script language

Front-End Design

AngularJS

The front-end website functions primarily as a single page application written in AngularJS. AngularJS allows us to easily update a page in real time and display customized results based on each manager's preferences.

Fleetmap.php

The main page of the application. This is where managers will land upon logging in. This is where they will see the map displaying the locations of each of their vehicles, as well as the charts they have chosen to appear.

editView.php

This page has two sets of checkboxes, one for fleet-wide statistics, and one for individual vehicle statistics. If a box is checked, then it's corresponding chart will appear on the main page, otherwise it will be absent.

editFleet.php

This page is for adding and removing vehicles from the fleet.

fleetMap.js

This page contains the logic to initialize the application and start up the other services to display the bulk of the main page. It is responsible for calling the server to obtain vehicle data that it stores in a scope variable so the other pages can use it without having to make their own server calls.

editView.js

This page is responsible for keeping track of what charts the manager wants displayed. It has lists of all possible charts that it's corresponding php page uses to construct it's checkboxes. Whenever a box is checked or unchecked, the list of chosen charts is updated to include or exclude that chart. Ideally, these selection would be stored and fetched server side, but we ran into issues so for the time being we're using local storage to remember what charts the manager has selected in between sessions.

chartService.js

The primary duty of this page is to create the charts to be displayed on the main page. It has two main functions. One to create the fleet charts, and one to create the individual charts. When either is called, they cycle through their respective lists of selected charts and call the functions necessary to create them. Whenever the charts need to be updated, the old ones are destroyed to make room for the new ones. The charts are created using Chart.JS.

mapService.js

The map service is used to create and update the map on the main page. The map utilizes the Google Maps API. The service initializes the map and vehicle markers when the page is first loaded. The main page calls a function from the service on an interval to check for any updates in a vehicle from the fleet's location. If there is a new location, the marker will move on the map.

Technologies Used

- AngularJS: Key to the primarily single page design. Allows the main page to be heavily customizable and easy to update on the fly.
- Bootstrap: Standard helper to make websites look better with minimal effort.
- Jquery: Used for server calls.

Server Design

Modules

The node.js server utilizes various node modules. For database and model management we use a combination of MongoDB and Mongoose. We use the body-parser module to parse the bodies of api calls as well as to manage json objects. For our project we also needed login and register functionality, and to ensure the security of user's passwords, we used bcrypt.js to encrypt passwords before we put them into the database.

server.js

The server.js file is the runnable server file. The main purpose is for it to set up the service and allow it to receive and send API requests. This file will make the connection to the MongoDB to be used, by using Mongoose to connect securely. The server will then initialize the routes from routes.js and allow API requests to be routed. The server is currently setup to run with a test database and on port 8080. This file is where most of the configuration changes will need to be made.

routes.js

The routes.js file provides a series of endpoints to be used by the API. The routes are structure by the url path to the call, and the type of call such as GET, PUT, POST and DELETE. The routes are responsible for sending the API request to their functions in their respective controllers.

vehicleModel.js

The vehicleModel contains the structure for the vehicle's Mongoose model. Each vehicle object has defined elements for each required type. The vehicle model also uses the data model, which allows the vehicle models to contain an array of data elements that can be easily accessed. A large change to this version of our software is the the vehicle model now contains a list of pids that should be read from the Raspberry Pi.

managerModel.js

The managerModel contains the structure for the manager's Mongoose model. The manager model contains a field for username, a hashed password, a name and an array of vehicle ids that the manager wants to monitor.

vehicleController.js

The vehicleController contains the functions that are routed to when an API call goes through the routes.js file. The vehicle controller contains various functions that are necessary for operation. An important use of this part of the project that is new in this version is the initiation. The vehicle controller has a new function init that will send data necessary to the Raspberry Pi at the Raspberry Pi start time. New in this version of the project is that data processing of the pids is now done by the server in the function updateVehiclePid. This change was done to make the project much more extensible.

managerController.js

The managerController contains function that are routed to by api calls on the /managers/* endpoints. The most notable of these is the login function which uses BCrypt to salt and hash the user's password. Managers are also able to add and remove vehicles from their fleet though these functions.

Technologies Used

- Node.js
- Express, BodyParser
- MongoDB, Mongoose: Used for Database operations and models
- BCrypt.js: Used to securely store passwords in the database

API Documentation

Full Swagger documentation of our API is available at https://sdmay18-18.sd.ece.iastate.edu/swagger-ui-master/swagger-ui-master/dist/index.html.

Standards

One standard that was important to use for our interpretation of the vehicle data is SAE J/1979, which defines the standard mappings from PIDs to vehicle data. We had to use this standard when finding what information could be pulled from the vehicle, and how to interpret data that comes from PIDs and convert it to usable statistics.

Testing

Testing Methods

For testing the hardware interfaces on the Raspberry Pi, the modules interacting with the hardware are able to run independently. This allows for isolated testing of the hardware for both GPS and OBD-II interaction. The GPS can be tested simply, as verifying that the GPS coordinates are correct is trivial with other tools (such as Google Maps) that can be used to show that your current location is the same as the GPS coordinates you are receiving. The OBD-II interface can be tested using an OBD-II simulator provided by our client, as well as using an actual vehicle. One difficulty for testing is that the PiCAN2 only works with vehicles that use the CAN for their OBD-II, which was not standard until 2008. Older vehicles we tested with did not work with the PiCAN2, so we had to rent a newer vehicle to perform our testing, making testing the can_interface module a much less frequent event than it could have been. The api_engine module was tested to ensure that all of the API calls we were making were valid and that the return values were what was expected after being called through the requests library.

In order to test our server, we use Postman to verify our API and perform regression testing. Postman allows us to setup automated tests that will run whenever we update the API for the server. This is done by creating Postman “runners” that will run through a series of API calls and verify the results of these calls. This allows us to verify that changes we make are no having unintended side effects. These tests will allow us to easily verify our API implementation. We also verify the data produced by our data operations.

Front-end code is always first written and tested locally, using UwAmp and XAmpp to simulate the code being run on a proper server. Testing is performed manually, most often by the developer. Testing is very frequent, with the page being reloaded with every small change to the codebase to make sure everything still works properly. The developer console on the browser is also useful for client side testing and debugging. Logging helpful messages to the console helps to confirm the desired functionality is working. When deploying new changes to our hosted website, manual regression testing is done on the interface to ensure that everything deployed correctly and successfully and that all other parts of the application still work as expected. Postman was also used in the testing of the application to confirm that updates in the database showed up live on the website.

Testing Results

The Raspberry Pi application was tested using a rented U-Haul van with our software running on it, and was successfully able to query and gather OBD-II PID information from the vehicle, obtain live GPS data, and send the data to the server. It was also able to do this for the OBD-II

simulator we were provided, and the server was checked to ensure that the data sent was properly formatted and ingested correctly into the database.

The server was tested using Postman. This allowed us to validate that the API calls were working correctly. We used Postman to setup automatic tests to ensure that the API calls remain correct. This is done by calling a specific API call and validating the response from the server. These tests will grow more complex as our models and data grow as we implement new features. Our API documentation will be created using Swagger. Swagger allows us to create documentation with an easily readable UI. Swagger will also allow people to test calls directly from the UI.

End to end testing was done on the website to confirm that the application flow worked correctly. We also used Postman to update the database, and then confirmed that the website showed this most up-to-date information live. Lastly, we monitored the website while the Pi was running in the U-Haul van to confirm that the data was displaying as expected.

Appendix 1 - Operation Manual

Server Deployment

For the initial setup of the server, you will need to decide whether you are using a local database or a hosted database. If using a local database you must change the variable “server” in server.js to false. You will also need to ensure that MongoDB is running before running the server. After this while in the directory of the server run “npm install” in order to install the required node modules. Once this is done in the directory of the server run “node server.js” to start the server.
If the server is being run on a dedicated server it is recommended to create a service to easily control running and stopping the server.

Raspberry Pi Installation

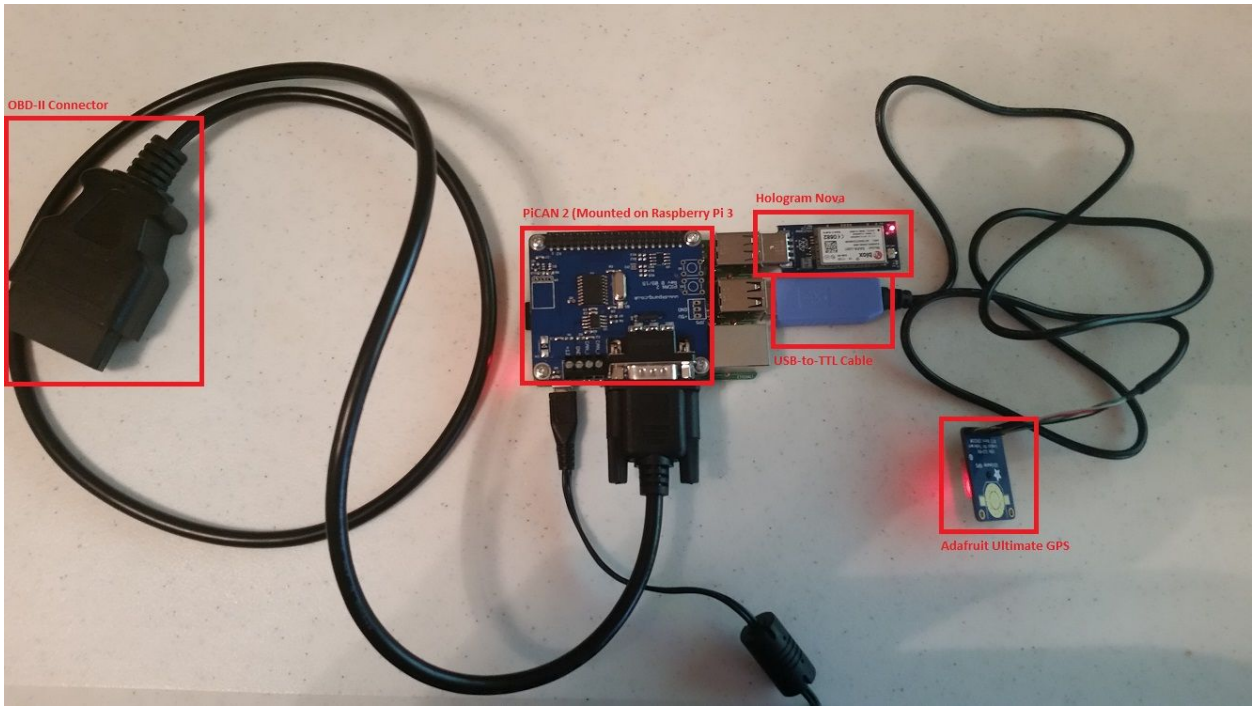
For the initial setup of the Raspberry Pi, there are several hardware and software components needed to integrate and use the components.

Physical Components:

- Raspberry Pi 3
- Adafruit Ultimate GPS and USB-to-TTL Cable
- Hologram Nova (or other mobile connectivity device)
- PiCAN2 and OBD-II to serial cable.

To put together the physical device, you will need to run through several steps:

1. Install the PiCAN2 onto the Raspberry Pi 3: see the “Hardware Installation” section at the PiCAN2 User Guide at http://copperhilltech.com/pican2-controller-area-network-can-interface-for-raspberry-pi/. The solder bridge should be set for the OBD-II connector configuration. This only needs to performed once.
2. If using the Hologram Nova: see the “Hologram Global IoT SIM Card” and “Quad-band Flexible UFL Antennas” sections at the Hologram Nova User Guide at https://hologram.io/docs/reference/nova/user-manual/#hologram-global-iot-sim-card. This only needs to be performed once.
3. The Hologram Nova should have an antenna attached to it on the uFL port and must be connected to the USB ports on the Raspberry Pi before running the application.
4. The USB-to-TTL cable must be connected to the USB ports on the Raspberry Pi and with the TTL connectors on the pins on the Adafruit Ultimate GPS as described at https://learn.adafruit.com/adafruit-ultimate-gps-on-the-raspberry-pi/setting-everything-up before running the application.
5. The PiCAN2 should be connected to the OBD-II port of the vehicle using the OBD-II to serial cable before running the application.
6. The device needs power, which it takes in through a MiniUSB port, before running the application.



To install the software prior to setup, you will need to perform the following steps:

1. Install gpsd, python-gps, and gpsd-clients: see the “Install a GPS Daemon” section at https://learn.adafruit.com/adafruit-ultimate-gps-on-the-raspberry-pi/setting-everything-up.

2. Install python-can, setup for PiCAN2: see the “Software Installation” and “Python Applications” section at http://copperhilltech.com/pican2-controller-area-network-can-interface-for-raspberry-pi/.
3. Install requests: see http://docs.python-requests.org/en/master/user/install/.
4. Install the application
    a. Download the application from https://github.com/lbenothmane/FleetManagement.
    b. Modify the file at rasp_pi_tests/startup/startFleetMon:
        i. Set the SERVER_URI variable to be the URI of the application server.
        ii. Set the VID variable to be the ID of a vehicle created through the front-end.
        iii. Set the DRIVER_FILE to be the location of a file where the driver ID is set. (This can be the location of a file in a flash drive if you want to have each driver put one in with their own ID, or can be any file where you have just a number stored.)
    c. Copy the file at rasp_pi_tests/startup/startFleetMon to /etc/init.d
        i. $ sudo cp .../rasp_pi_tests/startup/startFleetMon /etc/init.d
    d. Set the machine to run the script automatically on startup:
        i. $ sudo update-rc.d startFleetMon defaults
        ii. This allows the application to startup on boot: restart the Raspberry Pi to run the application.

Website Set-Up

To view the website locally, follow these steps.

1. Install UwAmp if you’re running Windows, or XAmpp for Mac.
2. Navigate to the www folder, probably located here: ‘C:\Program Files\UwAmp\www\’.
3. Clone our Git Repository into this folder.
4. Run UwAmp, then click the button labeled ‘Browser www’.
    a. Alternatively, manually navigate to localhost:8080 in your browser
5. Navigate to our client side, located here: FleetManagement/www/client_side/fleetmap.php

Any code edited will appear in this local browser now.

Appendix 2 - Alternative Designs

One alternative design that we considered early on was to use Java Spring Microservices for our server. This would have allowed us to use a better and newer technology to implement our server. We would have also been able to take advantage of the benefits of microservices such

as the ability to easily setup continuous integration continuous deployment using a combination of Jenkins, Docker and Kubernetes. Microservices work particularly well for having multiple processes that can be independently run, which would have worked well for separating data ingestion and website API requests into separate microservices. Another benefit of this approach is that we would have been able to take advantage of more processing power that could be scaled directly to our needs. Node.js by comparison does not have as much processing power due to the limitations of the language. However, one of the constraints put on us by our client was to implement the server using node.js.

The plan was originally to use an Android microcontroller device that was supposed to have built-in hardware and software support for communicating with the vehicle over the CAN (Controlled Area Network) bus. The reason we wanted to use the device is because it had a screen available, meaning that we would be able to provide driver functionality in the same device (drivers could use Google Maps for navigation, we could provide alerts for fleet managers and drivers). However, the device was not properly configured, and the support provided was not sufficient to allow the continued development of the project. The company was very slow to respond for requests to provide information or missing parts, and the documentation was not helpful in solving the issues. After discussion with our client in October, we decided that using a device with poor support was not a good idea, so we decided to pursue other options. In November, we began using the Raspberry Pi instead, which has much more general support and information available than our previous hardware.

A stretch goal of the project that did not get implemented was to have server calculations done with R. We researched R and how to implement R so that it uses a MongoDB database. We developed R code to calculate total time that a vehicle has been idle, and the percentage of time a vehicle has been idle. We were able to do analysis on sample data from MongoDB, but we completed the prototype too late to integrate the functionality into our system.

We also considered using the Google Roads API to plot paths that the vehicles had taken given a certain time interval. However, selecting the date/time interval and then filtering the vehicle's history for the relevant data proved to be a lengthier task than we had time for. This feature also didn't seem to have as much demand as other features we implemented. The Google Roads API also could have been used to gather data about the speed limits on the roads that the fleet vehicles were traveling on. We could have made an alert pop up on the website to alert the manager if a vehicle was driving faster than the speed limit. It turned out that the speed limit feature of the API wasn't free to use, so we decided against using it.

Resources

MongoDB. [Online]. Available https://www.mongodb.com/. [Accessed: 15-Sep-2017]
Swagger. [Online]. Available https://swagger.io/. [Accessed 28-Nov-2017]

Google Cloud Platform Documentation. [Online]. Available https://cloud.google.com/docs/. [Accessed 10-Sep-2017]
Introduction to AngularJS. [Online]. https://www.w3schools.com/angular/angular_intro.asp. [Accessed: 2-Oct-2017].
AngularJS API [Online]. https://docs.angularjs.org/api. [Accessed 2-Oct-2017].
Chart.js. [Online]. http://www.chartjs.org/. [Accessed: 30-Sept-2017].
Google Maps API. [Online]. https://developers.google.com/maps/documentation/javascript/tutorial. [Accessed: 30-Sept-2017].
High Point GPS. [Online]. Available: http://www.highpointgps.com/. [Accessed: 10-Sep-2017].
Fleet Complete Fleet Tracker. [Online]. Available: https://www.fleetcomplete.com/en/products/fleet-tracker/. [Accessed: 10-Sep-2017].
Fleetmatics Now. [Online]. Available: https://www.fleetmatics.com/now. [Accessed: 10-Sep-2017].
collectiveFleet. [Online]. Available: http://www.collectivedata.com/fleet_management_software.html. [Accessed: 11-Sep-2017].
ManagerPlus. [Online]. Available: http://www.managerplus.com/lps/managerplus. [Accessed: 11-Sep-2017].
Rhino Fleet Tracking. [Online]. Available: http://www.rhinofleettracking.com/. [Accessed: 11-Sep-2017].
Linxup. [Online]. Available: http://www.linxup.com/. [Accessed: 11-Sep-2017].
Silent Passenger. [Online]. Available: https://vehicletracking.com/features/. [Accessed: 11-Sep-2017].
Azuga Fleet. [Online]. Available: https://www.azuga.com/. [Accessed: 11-Sep-2017].
Onfleet. [Online]. Available: https://onfleet.com/. [Accessed: 11-Sep-2017].
Fletilla FleetFACTZ. [Online]. Available: https://fleetilla.com/products/fleet-management-solutions/real-time-tracking. [Accessed: 11-Sep-2017].
Fleetio. [Online]. Available: https://www.fleetio.com/. [Accessed: 11-Sep-2017].
MongoDB. [Online]. Available https://www.mongodb.com/. [Accessed: 15-Sep-2017]
Google Cloud Platform Documentation. [Online]. Available https://cloud.google.com/docs/. [Accessed 10-Sep-2017]
PiCan2 User Guide. [Online]. Available http://copperhilltech.com/pican2-controller-area-network-can-interface-for-raspberry-pi/. [Accessed 15-Nov-2017].
Adafruit Ultimate GPS Documentation. [Online]. Available https://www.adafruit.com/product/746. [Accessed 17-Nov-2017].
Hologram Nova Documentation. [Online]. Available https://hologram.io/nova/. [Accessed 23-Mar-2018].