A Novel System To Improve Over-Speeding Traffic Violation Ticketing using a Black Box

Gaurav Mohan Shende
gms1682@rit.edu

A Novel System To Improve Over-Speeding Traffic Violation Ticketing using a Black Box

by
Gaurav Mohan Shende

GRADUATE PAPER

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

Approved by:

____

Mr. Mark A. Indovina, Senior Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

____

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
DECEMBER 2019

I dedicate this work to my Mom, Dad and my close friends Shadaab and Siddhi for their constant love, support and encouragement throughout my academic years at Rochester Institute of Technology.

Abstract

The paper will discuss the implementation of a novel system using a black box to make the overspeeding ticketing system more robust. The system will provide a physical proof of the drivers driving statistics. The system consist of three parts. First being a black box implementation using a raspberry pi 3b+ on the drivers side. It will track the current speed of the car via OBD-ii port with timestamp. Second being IOT device on the sheriff's side which can fetch the speed information for past 20 minutes (speed information before vehicle came to compelete stantstill) from the black box and push it to the cloud. And the third part is cloud infrastructure which will receive the information from all the IoT devices and store in a S3 bucket. The sheriff can see the visualization of this data w.r.t. time on IoT device. The system will help sheriff make much informed decision as they will have past driving statistics of the driver. This will also help eliminate the corner cases such as over-speeding ticket when changing lanes or overtaking a vehicle.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This paper is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Gaurav Mohan Shende

December 2019

Acknowledgements

I would like to thank my advisor, professor, and mentor, Mark A. Indovina, for all of his guidance and feedback throughout the entirety of this project. He is the reason for my love of embedded system design and drove me to pursue it as a career path. He has been a tremendous help and a true friend during my graduate career at RIT. I would like to thank my friends Ajinkya, Akshay and Goral for giving me true friendship, endless laughs, and great company throughout the many, many long nights spent in the library and labs. Finally, This project would not have been possible without their love, advice, and companionship throughout my entire career at RIT.

Contents

Abstract **ii**

Declaration **iii**

Acknowledgements **iv**

Contents **v**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**
- 1.1 Organization 3

2 Bibliographic Research **4**

3 System Overview **8**
- 3.1 Black Box 10
  - 3.1.1 CAN Shield 11
  - 3.1.2 Raspberry Pi 3b+ 13
  - 3.1.3 Interface between CAN Shield and Raspberry Pi 3b+ 13
- 3.2 IoT Device 14
  - 3.2.1 Raspberry Pi 15
  - 3.2.2 AWS Infrastructure 15
  - 3.2.3 Data Representation 17
- 3.3 Software Flow 18
  - 3.3.1 Software Flow for Black Box Side 18
  - 3.3.2 Software Flow for IoT Device Side 25
- 3.4 Key Consideration for the System Design and Components 31
- 3.5 Cost Break-Down for the Project 33

4 Protocols and Standards **35**
4.1 OBD-ii Standard 35
4.2 CAN 37
4.3 Bluetooth 38
4.4 MQTT 41

5 Tests and Results **43**
5.1 Black Box Script Testing 43
5.1.1 CAN shield and Raspberry Pi 3b+ Interface Testing 43
5.1.2 Black Box and Car Interface Testing 45
5.1.3 Bluetooth Module Testing 47
5.2 IoT Device Script Testing 48
5.2.1 AWS IoT Core Interface 49
5.2.2 AWS S3 Core 50
5.3 Integration Testing 51

6 Conclusions **52**
6.1 Project Conclusions 52
6.2 Future Work 53

References **55**

I Guides **I-1**
I.1 Bluetooth Pairing I-1
I.2 Bluetooth Serial Port Profile I-3
I.3 AWS IoT Core I-5
I.4 AWS S3 Bucket I-10

II Source Code **II-1**
II.1 Python Script Related to Black Box II-1
II.1.1 Black Box Script II-1
II.1.2 Log Data Generator Script II-6
II.2 Python Script Related to IoT Device II-7
II.2.1 IoT Device Script II-7
II.2.2 Data Plotting Script II-12
II.2.3 Mock JSON files Generator Script II-17

List of Figures

2.1 View from the Lidar/Radar gun 6

3.1 Architecture 8
3.2 Black Box Side Architecture 10
3.3 CAN Shield 11
3.4 Configuration for SPI Interface 14
3.5 IoT Side Architecture 14
3.6 Rule created for the IoT thing 16
3.7 Final Speed Plot 17
3.8 Software flowchart for Black Box Part-I 20
3.9 Software flowchart for Black Box Part-II 21
3.10 Bluetooth Transmit Function 24
3.11 IoT Device Main Function 26
3.12 AWS IoT Core Software Configuration 28
3.13 AWS S3 Bucket 30

4.1 OBD-ii Port and Pin description 35
4.2 Format for PID request 36
4.3 CAN network 38
4.4 CAN frame format 39
4.5 Bluetooth data exchange 40
4.6 MQTT network 41

6.1 GPS View 54

I.1 Create a thing I-5
I.2 Naming a thing I-6
I.3 Create certificate I-7
I.4 Create certificate I-7
I.5 End point address I-9
I.6 Monitor on dashboard I-10
I.7 Action creation I-11

I.8 S3 bucket creation I-12

List of Tables

3.1 Overall Final System Cost 33
3.2 Cost to driver 33
3.3 Cost per Sheriff 34
3.4 Overall Development Cost 34

Chapter 1

Introduction

’Every once in a while, a new technology, an old problem and a big idea turn in to innovation.’

- Dean Kamen

National Highway Traffic and Safety Administration or more commonly known as NHTSA is a federal agency of the US state government. The agency did a survey in 2018 to estimate the speed for all types of motor vehicles on freeways, arterial highways, and collector roads across the United States. At 677 sites speed of close to 12 million vehicles were sampled across the united states. The survey [1] mentions that "Overall, speeds of free-flow traffic on freeways averaged 70.4 mph and were approximately 14 mph higher than on major arterials, which at 56.4 mph were in turn about 7 mph higher than the mean speed of 49.7 mph on minor arterials and collector roads. Most traffic exceeded the speed limits. Sixty-eight percent of the traffic on limited-access roads, about 56 percent of the traffic on arterials, and about 58 percent of the traffic on collectors exceeded the speed limit. About 16- to 19 percent of traffic exceeded the speed limit by 10 mph or more on freeways, arterials, and collector roads. While speeds of most vehicle size classes remained constant since 2009, the longest truck class (80-100 ft.) showed a 2 mph increase on limited-access highways." The results were almost the same in the survey done in 2015 [2].

This survey helps to understand the driving trend on the roads. In most cases, the vehicles traveling on the highways or the minor arterials travel at a considerably higher speed than the assigned limit. A thing to observe here is that drivers try to maintain the speed the vehicle with the flow of traffic. Going slow or too fast may result in a ticket. So, as the speed of the traffic flow itself is high, anyone trying to maintain that traffic speed is bound to be above the speed limit. Thus, it is not fair to the ticket only a few drivers for over-speeding. This also means that the over-speeding ticket citation is more of a relative process than absolute. There are cases where the drivers are let go with a warning or sometimes given a ticket for going a mere of 1 mph above the speed limit. The idea of this project is to bring in the robustness to the decision making the process. This is done by giving the sheriff a tool to identify the type of driver. If the sheriff has proof of speed statistics over a certain amount of period about the driver in question, he/she can identify whether the driver is aggressive or a good driver and then take the decision on the citation. This data will also be available in the judicial system, in case the driver decides to fight the ticket in the court. This will also provide as a second check before anyone being given an over-speeding citation.

In this project, a system is developed to address and fix these issues using the currently available technology. In this project, a raspberry pi 3b+ is used to implement a black box. A black box will always be present in the car and will record the actual speed of the vehicle along with the time stamp via OBD-ii port. The speed is sampled at one sample per second. Another raspberry pi 3b is used to implement an IoT device. This IoT device will be present with the sheriff. The IoT device will request the speed reading from the car for the past 20 minutes (1200 data points) over the Bluetooth. This data is then pushed to the secure cloud and can be accessed on a website or by a tool in real-time for the particular vehicle in focus. In this project, a tool is implemented to fetch the data from a cloud and plot it on a graph with clear, distinct speed limit of the car. The data fetched from the car is used to verify the speed of the vehicle in question. Thus, reducing the human error of misidentifying vehicle. Such a mechanism will help sheriff make a much-informed decision as he/she will have drivers speed statistics over a period of time, helping them to identify if the driver is aggressive or not and then give them a ticket.

One more advantage of this system is that as the data will be present in the cloud and anyone in the judiciary system can access it. So, in case the driver is given an over-speeding ticket and decides to fight it. He/she can have proof to present in the court. Hence, making the process more robust. In this system, the decision is not solely dependent on judgment on a single person but multiple cross-verification supported by proof.

1.1 Organization

- Chapter 2 discusses the motivation for the project, discussion about the current speeding violation rules and research, and systems available to tackle the problem so far.
- Chapter 3 discusses overall system architecture in detail. It will cover various components used, why they were used? Cost of the system etc.
- Chapter 4 discuss various protocols used such as CAN, BLE, MQTT. It will cover the brief description and mode in which they are used and a few important specifications.
- Chapter 5 discusses the various method in which all the scripts were tested individually. The difficulty faced during the project integration and the comparison between the actual and expected output.
- Chapter 6 discusses the future work and conclusion of the project.

Chapter 2

Bibliographic Research

The over-speeding ticket is one of the most common traffic citations given in the United States. According to the statistics, every year, close to 34 million speeding tickets are issued. The United States has close to 227 million [3] licensed drivers. This means that every year, one in every six drivers receives an over-speeding ticket. These numbers are on a higher side. So, does the United States have so many drivers who willingly Overspeed? The answer is no. Let's see the bigger picture. Any traffic citation (in this case, over-speeding ticket) can result in a fine and increase in driver's insurance rate. Thus, anyone breaking the traffic law knowing the consequences is highly unlikely. The problem here is the way over-speeding vehicles are identified. Currently, a handheld RADAR or a LIDAR gun (most commonly known as speed gun) is used to measure the speed of the car [4]. The gun is pointed at the target vehicle number plate to measure its speed. Using speed gun has few disadvantages:

- The gun has to be held with steady hands.
- There are chances that the sheriff may issue a ticket to the wrong vehicle.
- RADAR reading is affected by weather conditions and the color of the car/number plate.

Moreover, the major drawback of the current Overspeed ticketing system is that the driver does not have proof to counter the sheriff's claim, in case he/she is innocent. Thus, making him/her vulnerable if they decide to fight the ticket in the court.

In the past year, three over-speeding tickets were given to someone or the other I know. Although sure that they were not over-speeding, two of them decided to pay the fine in upwards of $240 reluctantly. They decided to do pay the fine because the area where the citation was given to them was around one to one and a half hours away from the place they stay. Fighting the ticket meant them to travel back to that particular county court where the citation was given. If the sheriff who gave them the citation is not present, then it meant they had to travel back again for the next trial. Thus, in order to avoid the hassle, they ended up paying up for the tickets. The third person decided to fight the ticket. The person traveled to county court thrice, and the sheriff who gave them the citation never showed up in the court once. This resulted in the judge let go of the ticket. As it can be observed from these incident fighting the over speed ticket is a tedious process and also even if the driver is sure that he/she was not over speeding they end up paying the fine in order to save time and lack of proof. After discussing the issue with them, I realized the gaps in the current system and tried to fill them using this project. But before going in deep with the issue first, we will discuss current over-speeding vehicle identification mechanisms and loopholes.

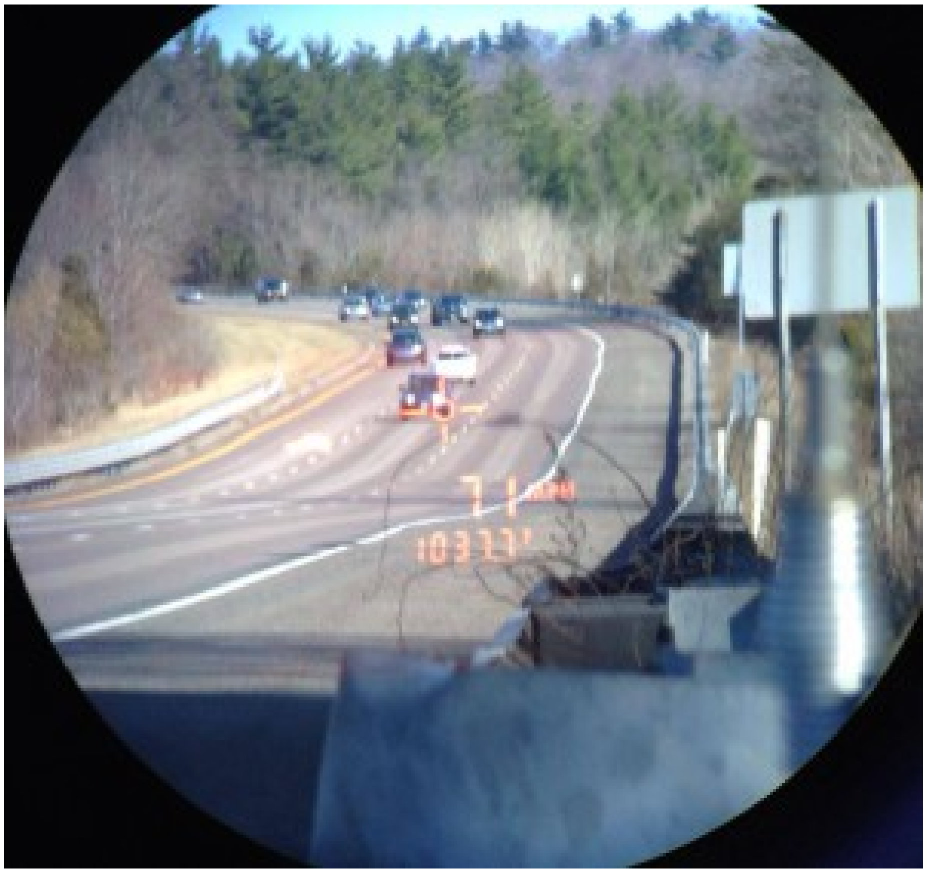


Figure 2.1: View from the Lidar/Radar gun

A LIDAR or RADAR guns are used by the sheriff to identify the over-speed vehicle. The gun can be handheld or car-mounted. RADAR guns use the Doppler effect [4] to identify the over-speeding vehicle. The gun is pointed at the vehicle in question, and it will transmit a radio wave. This radio wave will bounce back from the vehicle and is received by the gun [5]. Due to the Doppler effect, the frequency of the transmitted wave will be different, and comparing both of them will help identify the speed of the vehicle. These guns suffer from a problem of beam divergence [5], which makes it difficult to which an individual vehicle cannot be targeted. Thus, the operator has to be trained and certified before using such guns [6]. To avoid such a problem, a LIDAR gun is used [7]. It does not suffer from beam divergence problems, and the speed of the individual vehicle can be determined. The speed identification using both the guns takes around half a seconds. The gun may have a screen or viewfinder which have a pointer to focus on the vehicle in question. It displays the speed of the vehicle on the screen. A sample view is shown in Figure 2.1. From the figure, it is evident that if the traffic is less than it is easy to identify the over-speeding vehicle. But with a high volume of the vehicle on the road, there are chances of misidentification of the vehicle. Also, a slight movement when the gun is pointed at a vehicle in question may result in the wrong vehicle identification; thus, although the LIDAR or RADAR gun is an excellent method to identify the over-speeding vehicle with skilled hands and eyes but not the most accurate one.

Whenever any individual is identified for over-speeding, no proof or cross verification method is present at either party i.e., with the sheriff or the driver. Hence, in the current system, there is no method available to verify if the vehicle in question is misidentified. This is a big problem. Because if the driver wants to fight the ticket in the court, he/she does not have proof to defend there claim. And during the trial, it becomes more of sheriff's word to drivers word. Sheriff being a part of the judicial system, will always be given more emphasis. Also, the sheriff always assumes that he/she has identified the vehicle correctly as no cross-verification is present. In such a scenario driver is entirely at a losing end of the bargain. Hence, we see that the current system does some dangerous loopholes and not robust.

In this project, a system is designed to fill these loopholes. As discussed in the introduction, a black box will provide the speed statistics of the driver to the sheriff. The sheriff can fetch the data over IoT device push it on to a cloud so that it is available to anyone in the judicial system. Data is represented to the sheriff in the simple graph with clear distinction in speed limits. A sample image is shown in Figure 3.7. This will just make sure that the decision, accusation, and citation are crosschecked, making the entire process more robust.

Chapter 3

System Overview

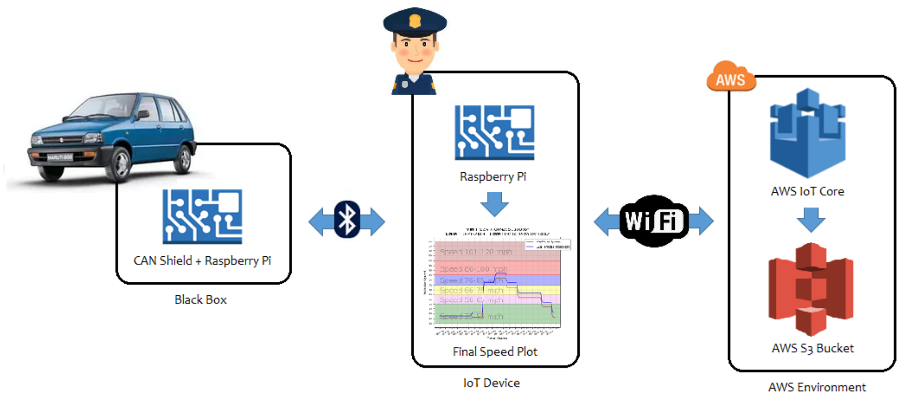


Figure 3.1: Architecture

The system is divided into three separate sections, as shown in Figure 3.1 viz Black Box, IoT Device, and AWS Environment. A black box will always be present in the car. It is implemented using a CAN shield (PiCAN2) and a Raspberry pi 3b+. Raspberry pi does not have a CAN interface. So, a CAN shield is used as a gateway module to interface CAN protocol from OBD-ii port and the SPI protocol of the Raspberry pi 3b+. This is discussed in Subsection 3.1.3. The

Raspberry pi 3b+ is programmed to fetch the speed, timestamp, and accelerator pedal position data from OBD-ii port and store it in a log file. IoT devices can request this data from a black-box whenever necessary. A black box will read the data only when the car is in motion. The vehicle data is sampled at one sample per second. A black box will share only the latest 1200 samples of vehicle data with the IoT device i.e., past 20 minutes before the vehicle came to a standstill.

IoT devices will always be present with the sheriff. IoT device is built using a Raspberry pi 3b and a Raspberry pi screen. It is programmed to initiate the request and fetch data from the black box over the Bluetooth. It will then send this data to AWS IoT Core over the internet using the MQTT protocol. MQTT is an abbreviation for Message Queuing Telemetry Transport. It is a lightweight messaging protocol designed specifically for communication between sensors, IoT devices, or mobile devices. This protocol is discussed in detailed in Section 4.4. The IoT device acts as an MQTT protocol publisher, and AWS IoT core acts as an MQTT broker. The AWS IoT core is used as it can receive data from multiple IoT devices. It then sends this data to an AWS S3 bucket. AWS S3 bucket is used to store vehicle data. Each message received from AWS IoT core is stored as a file with a user-assigned device-specific unique name in the AWS S3 bucket. This helps to maintain the files from different vehicles. The IoT device then downloads the data from the AWS S3 bucket and will make a plot of speed and gas pedal position versus time. This plot contains vehicle data for the past 20 minutes before it came to a standstill. A thing to note is that none of the user data is stored locally on the device; this is done to protect the user's privacy.

3.1 Black Box

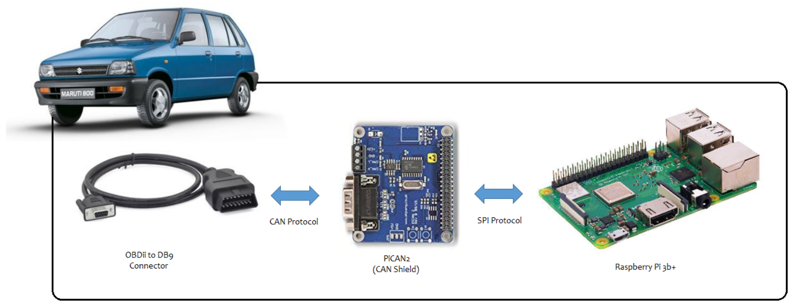


Figure 3.2: Black Box Side Architecture

Figure 3.2 shows the black box architecture. The CAN shield sits on the Raspberry pi 3b+. The connection is established using a 40 pin on board male-female connectors on Raspberry pi and CAN shield, respectively. The CAN shield has a DB9 female connector present on the board. Every car has an OBD-ii port as mandated by the California Air Resources Board or generally known as CARB. The details about the OBD-ii port will be discussed in Section 4.1. An OBD-ii to DB9 connector is used to connect CAN shield to the car. A thing to note is that the black box is powered from the car itself. OBD-ii port provides a 12V supply, and the CAN shield has a switch-mode power supply, which will convert this 12V to 5V and draws a current up to 1 Amp.

3.1.1 CAN Shield

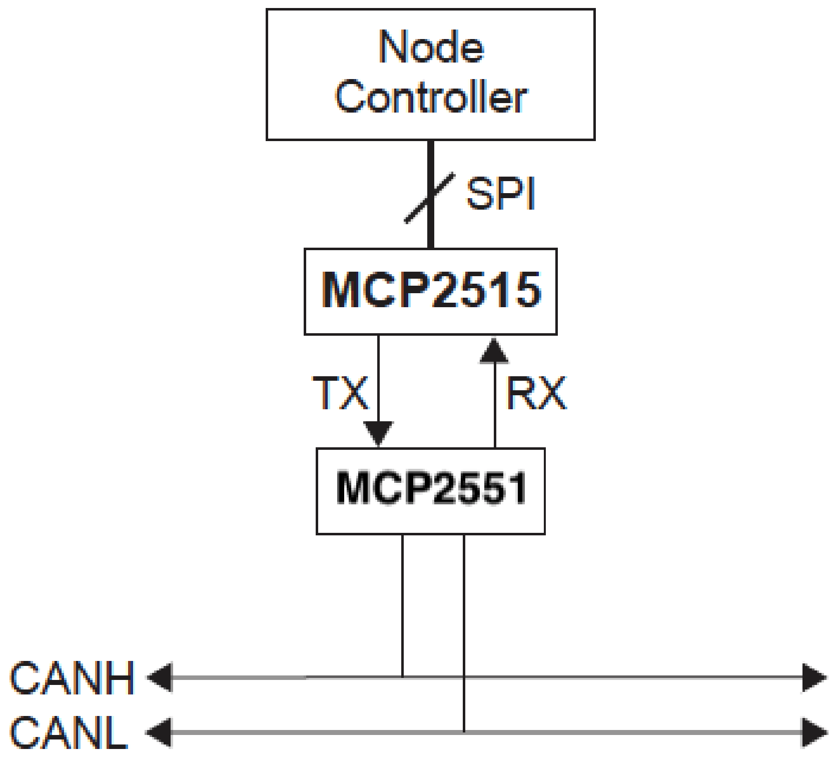


Figure 3.3: CAN Shield

For CAN shield, a PiCAN2 with Switch Mode Power Supply board from SK Pang electronics is used. A simple CAN shield outline is shown in Figure 3.3. The board consists of MCP 2515, a standalone CAN controller with SPI interface, and MCP 2551 CAN transceiver. Both the chips are compatible with ISO 11898 standard set for CAN protocol. A CAN controller requires CAN transceiver to convert received transmission signal to digital signal and vice versa when sending it over a CAN channel. MCP 2551 suffice this need on the PiCAN2 board. MCP 2551 is compatible with 12V and 24V systems. In this project, we use a 12V system drawn from the OBD-ii port [8]. It can handle up to 1 Mbps of data [9]. MCP 2551 can be operated in 3 different modes viz High Speed, Slope Control, and Stand-By Mode [9, 10]. As the name indicates, the high-speed mode and standby mode are used for high-speed data transfer and sleep, respectively. The mode can be selected using Rs pin on the chip. On the PiCAN2 board, the MCP 2551 chip is used in Slope Control mode [10]. This mode can be achieved by connecting a resistor between the Rs Pin and the ground. This mode is used to keep EMI emissions low by restricting the rise time and fall time at both the CANL and CANH pin low.

MCP 2515 standalone CAN controller is used for transmitting and receiving standard/extended data frames and remote frames. A CAN controller acts as a gateway to connect CAN protocol from the OBD-ii port to the SPI protocol on Raspberry pi 3b+ side. CAN controller consists of three main blocks [11]:

- **CAN Module:** It consists of a CAN engine block, Tx and Rx buffers, masks, and filters block. Masks and filters block is used to remove unwanted messages and reduce host (Raspberry pi in this case) overheads. The CAN engine block is used for reception and transmission of the message. To initiate message transmission, appropriate control registers and buffers are loaded. Control registers initiate transmission via SPI interface or by using transmit enable pins. Received CAN message is checked for errors and then verified against the filter values and then is transferred to the next buffer.

- **SPI Interface Engine:** SPI interface is used to connect the micro-controller(Raspberry pi in this case). The read and write to all the registers is done using a standard and specialized SPI commands.

- **Control Logic:** This block is used to provide setup and sequential operation of the other blocks in order to pass the information. A multi-purpose interrupt pin is available to indicate the reception of a valid message into the buffer. There are also three pins available for transmission of a message from one of the three transmit registers.

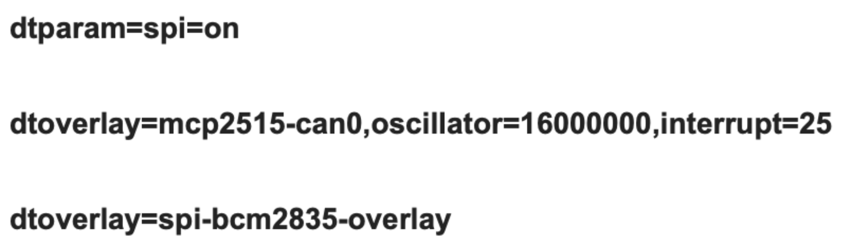
For this project, CAN and SPI are used at 500Kbps. This is a standard OBD-ii CAN speed for most of the vehicles available in the market.

3.1.2 Raspberry Pi 3b+

Raspberry Pi 3b+ is used as it is one of the most affordable powerful computers available in the market. Raspberry Pi 3b+ has 1.2 GHz 64-bit quad-core processor, an onboard IEEE 802.11n Wi-Fi, 5.50 Bluetooth, and four USB ports. For black-box Bluetooth and SPI interface are the only two peripherals used. Raspberry pi uses CAN shield with an SPI interface to fetch the data from the OBD-ii port over CAN protocol whereas Bluetooth is used to transmit the vehicle data to the IoT device. For Bluetooth PyBluez and for CAN PyCAN, Python package is used. The details of the Bluetooth is discussed in Section 4.3. A Python script is used to perform the fetch and transfer of the vehicle data from the car and to IoT devices, respectively. The settings needed for the interface of CAN shield and Raspberry pi is discussed in the next Subsection 3.1.3.

3.1.3 Interface between CAN Shield and Raspberry Pi 3b+

In Raspberry pi, a device tree is used to enable the peripherals. So in order to enable the SPI interface, **dtparam** is used. In second-line **dtoverlay** are used to set the parameters for the peripherals that should be enabled. In this case, MCP2515 is connected to the SPI interface. Also, the chip has an 8 MHz quartz, so the frequency of the oscillator has to be twice the crystal, which is 16 MHz [9]. GPIO pin 25 is connected to the interrupt pin of MCP2515, hence interrupt = 25. On the Raspberry pi side, the BCM 2835 SPI controller is used. It is enabled in the third line.



```
dtparam=spi=on

dtoverlay=mcp2515-can0,oscillator=16000000,interrupt=25

dtoverlay=spi-bcm2835-overlay
```

Figure 3.4: Configuration for SPI Interface

In order to interface CAN shield and Raspberry pi three lines shown in the Figure 3.4 are added to the config.txt file[8].

3.2 IoT Device

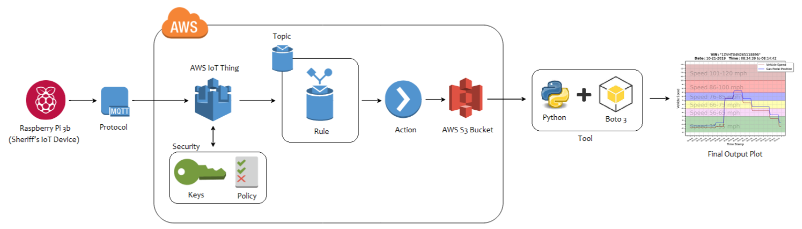


Figure 3.5: IoT Side Architecture

The architecture on the IoT side is shown in Figure 3.5. It consists of two sections a Raspberry pi 3b and AWS IoT and Cloud infrastructure. A Python script is implemented on Raspberry pi, which will perform tasks such as fetching of the vehicle data from the black box, sending this data to the AWS infrastructure over MQTT, fetching data from the cloud and creating the plot of the data.

3.2.1 Raspberry Pi

A Raspberry pi 3b and the Raspberry pi screen is used as an IoT device. Bluetooth and Wi-Fi are the peripherals used from the Raspberry pi. Python package PyBluez, AWS IoT SDK, AWS Boto, and Boto3 are also installed. PyBluez is used for Bluetooth; AWS IoT SDK is used to set up and configures MQTT protocol, whereas AWS Boto and Boto3 are used to fetch the vehicle data files from AWS S3 bucket. Three different files with encryption keys are used for secure communication between Raspberry pi and AWS IoT Core. The generation of these keys and using them will be discussed in Appendix A I. IoT device will initiate the request for vehicle data i.e., speed, timestamp, and gas pedal position from the black box over the Bluetooth. It will fetch the latest 1200 data points and then send it to AWS IoT Core over MQTT protocol. IoT device then will download all the vehicle data from the cloud and create a speed and gas pedal position versus time plot. Both these tasks are done using a single Python script. A thing to note here is that once the data is sent to AWS IoT Core, it is instantaneously sent AWS S3 bucket.

3.2.2 AWS Infrastructure

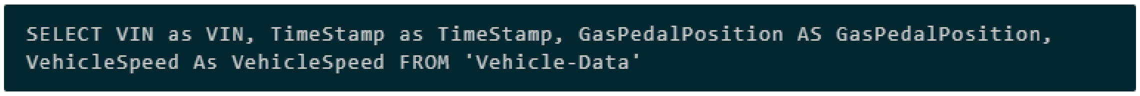
Following are the components of used from AWS [12]:

- **AWS IoT Thing:** A thing is a block that will communicate directly with the sheriff's IoT device. The communication between AWS IoT Thing and IoT Device is encrypted. A private and RootCA1 key and certificate are required on both the ends for reception and transmission of the messages. These keys are unique for each IoT thing. Thus, for multiple IoT devices, multiple IoT things should be created, and each pair are having its own unique set of keys. Keys are created by AWS IoT Core.

- **AWS Policy:** A policy is used to enable connection, subscription, reception, and publishing for the IoT thing. These terms are MQTT protocol specific and will be discussed in

Section 4.4. The same policy can be attached to multiple IoT things.

- **AWS Rule:** A rule is used to extract a specific information form the received message for a particular topic. As seen in Figure fig:Rule-created-for, the information extracted for the topic Vehicle-Data is VIN number, Time Stamp, and Gas Pedal Position. Similar to the policy, the same rule can be attached to multiple IoT things.



```
SELECT VIN as VIN, TimeStamp as TimeStamp, GasPedalPosition AS GasPedalPosition,
VehicleSpeed As VehicleSpeed FROM 'Vehicle-Data'
```

Figure 3.6: Rule created for the IoT thing

- **AWS Action:** Once the data is extracted from the message we have to store in the AWS S3 bucket. This step is called the action. Each message is stored in a separate JSON file with the current time appended to its name. The time is in epoch format.

- **AWS S3 Bucket:** As the name suggests AWS S3 bucket is used to store the information from all the IoT devices.

The stepwise configuration for all the blocks will be discussed in detail in Appendix A I.

3.2.3 Data Representation

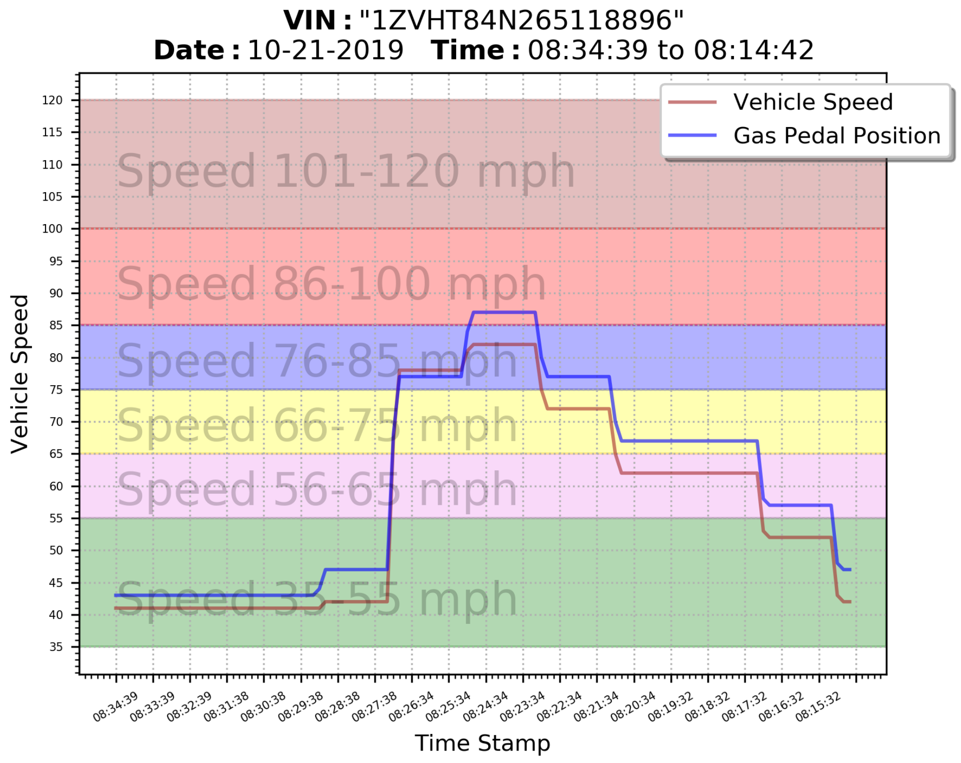


Figure 3.7: Final Speed Plot

A tool is developed using Python AWS Boto and Boto3 package. These packages are used to access the AWS S3 bucket. The tool downloads all the JSON files, extract the vehicle data, and plot speed, the gas pedal position with respect to time, as shown in Figure 3.7. The tool scales the 1200 points data to 120 points by averaging the speed for the 10 points. A similar is done for the gas pedal position. As the vehicle movement is not significant in just 10 seconds, so the average of the speed can be considered. Also, the code has a functionality where users can make a plot of all the 1200 points instead of 120 points if need be.

Figure 3.7 shows the final speed plot. As we can see plot has the VIN number at the top to identify the vehicle, the date when the incident occurred, and the time span for which the data is being plotted. The vehicle speed and gas pedal position have a resolution of 1. The gas pedal position is in percentage. So, it can go from 0 to a maximum of 100. The gas pedal position is used to estimate the stopping distance since the acceleration applied was zero. The graph has separate sections to indicate the various speed limit on roads such as city roads, interstate highways, and national highways. Such type of clear distinction helps the sheriff to identify the violation of speed limit by the driver if any.

3.3 Software Flow

This section will explain the software/Python scripts flow for the project. The software is divided in two Python scripts. One runs on the Raspberry pi of the black box and the second one on Raspberry pi of IoT device. Next four section will discuss in details of the working these Python scripts.

3.3.1 Software Flow for Black Box Side

As mentioned in the Section 3 the black box side has, Raspberry pi has two Python packages installed on it viz PyCAN and PyBluez. OBD-ii port uses CAN protocol for communication. PyCAN package in Python provides built-in API for sending and receiving CAN messages. It also provides the user with ease of CAN message configuration such as arbitration id, data, data length code, etc. One more thing to note is the OBD-ii Parameter ID's or more commonly know as OBD-ii PID's or just PID's. PID's are standardized code set by SAE (Society of Automotive Engineer). PID's code is used to request specific information such as vehicle speed, engine coolant temperature, gas pedal position, etc. from the car. The details about the PID's is discussed in Section 4.1.

PyBluez [13] provides simple implementation for Bluetooth communication. In this project, a serial port profile with a client-server communication is used. PyBluez provides with a simple client-server communication example. The Bluetooth communication developed for this project is based on those examples. The details about this protocol and module are discussed in Section 4.3.

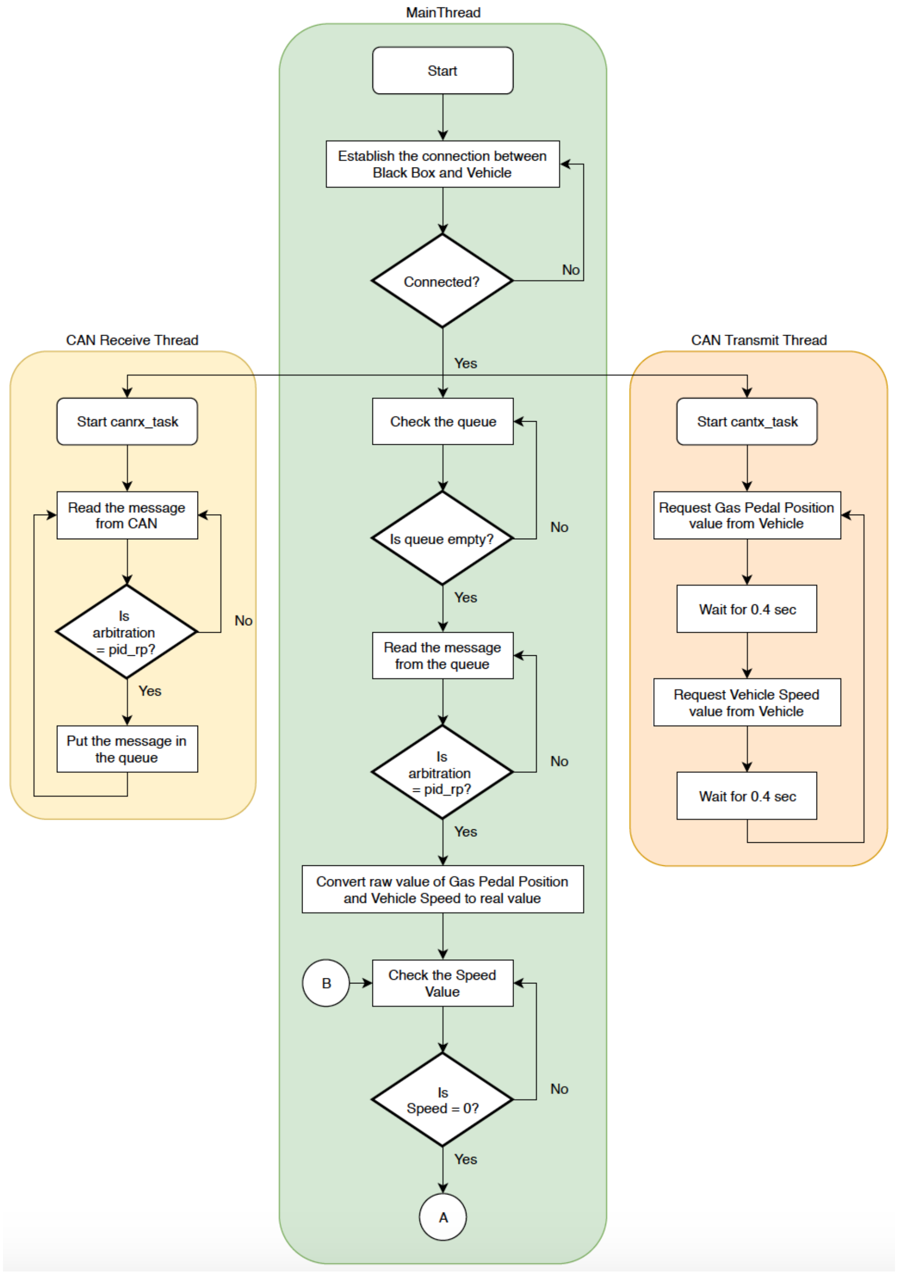

MainThread
Start
Establish the connection between Black Box and Vehicle
Connected?
No
Yes
CAN Receive Thread
Start canrx_task
Read the message from CAN
Is arbitration = pid_rp?
No
Yes
Put the message in the queue
Check the queue
Is queue empty?
No
Yes
Read the message from the queue
Is arbitration = pid_rp?
No
Yes
Convert raw value of Gas Pedal Position and Vehicle Speed to real value
B
Check the Speed Value
Is Speed = 0?
No
Yes
A
CAN Transmit Thread
Start cantx_task
Request Gas Pedal Position value from Vehicle
Wait for 0.4 sec
Request Vehicle Speed value from Vehicle
Wait for 0.4 sec

Figure 3.8: Software flowchart for Black Box Part-I

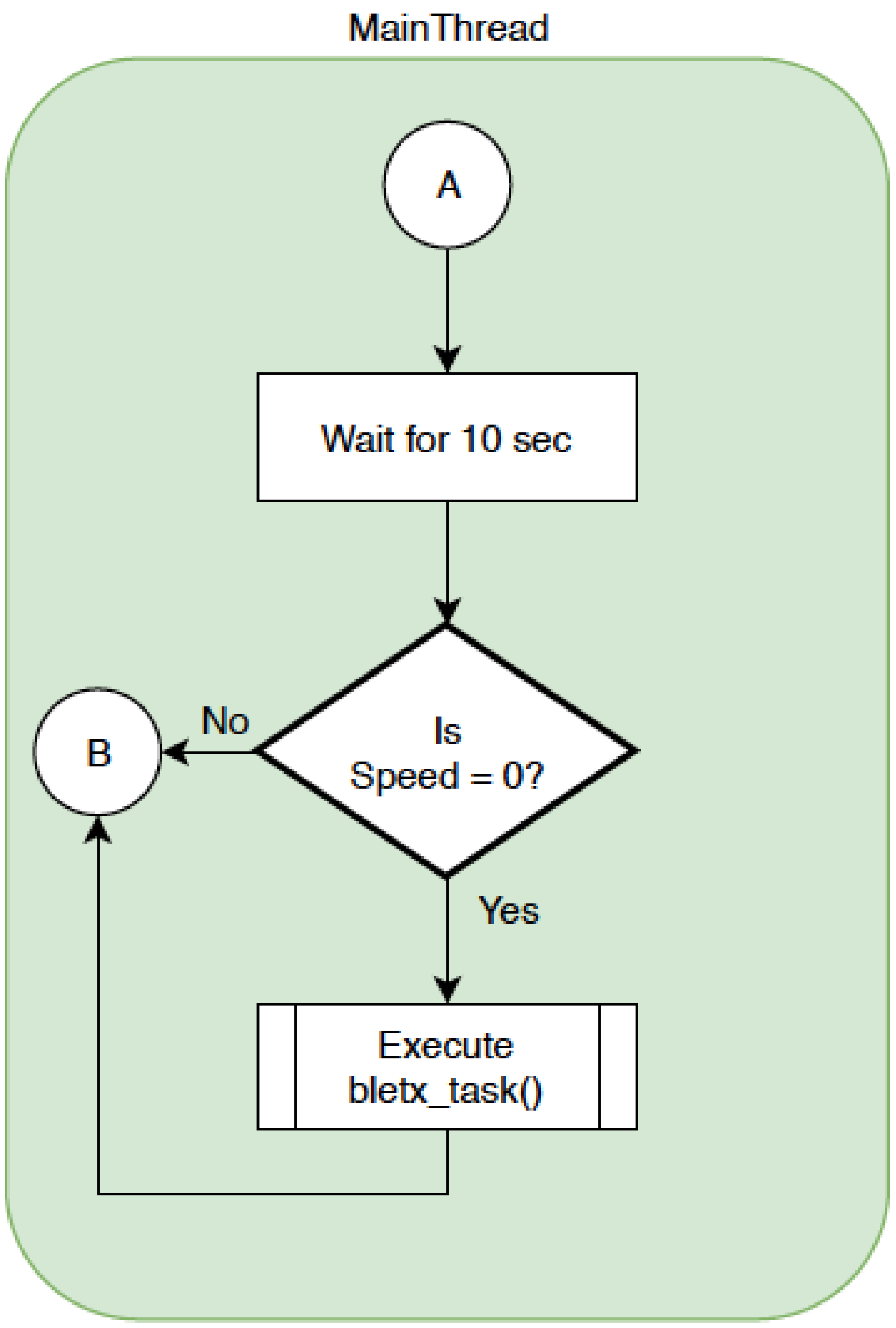


Figure 3.9: Software flowchart for Black Box Part-II

Figure 3.8 and 3.9 shows the flow chart for the software in the Python script, developed for black box Raspberry pi. As we can see from the image there are three threads being used. The main thread, a CAN receive thread and a CAN transmit thread. The detailed working of the threads are given below:

- **Main Thread:** This thread establishes a connection between the black box and the car via CAN shield. If the connection is successful, it starts CAN receive thread and CAN transmit thread. If the connection fails, then it re-tries to establish the connection. A message queue is created to store the received messages from the car in CAN receive thread. The original thread checks if a message is present in the queue. In order to check if the message is valid, the arbitration id of the received CAN message is checked. It has to be **0x7E8** [14]. The received values form the car are raw data. Thus, they are converted to readable format i.e., and the timestamp is epoch for so it is converted standard HH:MM:SS format, the vehicle speed is converted to miles per hour and gas pedal position to a percentage from 0 to 100. This data is written into the log file only if the vehicle is in motion. This is done in order to avoid unnecessary repetitive data points. This mechanism is also used to trigger the Bluetooth function. This function is discussed in a separate point later. Thus, the main thread keeps tracking the vehicle data and write it to the log file if the vehicle is in the motion. It also monitors the nearby sheriff's device if the car is not in motion. This is done to simulate that the vehicle is actually stopped by the cop for the over-speeding violation. An important thing to note here is that the vehicle data is written in the JSON format into the log file. This is done because of the AWS IoT core transfer of the data to the AWS S3 bucket in JSON format. The format of the data stored is given below:

```
{"VIN": "1ZVHT84N265118896", "TimeStamp": 1568463191, "GasPedalPosition": 47,
↪  "VehicleSpeed": 42}
```

- **CAN Transmit Thread:** This thread requests the gas pedal position and vehicle speed for the current timestamp. The request is sent per second. The reason for the selection of this sample rate is discussed in Section 3.4. To request data from car an arbitration id **0x7DF** [13] is sent along with PID's (type of data to be requested) in the CAN message. For this project, we need three vehicle-specific data, VIN number, vehicle speed, and gas pedal position. The PID's for vehicle speed is **0x0D**, and gas pedal position is **0x49** [14]. The timestamp is provided by default. The VIN number request from the vehicle is OEM specific. The being used for testing the project did not provide with VIN number. So, vehicle identification has been hardcoded. The details of the CAN transmit message request is discussed in Section sec: can. This thread runs continuously and keeps sending the messages.

- **CAN Receive Thread:** This thread is used to collect the response from the car. The thread will check for valid arbitration id i.e. **0x7DF** [15]from the received message. It will then store it in a message queue. The message from this queue is used by the main thread for raw data conversion and storing it in the log file.

Figure 3.10 shows the software flow for the script of Bluetooth data transfer to the IoT device. One thing to note is that both the black box and IoT devices are already paired manually. This can be done by the script, as well. So whenever a Bluetooth function is called, it searches for the nearest IoT device. As both the device are already paired, they will connect immediately. Once the connection is established latest 1200 readings are read from the log file and sent to the IoT device. A serial port profile is used in the client-server mode for Bluetooth communication. Bluetooth is discussed in detail in Section 4.3. The Bluetooth communication used is unidirectional. A black box acts as a client, whereas the IoT device acts as a server.

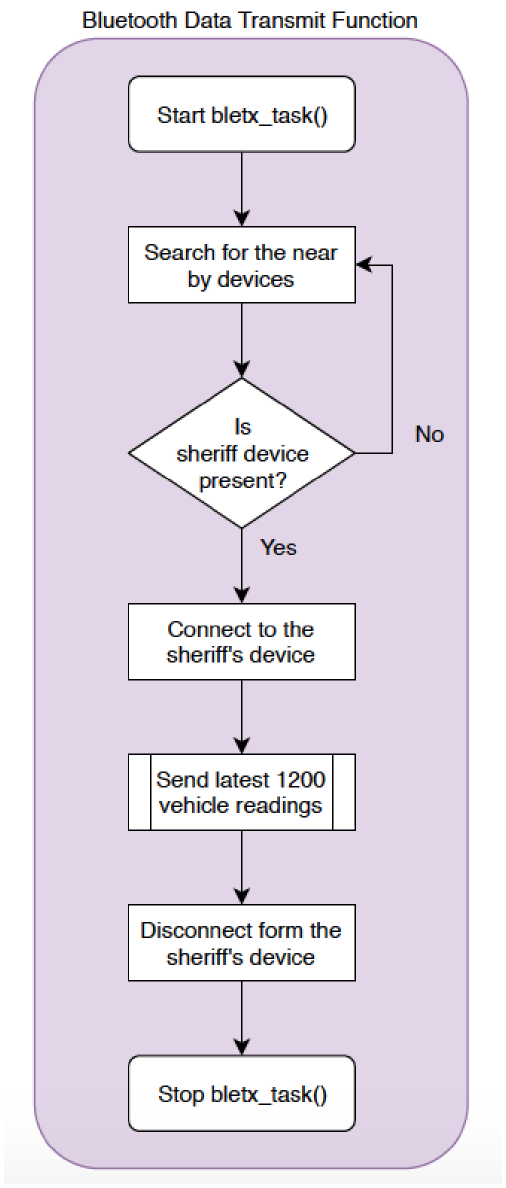

Bluetooth Data Transmit Function
Start bletx_task()
Search for the near by devices
Is sheriff device present?
No
Yes
Connect to the sheriff's device
Send latest 1200 vehicle readings
Disconnect form the sheriff's device
Stop bletx_task()

Figure 3.10: Bluetooth Transmit Function

3.3.2 Software Flow for IoT Device Side

As mentioned in the Section 3 IoT device, Raspberry pi has three Python packages installed on it viz AWS IoT SDK, Boto, and Boto3. AWS IoT SDK is used to set up the MQTT protocol for communication between the IoT devices and AWS IoT Core, whereas Boto and Boto3 [16] are used to access the AWS S3 bucket. Both these packages provide various API's, which helps to simplify complex operations related to AWS infrastructure. WiFi and Bluetooth peripherals are used on IoT devices. The IoT device is connected to WiFi manually. As mentioned in Subsection 3.3.1, both the IoT and black box are paired manually.

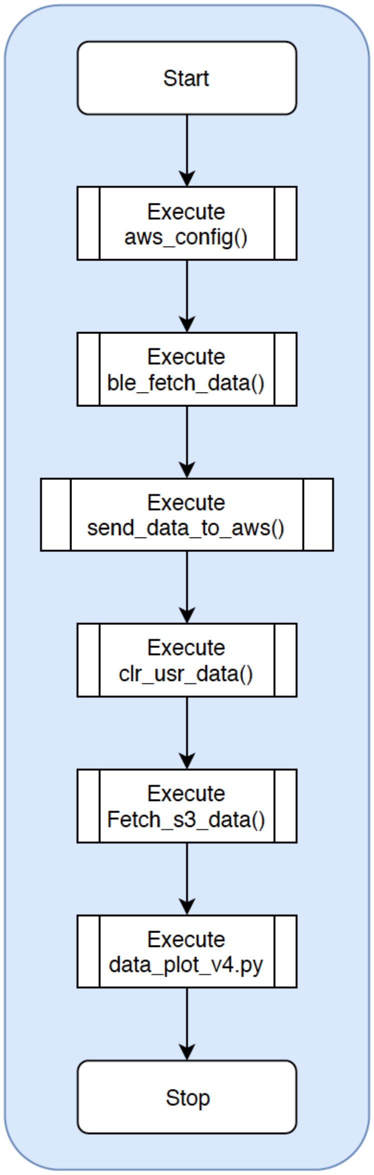


Figure 3.11: IoT Device Main Function

The Figure 3.11 shows that the entire script is divided in to 5 functions and a Python script used to plot the data from the cloud. The details of the functions are discussed below:

- **aws_config():** This function is used to configure the MQTT protocol on the IoT device.

A handle is created, which points to the AWS client connection. The settings for MQTT protocol are configured to this handle. A thing to consider is that the handle is created using AWS IoT SDK provided API, so this handle should all the configuration parameters required to initiate the MQTT protocol. Once the handle is created and configured a host id and port number of the AWS IoT Core is assigned. The host id differs as per the account. It is present in the settings section of the AWS IoT core website under custom endpoint. The port number for AWS IoT core always be the same for everyone. The port number for AWS is 8883 [12]. For the secure communication between the IoT device and AWS IoT core X.509, the cryptography standard is used by AWS. AWS IoT core generates 3 keys for each IoT thing. The creation of the keys is discussed in detail in Appendix A I. These keys have to present at both points of the communication in our case IoT device and AWS IoT core. Other things to be configured are disconnect timeout time and operation time out. Disconnection timeout means that if the IoT device and AWS IoT core are disconnected, it will wait for 10 seconds to establish the new connection automatically. The code is set to 10 seconds. Operation time out means if the send or reception of data is lost, then the retransmission or reception is initiated after this time. In the code, it is set to 5 seconds.

- **ble_fetch_data():** As both IoT devices and black boxes are already paired, the name of both the devices is known. The IoT device starts the Bluetooth and starts the advertisement of its device name and UUID number using a serial port profile. Whenever the black box is in the vicinity of the IoT device, which is about 1 to 10 meters, both the device automatically connects. Once the connection is established, the IoT device acts as a server and requests the vehicle data from the black box. When all the 1200 data points are received, the Bluetooth is disconnected. A socket model is used for Bluetooth implementation. If there is any data loss i.e., a number of data points received are less than 1200 points, and then a message is popped up, telling the sheriff to reinitiate the data transfer. All the received data is stored in a temporary array.

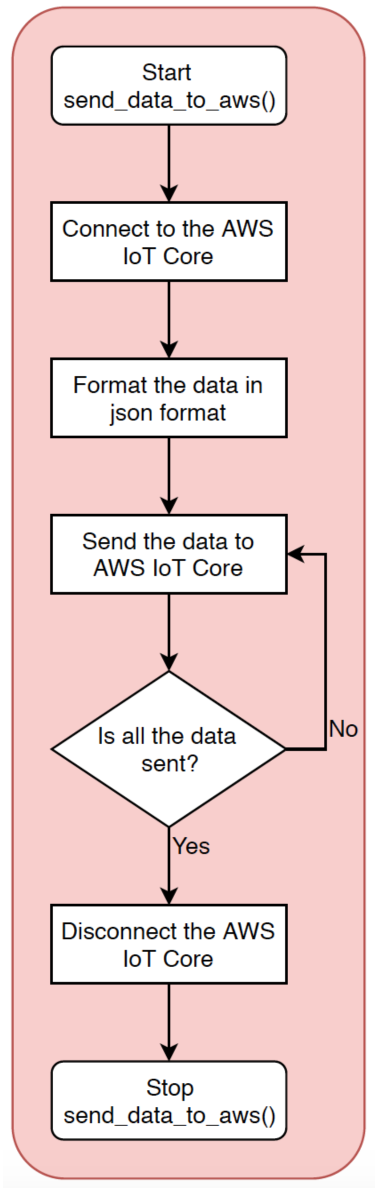

Start
send_data_to_aws()
Connect to the AWS IoT Core
Format the data in json format
Send the data to AWS IoT Core
Is all the data sent?
No
Yes
Disconnect the AWS IoT Core
Stop
send_data_to_aws()

Figure 3.12: AWS IoT Core Software Configuration

- **send_data_to_aws():** The flow for this function is shown in Figure f3.12. An AWS handle is already created in aws_config() function. An API <awshandle>.connect() is used to connect to the AWS IoT core. Once the connection is established, the vehicle data received from the black box sends to the AWS IoT core. When all the 1200 data points are sent, the IoT device disconnects from the AWS IoT core. An API <awshandle>.disconnect() is used for disconnection.

- **clr_usr_data():** This function is used to delete all locally store user data. This is done to protect users privacy.

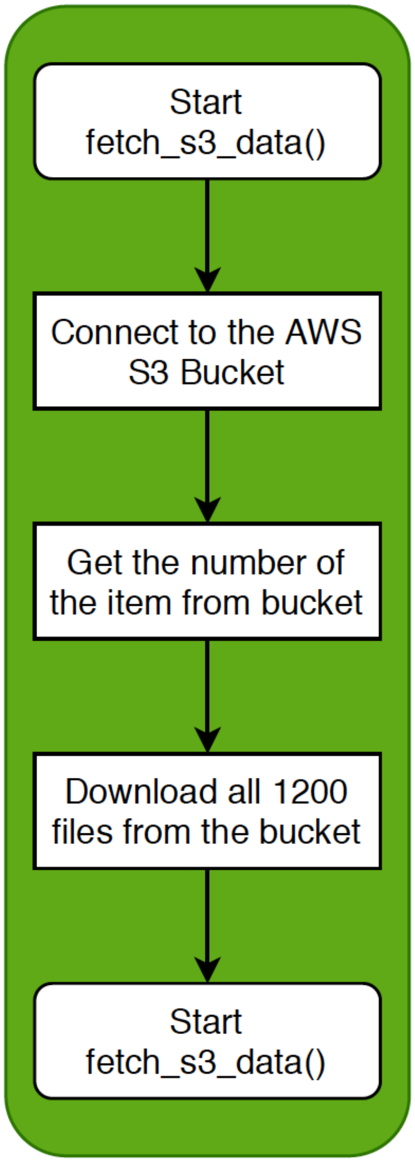

Start
fetch_s3_data()
Connect to the AWS S3 Bucket
Get the number of the item from bucket
Download all 1200 files from the bucket
Start
fetch_s3_data()

Figure 3.13: AWS S3 Bucket

- **fetch_s3_data():** This function is used to fetch the data from the AWS S3 Bucket. A bucket with the name "www.giotdata-east.com" is created in the AWS S3 bucket [16]. The process of creating an AWS S3 bucket is mentioned in Appendix A I. In order to access this bucket using a script, an access key and secret key are required. These keys are created in the AWS environment. The process to generate these keys is mentioned in Appendix A chap: Guides. A handle is created for a connection to the AWS S3 bucket. The region, access key, secret key, and the bucket name is assigned to this handle. The region used in this project is US-East-1. One can use any region they want; it does not make a difference. After the assignment and API <awsbuckethandle>.connect_to_region() is used to connect the S3 bucket. After this connection is established, another API <awsbuckethandle>.list() is used to fetch the count of the number of files present in the S3 bucket. Now the content of each file is fetched using a API <awshandle>.get_contents_to_filename(). When all the 1200 JSON files are downloaded, the IoT device disconnects from the AWS S3 bucket.

- **data_plot_v4.py:** This file is executed from the main IoT device Python script. This script reads all the 1200 JSON files and extract information viz VIN number, vehicle speed, gas pedal position, and time stamp and store it in different arrays. The script also covert the time stamp from epoch to HH:MM:SS. It then plots the graph for vehicle speed, gas pedal position versus time stamp.

3.4 Key Consideration for the System Design and Components

This section discusses reason to choose various components and design decision.

- **Raspberry Pi 3b/3b+:** Raspberry pi is one of the most affordable computers available on the market. It is small and has many useful peripherals such as onboard WiFi, Bluetooth 4.2 and upwards, SPI Interface with speed up to 10MHz Ethernet, HDMI input, 4 USB ports, etc. Raspberry pi used runs on Debian OS, which is free. Access and control of the peripherals are relatively easy using Python scripts. Many Python packages are available to use these peripherals and easy integration of AWS infrastructure. As the OS used is open source, the plethora of literature is available on the internet.

- **CAN Shield:** The reason to choose this board was its versatility and support form the manufacturers. It is to easy interface to with the Raspberry pi and user manual guides user in a stepwise manner to establish the connection. Users can test the setup of the CAN shield and Raspberry pi by using a loopback functionality available. This can be done without actually connecting it to the external CAN bus.

- **AWS Infrastructure:** AWS IoT Core can connect and fetch information from multiple IoT devices. It has a very sophisticated GUI available along with an ample amount of documentation and sample code provided by AWS.

- **MQTT Protocol:** MQTT is a lightweight, easy to implement, and secure protocol widely used for IoT communication. AWS provides a package to implement and communicate with its infrastructure over MQTT. This was one of the main reasons to choose this protocol.

- **Bluetooth Protocol:** Bluetooth is simple, secure, and easy to implement the wireless protocol. For the communication of two Bluetooth devices, both of them have to be paired and authenticated. This makes sure that the data is not being transferred to any unauthorized device. Also, Bluetooth devices are inexpensive, readily available in the car and market, respectively.

- **Sampling Rate:** Human reaction time is a time elapsed after stimulus action until a reaction/response occurs from a human. Numerous research has been done on this topic and they estimate it to be around 2.3 seconds [17, 18]. Thus, the sampling rate has to be faster than human reaction time. Hence, the sampling rate of one second per second is appropriate as it is at double than the human reaction time.

3.5 Cost Break-Down for the Project

Sr. No.	Name of Equipment	Quantity	Cost ($)
1.	Raspberry Pi	2	70
2.	CAN Shield	1	59
3.	OBD-ii to DB9 Cable	1	28
4.	AWS S3 Storage (Variable)	N/A	0.021/GB
5.	AWS S3 Access (Variable)	N/A	0.0125/GB
	Total Cost		157+

Table 3.1: Overall Final System Cost

Sr. No.	Name of Equipment	Quantity	Cost ($)
1.	Raspberry Pi	1	35
2.	CAN Shield	1	59
3.	OBD-ii to DB9 Cable	1	28
	Total Cost		122

Table 3.2: Cost to driver

Sr. No.	Name of Equipment	Quantity	Cost ($)
1.	Raspberry Pi	1	35
2.	Raspberry Pi Screen	1	60
3.	AWS S3 Storage (Variable)	N/A	0.021/GB
4	AWS S3 Access (Variable)	N/A	0.0125/GB
	Total Cost		35+

Table 3.3: Cost per Sheriff

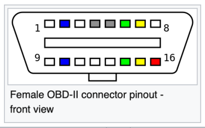
Sr. No.	Name of Equipment	Quantity	Cost ($)
1.	Raspberry Pi	2	70
2.	CAN Shield	1	59
3.	OBD-ii to DB9 Cable	1	28
4.	AWS S3 Storage (Variable)	N/A	0.021/GB
5.	AWS S3 Access (Variable)	N/A	0.0125/GB
6.	Raspberry Pi Screen	1	60
7.	Raspberry Pi Kit	1	35
8.	Raspberry Pi Keyboard	1	18
	Total Cost		270+

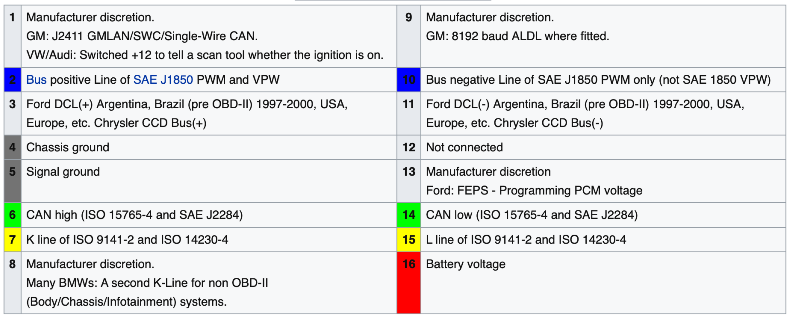
Table 3.4: Overall Development Cost

Chapter 4

Protocols and Standards

4.1 OBD-ii Standard

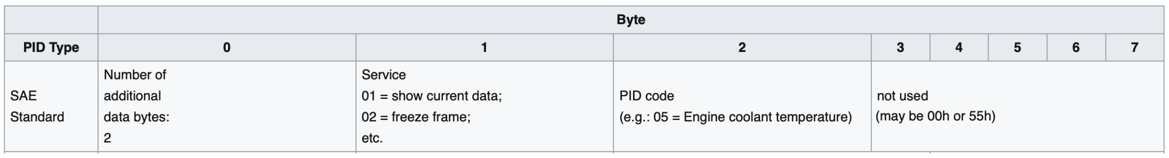




1	Manufacturer discretion. GM: J2411 GMLAN/SWC/Single-Wire CAN. VW/Audi: Switched +12 to tell a scan tool whether the ignition is on.	9	Manufacturer discretion. GM: 8192 baud ALDL where fitted.
2	Bus positive Line of SAE J1850 PWM and VPW	10	Bus negative Line of SAE J1850 PWM only (not SAE 1850 VPW)
3	Ford DCL(+) Argentina, Brazil (pre OBD-II) 1997-2000, USA, Europe, etc. Chrysler CCD Bus(+)	11	Ford DCL(-) Argentina, Brazil (pre OBD-II) 1997-2000, USA, Europe, etc. Chrysler CCD Bus(-)
4	Chassis ground	12	Not connected
5	Signal ground	13	Manufacturer discretion Ford: FEPS - Programming PCM voltage
6	CAN high (ISO 15765-4 and SAE J2284)	14	CAN low (ISO 15765-4 and SAE J2284)
7	K line of ISO 9141-2 and ISO 14230-4	15	L line of ISO 9141-2 and ISO 14230-4
8	Manufacturer discretion. Many BMWs: A second K-Line for non OBD-II (Body/Chassis/Infotainment) systems.	16	Battery voltage

Figure 4.1: OBD-ii Port and Pin description

OBD-ii port is mandated by the California Air Resources Board or generally known as CARB, to be present in all the cars that are on the roads since 1996. The pin layout fo the port is shown in Figure 4.1. It is generally present below the steering wheel in the cars. For this project, only CAN high CAN low, and Battery voltage lines are used. As the name suggests, the port is used to fetch the diagnostic information about the vehicle's health. This topic is vast and not in scope for this project. We discuss only OBD-ii PID's, which are used in the project. An OBD-ii PID's or OBD-ii Parameter Identifiers [19] is a set of commands that are sent to the car via OBD-ii port form an external device. The device can be any laptop, computer, or specialized device having software designed to communicate with the car over CAN. PID's are code used to request information about cars such as engine coolant temperature, vehicle speed, valve position. For this project, the black box requests the gas pedal position, vehicle speed, and time stamp for the same from the car. The data is requested in the specific format shown in Figure 4.2.



	Byte							
PID Type	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
SAE Standard	Number of additional data bytes: 2	Service 01 = show current data; 02 = freeze frame; etc.	PID code (e.g.: 05 = Engine coolant temperature)	not used (may be 00h or 55h)				

Figure 4.2: Format for PID request

A typical CAN data frame consists of 8 data bytes. Thus, we send 8 bytes of data. The first byte indicates the number of bytes to be sent. As we are going to send only the type of service and PID code, it is always set to **0x02**. OBD-ii standard provides different services such as fetch current data from the car, show diagnostic trouble codes, clear diagnostic trouble code. For this project, we are using service 0x01, which is fetching the current data as we need a real-time speed and gas pedal position. The third byte is the PID code. The PID code for speed and gas pedal position is **0x0D** and **0x49**, respectively. All other bytes are set to zero. In a CAN frame, an arbitration id needs to be set. An arbitration id is an address or the device id of an ECU (Electronic Control Unit). For the PID request, this id is **0x7DF**. A sample message request for vehicle speed is **7DF 02 01 0D 00 00 00 00 00**. A thing to note is that all the data sent is in hexadecimal format [14]. To such a request, the respective ECU responds with the same format of CAN message, but only the arbitration id is different. As per the OBD-ii standard, any device connected to OBD-ii port has an address or an arbitration id as **0x7E8**. So in the script to check if a valid message is received, this arbitration id is checked. The data received from the car is in raw format that is a fixed point data. This data is converted into standard or readable data. Received data is converted to miles per hour for speed and percentage for the gas pedal position.

The conversion formula [14] for speed is:

$$Speed_{mph} = \frac{1}{1.609344} * ReceivedValue \tag{4.1}$$

The conversion formula [14] for gas pedal position is:

$$GasPedalPosition_{\%} = \frac{255}{100} * ReceivedValue \tag{4.2}$$

The received values are in hexadecimal. They are converted to decimal and than used in formulas.

4.2 CAN

CAN is an abbreviation for Controller Area Network. It is a two-wire, multi-master slave, broadcast protocol. It has two wire called CAN high and CAN low. All the nodes are connected to these two lines, as shown in Figure 4.3. Any ECU can transmit the data if the bus is free. As this is a broadcast protocol, all the ECU receives the message. All ECU checks for the arbitration id of the message. Arbitration id is the address of the ECU for which the message is to send. If the arbitration id matches, then the ECU accepts the message and responds. As mentioned in

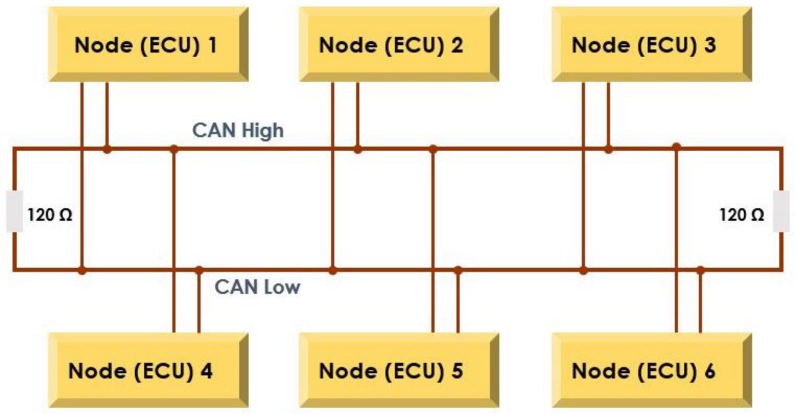


Figure 4.3: CAN network

Section 4.1, the arbitration id used for requesting the vehicle data is **0x7Df**. So when black box sends this arbitration id on to a CAN channel for data request. The CAN message is sent to all the ECU in the car. The ECU, which is responsible for that particular data request, responds with the data. OBD-ii port is like an open connection in the network where an external device can be connected [20].

A typical CAN frame is shown in Figure 4.4. The CAN message used in this project is the standard format. As PyCAN package [21] is used setting different fields in the message using API is simple. Only the arbitration id and data field are set in the Python script everything else such as DLC, CRC, RTR bit, etc. are taken care of by the PyCAN package and the CAN controller on the CAN shield.

4.3 Bluetooth

Bluetooth is a wireless protocol. It uses the unlicensed ISM 2.4GHz wireless link to send and receive data [22]. It is a secure method of communication for short-range data transfer. It supports

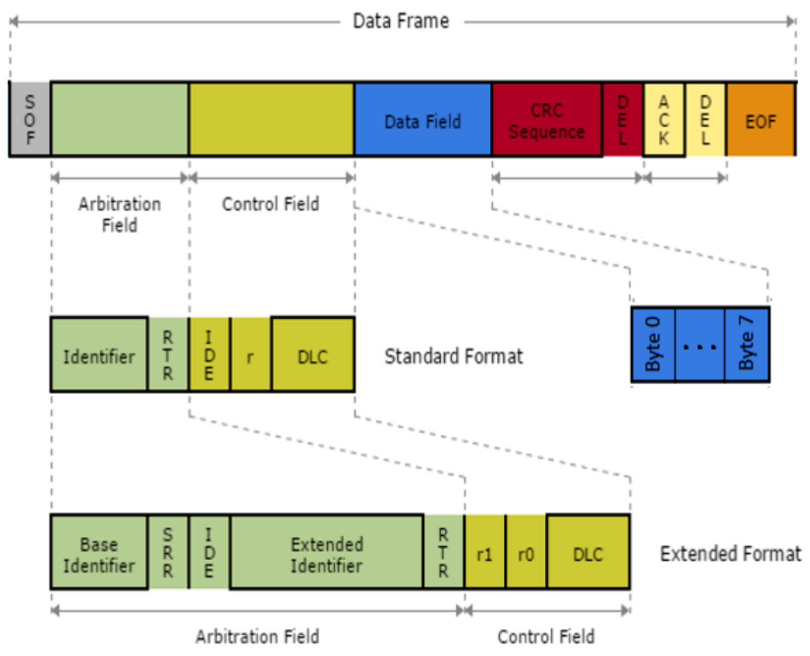


Figure 4.4: CAN frame format

up to 1Mbps of data transfer. Both the Raspberry pi used is Bluetooth 4.2 and upward compatible. As per the Bluetooth standard, a specific profile needs to be used for communication. A profile is a way in which data is shared between devices. In this project, the serial port profile is used in client-server mode. A serial port profile means that the device sends or receives data from the Bluetooth module over its UART interface. The method to activate the serial port profile on Raspberry pi is discussed in Appendix A I. In a client-server model, a server can request data from the client, and the client sends the data. The client-server communication is unidirectional. Bidirectional communication can be used in this mode but is not advisable. A black box acts as a client, and the IoT device acts as a server. A Bluetooth code is built on the sample code provided in the PyBluez Python for the Raspberry pi. Another important thing to consider is the Universal Unique Id or UUID and device address. A UUID can be a 16 or 128-bit unique id used by the Bluetooth devices to identify the type of services and its attribute [23]. The UUID present in the code is used from the client-server communication as the same profile is used. The device address is the physical address of the Bluetooth chip. It is like a MAC address for the Bluetooth chip.

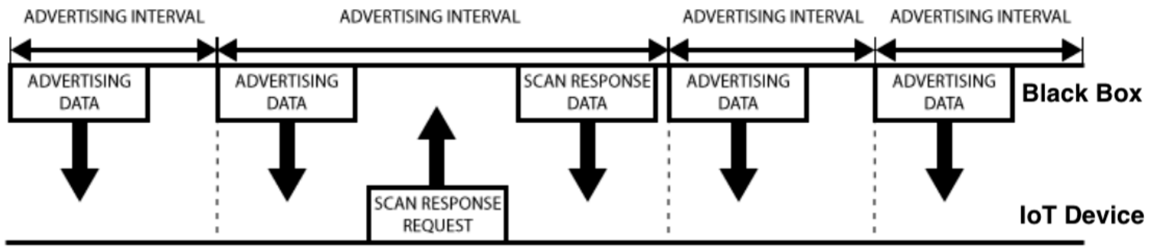


Figure 4.5: Bluetooth data exchange

Bluetooth profile works on the advertise and scan method. As shown in Figure 4.5. A server advertises that it is available for connection, and the client scans for the available server devices. Before the start of the data transfer, both the Bluetooth devices need to be paired [13]. This is done manually, and the process is discussed in Appendix A I. Once the device is paired, the Bluetooth on both the devices is turned on. The address of the assigned name of the Bluetooth device is pair displayed on both the ends. As they are paired, the devices connect and start the data transfer. The connection of both the devices and data transfer control is made using a Python script. PyBluez does provide with very efficient API's to do these operations.

4.4 MQTT

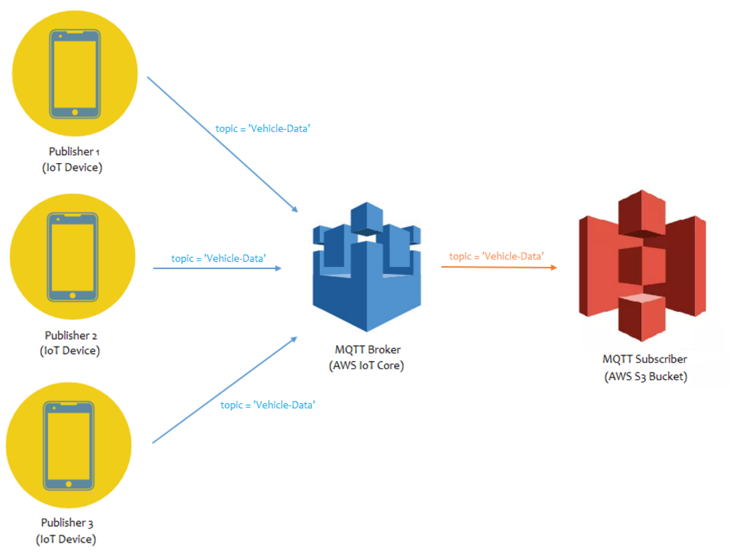


Figure 4.6: MQTT network

Message Queueing Telemetry Text or commonly know as MQTT protocol. It is designed for lightweight, secure communication between IoT devices, sensors or a mobile device [24]. The protocol is versatile and easy to implement. It runs on TCP/IP and works on publish and subscribe model. In such models, the data is shared only amongst devices that are publishing and subscribed to a particular topic. A topic can be considered as a filter used by communicating devices before accepting the message [25]. A simple MQTT network representation is shown in Figure 4.6. MQTT protocol has three main components:

- **Publisher:** It is a device that actually generates/sends the data. In this case, it will be IoT devices. IoT device receives the data form the black box then converts it to a JSON format and sends it to AWS IoT core.

- **Broker:** It can be a device or software which receives the data from the publisher and routes it to the subscriber. It acts as a gateway for the message. AWS IoT core acts as an MQTT broker. It receives a message from all the IoT devices/publishers and separates them as per the topic then pushes them to the AWS S3 bucket.

- **Subscriber:** It can be a device or software which receives the message and acts as an endpoint for the message. In this case, it is an AWS S3 bucket. All the messages from the MQTT broker/AWS IoT Core are stored in the JSON file with current epoch time appended to its name. The message can be stored in any text format and any supported text file format. The reason for choosing JSON was the ease of creation and extraction of the data using the keywords/parameters.

Chapter 5

Tests and Results

5.1 Black Box Script Testing

The black box script was developed in parts.

5.1.1 CAN shield and Raspberry Pi 3b+ Interface Testing

- The first part was checking the interaction between Raspberry pi and CAN shield. The CAN controller on the CAN shield has loopback functionality available on it. This functionality sends loopback/send back any data it receives from the node(in this case, it is Raspberry pi). For this testing, no black box i.e. CAN shield and Raspberry pi, is used with the screen, keyboard, and a power supply.

- Before starting the testing the SPI is enabled and the PyCAN Python package is installed on the Raspberry pi for the black box. Now in the terminal following lines of code is entered to enable the loopback functionality and establish the connection with the CAN shield:

```
sudo /sbin/ip link set can0 up type can bitrate 500000 loopback on
```

- A cansend and candump file are provided by the SkPang electronics. http://www.skpang.co.uk/dl/can-test_pi2.zip. These files are can be used to send and receive the data over can respectively. To send the message following commands are used in terminal:

```
./cansend can0 7DF#1122334455667788
```

- In a separate terminal window used following command:

```
./candump can0 -n1
```

- One should receive the data same sent using cansend command in candump terminal window as:

```
./candump can0 -n1
can0 7DF [8] 11 22 33 44 55 66 77 88
```

- If the received data is same it means that the CAN shield and Raspberry Pi are interacting correctly.
- An important note is that both the files candump and cansend should be in current directory.

5.1.2 Black Box and Car Interface Testing

- In this testing, the speed fetched from the car is cross verified to make sure that the black box is interacting with the car. The Black Box and the car is interfaced using an OBD-ii to DB9 connector. A screen and keyboard are also used. Raspberry pi is powered from a car via OBD-ii to DB9 connector, whereas a separate power source(battery bank) is used for the screen. A Python script given below is used to fetch the data. The script is developed using open-source examples.

```
import RPi.GPIO as GPIO
import can
import time
import os
import queue
from threading import Thread

led = 22
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(led,GPIO.OUT)
GPIO.output(led,True)

ENGINE_RPM          = 0x0C
VEHICLE_SPEED       = 0x0D

PID_REQUEST         = 0x7DF
PID_REPLY           = 0x7E8

outfile = open('log.txt','w')


print('\n\rStart Reading CAN Data')
print('Connecting to CAN0....')

#Connect to can0 interface at 500kbps
os.system("sudo /sbin/ip link set can0 up type can bitrate 500000")
time.sleep(0.1)
print('Ready')

try:
    bus = can.interface.Bus(channel='can0', bustype='socketcan_native')
```

```
except OSError:
    print('Cannot connect PiCAN board.')
    GPIO.output(led,False)
    exit()

def can_rx_task():
    while True:
        message = bus.recv()
        if message.arbitration_id == PID_REPLY:
            q.put(message)

def can_tx_task():
    while True:

        GPIO.output(led,True)

        # Sent a Engine RPM request
        msg =
         ↪  can.Message(arbitration_id=PID_REQUEST,data=[0x02,0x01,ENGINE_RPM,0x00,0x00,0x00,0x00,0x00]
        bus.send(msg)
        time.sleep(0.001)

        # Sent a Vehicle speed  request
        msg =
         ↪  can.Message(arbitration_id=PID_REQUEST,data=[0x02,0x01,VEHICLE_SPEED,0x00,0x00,0x00,0x00,0x
        bus.send(msg)
        time.sleep(0.001)

q = queue.Queue()
rx = Thread(target = can_rx_task)
rx.start()
tx = Thread(target = can_tx_task)
tx.start()

temperature = 0
rpm = 0
speed = 0
throttle = 0
c = ''
count = 0

# Main
try:
    while True:
        for i in range(2):
            while(q.empty() == True):
                pass
            message = q.get()
            c = '{0:f},{1:d},'.format(message.timestamp,count)

            if message.arbitration_id == PID_REPLY and message.data[2] == ENGINE_RPM:
                rpm = round(((message.data[3]*256) + message.data[4])/4)

            if message.arbitration_id == PID_REPLY and message.data[2] ==
             ↪  VEHICLE_SPEED:
                speed = round(message.data[3]*0.621)

        c += '{0:d},{1:d}'.format(rpm,speed)
        print('\r {} '.format(c))
        print(c,file = outfile)
        count += 1

except KeyboardInterrupt:
    GPIO.output(led,False)
    outfile.close()
    os.system("sudo /sbin/ip link set can0 down")
    print('\n\rStop Operation')
```

- This script will just fetch engine rpm and vehicle speed at a sample rate of 3 readings per second and store it in the log file along with the time stamp and reading count number. The output of the script did match the speed on the vehicle cluster.

5.1.3 Bluetooth Module Testing

- In order to test the Bluetooth module, both the boards are paired and trusted. Bluetooth, in order to communicate with the device, should have a profile installed on it. For this project, the serial port profile is used. The pairing and serial port profile installation are discussed in detailed in Appendix A I. Stand-Alone testing is done by connecting the boards via Bluetooth and sending a message from the client board to the server board. The sample code for client-server communication was used from the PyBluez Bluetooth module. The sample codes are [13]:

```
#rfcomm-server.py

from Bluetooth import *

port = 1

server_sock=BluetoothSocket( RFCOMM )
server_sock.bind(("",port))
server_sock.listen(1)

client_sock, client_info = server_sock.accept()

print "Accepted connection from ", client_info
data = client_sock.recv(1024)

print "received [%s]" % data

client_sock.close()
server_sock.close()

#rfcomm-client.py

from Bluetooth import *

server_address = "01:23:45:67:89:AB"

port = 1

sock=BluetoothSocket( RFCOMM )

sock.connect((server_address, port))

sock.send("hello!!")

sock.close()
```

5.2 IoT Device Script Testing

AWS provides with Python packages for cloud infrastructure for Raspberry pi. Excellent documentation with example is also provided in these documentation.

5.2.1 AWS IoT Core Interface

- For this testing AWS IoT Core message reception is configured on the AWS website and then MQTT protocol configuration using AWS IoT SDK on the Raspberry pi.
- For AWS IoT Core and Raspberry pi communication testing following code [16]:

```
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient
import logging
import time
import json

host = "YOUR-THING-END-POINT"
certPath = "PATAH-OF-THE-KEY-CERT"
clientId = "IOT-DEVICE"
topic = "Vehicle-Data"

# Init AWSIoTMQTTClient
myAWSIoTMQTTClient = None
myAWSIoTMQTTClient = AWSIoTMQTTClient(clientId)
myAWSIoTMQTTClient.configureEndpoint(host, 8883)
myAWSIoTMQTTClient.configureCredentials("{}aws-root-cert.pem".format(certPath),
 ↪  "{}private-key.pem.key".format(certPath), "{}iot-cert.pem.crt".format(certPath))

# AWSIoTMQTTClient connection configuration
myAWSIoTMQTTClient.configureAutoReconnectBackoffTime(1, 32, 20)
myAWSIoTMQTTClient.configureOfflinePublishQueueing(-1)  # Infinite offline Publish
 ↪  queueing
myAWSIoTMQTTClient.configureDrainingFrequency(2)  # Draining: 2 Hz
myAWSIoTMQTTClient.configureConnectDisconnectTimeout(10)  # 10 sec
myAWSIoTMQTTClient.configureMQTTOperationTimeout(5)  # 5 sec
myAWSIoTMQTTClient.connect()

# Publish to the same topic in a loop forever
Count = 0
while True:
    message = {}
    message['message'] = "YOUR-MESSAGE"
    message['sequence'] = Count
    msgjson = json.dumps(message)
    myAWSIoTMQTTClient.publish(topic, msgjson, 1)
    print('topic published %s: %s\n' % (topic, msgjson))
    loopCount += 1
    time.sleep(10)
myAWSIoTMQTTClient.disconnect()
```

- Flow of this code is discussed in chapter 3 section 3.3.2. AWS IoT Configuration will be discussed in the appendix A I.

5.2.2 AWS S3 Core

- The file connection to S3 bucket and file download we done separately using the following script:

```
        bucket_name = "BUCKET_NAME"
    aws_access_key_id = os.getenv("ACCESS_KEY")
    aws_access_secret_key = os.getenv("SECRET_KEY")

    conn = boto.s3.connect_to_region('us-east-1',
       aws_access_key_id = aws_access_key_id,
       aws_secret_access_key = aws_access_secret_key,
       is_secure = True,
       calling_format = 'boto.s3.connection.OrdinaryCallingFormat'
       )

    bucket = conn.get_bucket(bucket_name)

    bucket_list = bucket.list()

    cnt = 0

    for l in bucket_list:
        key_string = str(l.key)
        s3_path = local_cloud_location + key_string
        try:
            cnt += 1
            if (data_print_flag == 1):
                print ("Current File is ", s3_path)
            l.get_contents_to_filename(s3_path)
        except (OSError,S3ResponseError) as e:
            pass
            if not os.path.exists(s3_path):
                try:
```

```
        os.makedirs(s3_path)
    except OSError as exc:
        import errno
        if exc.errno != errno.EEXIST:
            raise
```

5.3 Integration Testing

Once the module separately tested and was working properly. They were integrated one by one on both the black box and IoT device side and tested. Following are the steps in which modules were integrated:

- First, on the black box side, the fetching of the vehicle data and the Bluetooth data transmission was integrated into a single Python script and tested. On the IoT side, only the data reception/client code from Bluetooth was tested without any other functionality.
- On the IoT side first, the client Bluetooth code and the AWS IoT core module was tested with the finished black box script. The received messaged were then observed on the monitor dashboard of the AWS website.
- Once it was concluded that the connection between the IoT device and AWS IoT core is established. The AWS S3 bucket Python script was integrated with the IoT device script and checked if the files pushed to the S3 bucket were able to download.
- As the vehicle was not always available for testing tow mock data generator scripts were developed for Bluetooth data exchange and AWS data push and fetch testing.

Chapter 6

Conclusions

This chapter discusses future work that could be completed as well as the conclusions from this project.

6.1 Project Conclusions

The project was successfully implemented. It addressed the following issues in the current over-speed ticketing system:

- **Lack of Proof to Support Claimant(Sheriff) or Defendant(Driver):** The system implemented duly addresses this issue. Proof of vehicle speed along with a timestamp is provided from the vehicle itself.

- **Wrong Identification of Over-Speeding Vehicle:** The speed record is provided by the vehicle itself. Thus, no wrong implication of the ticket can be done.

- **Covers Corner Cases:** Even if the caught vehicle was over-speeding, the sheriff could see the speed details for the past 20 minutes. This helps him/her make a much more educated decision in identifying if the driver is an aggressive driver or it was just a corner case where the driver was overtaking a truck, changing lane, or passing by the slow vehicle.

- **Court Scenario:** In the current system, both the claimant(Sheriff) and the defendant(Driver) have to be present in the court if the driver decides to defend the over-speeding ticket. In this scenario, as no proof is present, the case becomes more to sheriff's word to drivers word. And the judges, in most cases, assume that sheriff has identified the correct vehicle using a radar gun. In such cases, the black box system implemented here is beneficial. The driver can request the vehicle records from the secure cloud to counterclaim the assigned ticket. In this scenario, the sheriff need not be present in court. Thus, saving the time of the court, sheriff, and driver as if either party is not present, a new date is assigned.
- **Cost advantage:** The system introduced is inexpensive, easy to implement and uses very secure communication protocols.
- **Privacy Concerns:** The driver's data can be shared with the sheriff only with his/her approval via Bluetooth. The shared data is only for the latest 20 mins, and no prior data is shared. The driver's data is not stored locally on the IoT device.

6.2 Future Work

Theres always a room for improvement in any system. After lots of consideration and going through the final expected output following are the two add-in features which can enhance the usability of the project:

- **GPS Module:** A GPS module can provide a much clearer graphic view of the data about the over-speed traffic violation. GPS module provides the speed with a limit of the roads

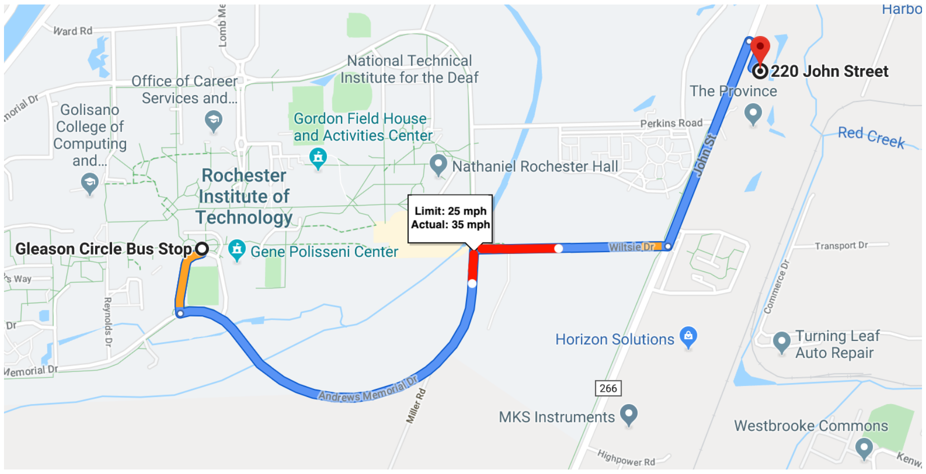


Figure 6.1: GPS View

on which the driver is going. Thus, it can be used to pinpoint the exact location at which speed violation occurred. A visual representation of this is shown in Figure 6.1.

- **Mobile App:** Instead of using an IoT device, a mobile app can be developed to replace it. This makes sure that the sheriff does not carry an additional device and instead uses his cellphone to carry out the task. It also helps to reduce the cost to the device for sheriff.

References

[1] National Highway Traffic Safety Administration. Traffic records program assessment advisory. Technical report, National Highway Traffic Safety Administration, 2018.

[2] Richard Huey Doreen De Leonardis and James Green. National Traffic Speeds Survey III: 2015. Technical report, National Highway Traffic Safety Administration, 2015.

[3] I. Wagner. Total number of licensed drivers in the U.S. in 2017, by state. February 2019. URL: https://www.statista.com/statistics/198029/total-number-of-us-licensed-drivers-by-state/.

[4] Siliang Wu Guohua Wei, Yuxiang Zhou. Detection and localization of high speedmoving targets using a short-range uwbimpulse radar, 2008.

[5] Improving on police radar. *IEEE Spectrum*, 1992.

[6] Mai T. Ngo Vilhelm Gregers-Hansenm. Emi repair in pulse doppler radar, 2008.

[7] Mahendra Mandava, Robert S. Gammenthaler, and Steven F. Hocker. Vehicle Speed Enforcement using Absolute Speed Handheld Lidar, 2018.

[8] S. K. Pang. PiCAN 2 User Manual. Website.

[9] Microchip Technology Inc. *MCP 2551 Stand-Alone CAN Controller with SPI Interface*, 2003-2016.

[10] SK Pang. *PiCAN 2 Board Schematic*, 2015.

[11] Microchip Technology Inc. *MCP 2515 Stand-Alone CAN Controller with SPI Interface*, 2003-2019.

[12] Amazon Web Services, Inc. *AWS IoT: Developer Guide*, 2019.

[13] Albert Huang and Larry Rudolph. *Bluetooth Essentials for Programmers*. Cambridge University Press, 2007.

[14] Wikipedia. OBD-II PIDs. Wikipedia. URL: `https://en.wikipedia.org/wiki/OBD-II_PIDs`.

[15] Yu-Wei Huang, Jieh-Shian Young, Chih-Hung Wu, and Hsing-Jung Li. A Practice Learning of On-board Diagnosis (OBD) Implementations with Embedded Systems. Technical report, American Society for Engineering Education, 2010.

[16] Amazon.com, Inc. *Boto3 Documentation*, 2015.

[17] Ma, Xiaoliang and AndrÃ©asson, Ingmar. Driver reaction time estimation from real car followingdata and application in GM-type model evaluation. Master's thesis, Royal Institute of Technology (KTH), Stockholm 10044, Sweden.

[18] Thomas J. Triggs and Walter G. Harris. REACTION TIME OF DRIVERS TOROAD STIMULI. Master's thesis, MONASH UNIVERSITY, 1982.

[19] John Keenan III. Creating A Wireless OBDII Scanner. Master's thesis, WORCESTER POLYTECHNIC INSTITUTE, 2009.

[20] BOSCH. *CAN Specification*, 1995.

[21] Python. python-can. Website. URL: https://python-can.readthedocs.io/en/master/.

[22] Joaquim Oller Carles Gomez and Josep Paradells. Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Techn. *MDPI*, 2012.

[23] Mrs. Pratibha Singh, Mr. Dipesh Sharma, and Mr. Sonu Agrawal. A Modern Study of Bluetooth WirelessTechnology. In *International Journal of Computer Science, Engineering and Information Technology*, 2011.

[24] Ashwin Makwana Dipa Soni. A Survey on MQTT: A Protocol Of Internet of Things (IoT).

[25] Srijan Manandhar. MQTT BASED COMMUNICATION IN IOT. Master's thesis, Tampere University of Technology, 2017.

Appendix I

Guides

I.1 Bluetooth Pairing

- To pair two Raspberry pi via Bluetooth is to update OS and install the Bluetooth and PyBluez on both the Raspberry by run following commands:

```
sudo apt-get dist-upgrade
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install bluez pi-Bluetooth Python-bluez
```

- Next step is to kill all the process that may be blocking Bluetooth chip using following commands on both the boards:

```
sudo rfkill unblock all
```

- Now in the next few the both the Raspberry pi will be paired with each other. Make sure the Bluetooth is on using following commands on both the boards:

```
Bluetoothctl
[Bluetooth]#  power on
```

- In order to pair the Raspberry pi's, they should be made discoverable and the pairing mode should be turned on using following commands on both the boards:

```
[Bluetooth]# discoverable on
[Bluetooth]# pairable on
```

- The next step is to discover the boards using following commands on both the boards:

```
[Bluetooth]# scan on
```

- The output on screen of both the Raspberry pi should be like this:

```
Discovery started [CHG]
Controller B8:27:EB:56:F3:62 Discovering: yes
[NEW] Device 80:7A:87:0D:A8:68 HTC 691DC1
[NEW] Device 80:7A:09:10:D4:D4 MOTO D5D068
[NEW] Device B8:27:34:25:BH:F5 Raspberrypi
```

- Note down the address <bdaddr> of both the board to be paired and turn off the scanning. Now to pair the boards with each others use following commands:

```
[Bluetooth]# scan off
[Bluetooth]# pair <bdaddr>
```

- Both the boards should pair and a GUI should pop-up message asking to pairing. Click yes to all of options. An error message may be thrown and the devices will disconnect.

- To verify if the both the boards are paired use following commands on both the boards:

```
[CHG] Device <bdaddr> Paired: yes
Pairing successful
[CHG] Device <bdaddr> Trusted: yes
```

- To check if the paired board is trusted type following command on one of the board and you should get the output as below:

```
[Bluetooth]# info <bdaddr>
[Bluetooth]# info B8:27:EB:25:95:F5
Device B8:27:EB:25:95:F5
    Name: Raspberrypi
    Alias: Raspberrypi
    Paired: yes
    Trusted: yes
    Blocked: no
    Connected: no
    LegacyPairing: no
    UUID: Serial Port               (00001101-0000-1000-8000-00805f9b34fb)
    UUID: A/V Remote Control Target (0000110c-0000-1000-8000-00805f9b34fb)
    UUID: A/V Remote Control        (0000110e-0000-1000-8000-00805f9b34fb)
    UUID: PnP Information           (00001200-0000-1000-8000-00805f9b34fb)
    UUID: Generic Access Profile    (00001800-0000-1000-8000-00805f9b34fb)
    UUID: Generic Attribute Profile (00001801-0000-1000-8000-00805f9b34fb)
    Modalias: usb:v1D6Bp0246d0517
```

- If not trusted use following command:

```
[Bluetooth]# trust <bdaddr>
```

I.2 Bluetooth Serial Port Profile

- To add Bluetooth serial port profile edit file ***dbus-org.bluez.service***.

- To edit enter this lines in the terminal:

```
sudo nano /etc/systemd/system/dbus-org.bluez.service
```

- Search for the line: ***ExecStart=/usr/lib/Bluetooth/Bluetoothd***
- Add -C at the end for the line. It should look like this: ***ExecStart=/usr/lib/Bluetooth/Bluetoothd -C***
- Add this line after the above line: ***ExecStartPost=/usr/bin/sdptool add SP***
- Save the file and then reboot the Raspberry pi.
- Open the terminal and type following line:

```
sudo sdptool browse local
```

- After this if you see following output then serial port profile is installed on the Raspberry pi.

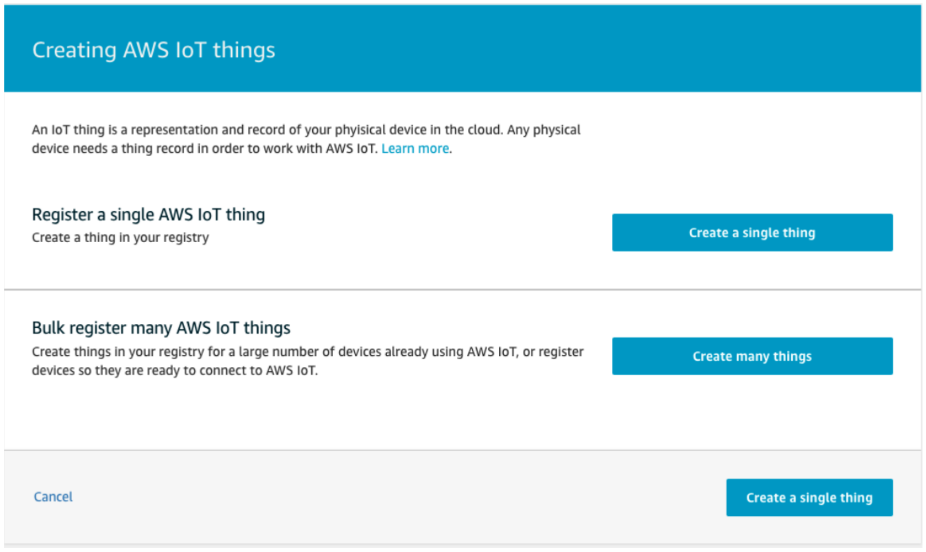
```
Browsing FF:FF:FF:00:00:00 ...
Service Search failed: Invalid argument
Service Name: Serial Port
Service Description: COM Port
Service Provider: BlueZ
Service RecHandle: 0x10001
Service Class ID List:
  "Serial Port" (0x1101)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 1
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
Profile Descriptor List:
  "Serial Port" (0x1101)
    Version: 0x0100
```

- A lot more information will also be present. Please be sure to do this on both the Raspberry pi boards.

I.3 AWS IoT Core

AWS IoT Core is a service provided by AWS. To configure AWS IoT core for message reception from IoT core following are the steps:

- Create a single IoT thing. For each IoT device to be connected to the AWS IoT core a separate IoT thing needs to be created in the AWS IoT Core. The screenshot of the same is shown in figure I.1.


Creating AWS IoT things

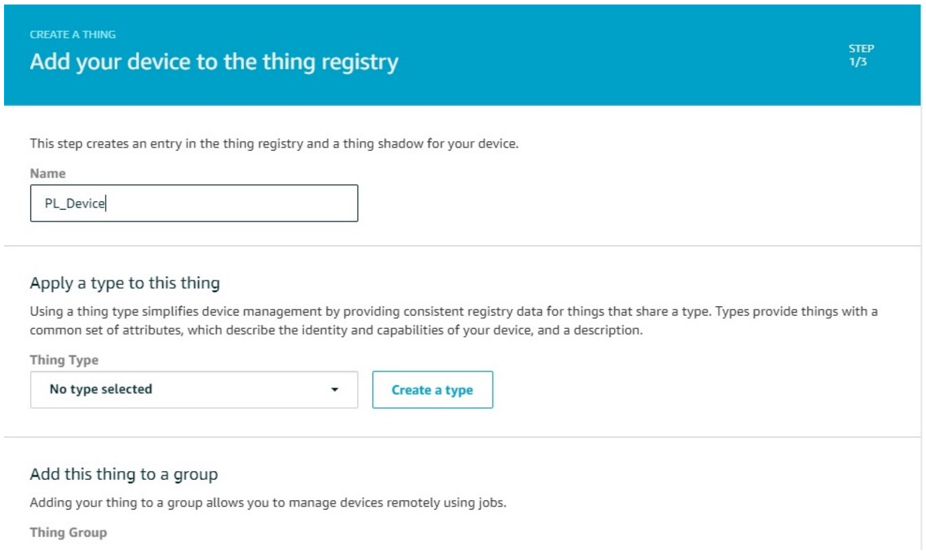
An IoT thing is a representation and record of your phyisical device in the cloud. Any physical device needs a thing record in order to work with AWS IoT. Learn more.

Register a single AWS IoT thing
Create a thing in your registry

Bulk register many AWS IoT things
Create things in your registry for a large number of devices already using AWS IoT, or register devices so they are ready to connect to AWS IoT.

Figure I.1: Create a thing

- After creating a thing add name to the thing you created. In this case it is PL_Device aka police device. After creating a thing you will be asked to create a certificate. You can use the “One-click certificate creation”. By doing AWS IoT will creates the certificates for user. The screenshot of the same is shown in figure I.2.


CREATE A THING

Add your device to the thing registry

STEP 1/3

This step creates an entry in the thing registry and a thing shadow for your device.

Name: PL_Device

Apply a type to this thing

Using a thing type simplifies device management by providing consistent registry data for things that share a type. Types provide things with a common set of attributes, which describe the identity and capabilities of your device, and a description.

Thing Type: No type selected

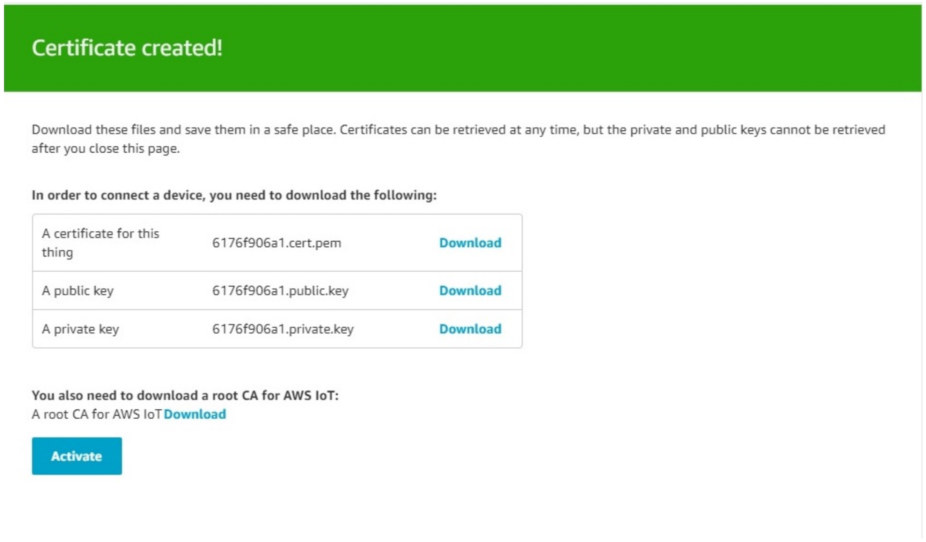
Add this thing to a group

Adding your thing to a group allows you to manage devices remotely using jobs.

Thing Group

Figure I.2: Naming a thing

- After creating the certificate make sure to activate the certificate and download all of them. This keys will be used on both the ends of the MQTT communication i.e. IoT device and IoT Core. The screenshot of the same is shown in figure I.3.



Certificate created!

Download these files and save them in a safe place. Certificates can be retrieved at any time, but the private and public keys cannot be retrieved after you close this page.

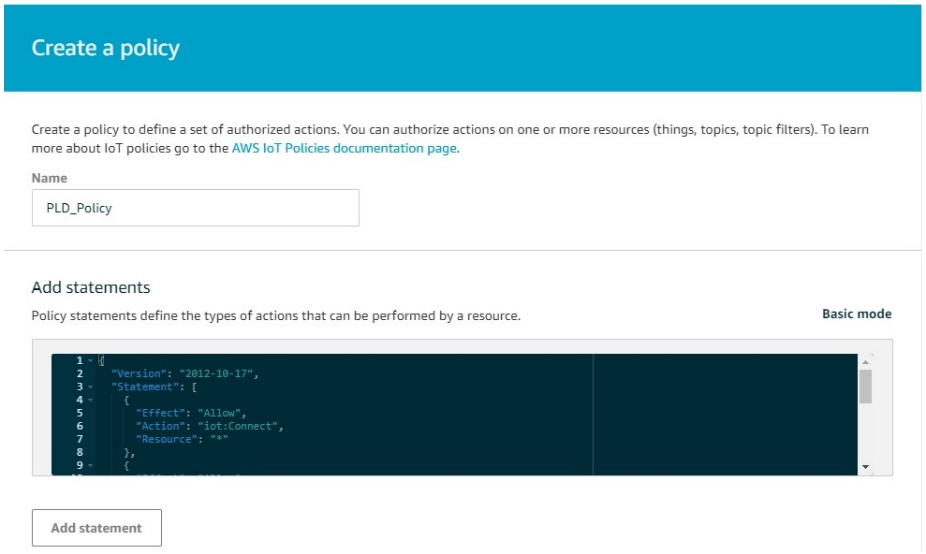
In order to connect a device, you need to download the following:

A certificate for this thing	6176f906a1.cert.pem	Download
A public key	6176f906a1.public.key	Download
A private key	6176f906a1.private.key	Download

You also need to download a root CA for AWS IoT:
A root CA for AWS IoT **Download**

Figure I.3: Create certificate

- After the certificates user should create a policy. The option for this will be present under the secure heading on the left side. On click on create a policy and name you policy. In this case it PD_Policy. The screenshot of the same is shown in figure I.4.



Create a policy

Create a policy to define a set of authorized actions. You can authorize actions on one or more resources (things, topics, topic filters). To learn more about IoT policies go to the AWS IoT Policies documentation page.

Name: PLD_Policy

Add statements

Policy statements define the types of actions that can be performed by a resource.

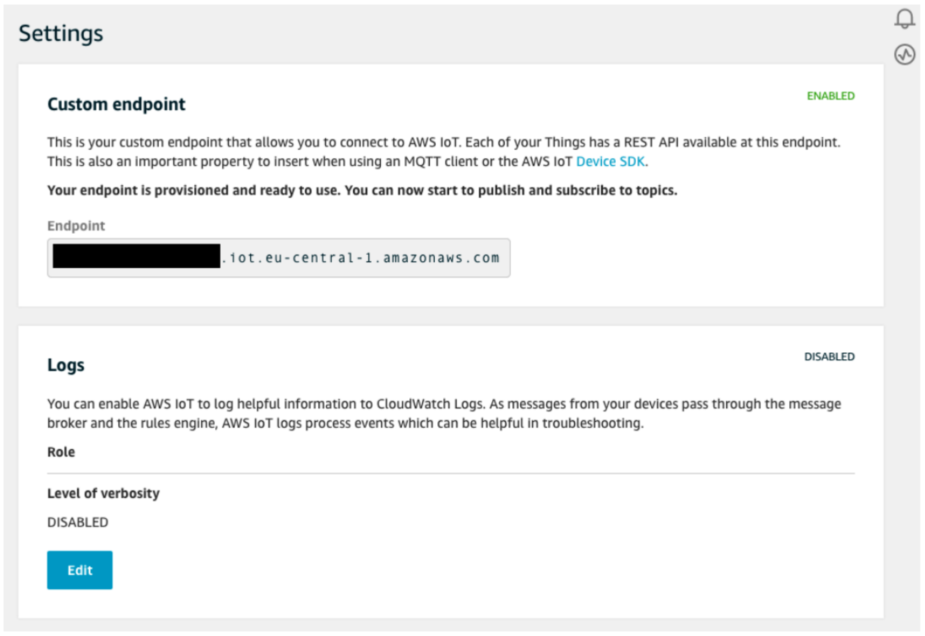
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "*"
    },
    {
```

Figure I.4: Create certificate

- Now in the add statements in the **Add Statements** section to enable the connection, publishing, subscription and reception of message from IoT device.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Subscribe",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Receive",
      "Resource": "*"
    }
  ]
}
```

- Now go to settings and copy the custom end point address and use it for AWS IoT configuration on Raspberry pi. The screenshot of the same is shown in figure I.5.



Settings

Custom endpoint ENABLED

This is your custom endpoint that allows you to connect to AWS IoT. Each of your Things has a REST API available at this endpoint. This is also an important property to insert when using an MQTT client or the AWS IoT Device SDK.

Your endpoint is provisioned and ready to use. You can now start to publish and subscribe to topics.

Endpoint

```
[redacted].iot.eu-central-1.amazonaws.com
```

Logs DISABLED

You can enable AWS IoT to log helpful information to CloudWatch Logs. As messages from your devices pass through the message broker and the rules engine, AWS IoT logs process events which can be helpful in troubleshooting.

Role

Level of verbosity

DISABLED

Figure I.5: End point address

- To check if you are receiving message check the AWS IoT Core dashboard monitor. You can see when the connection was established and number of messages received. The screenshot of the same is shown in figure I.6.

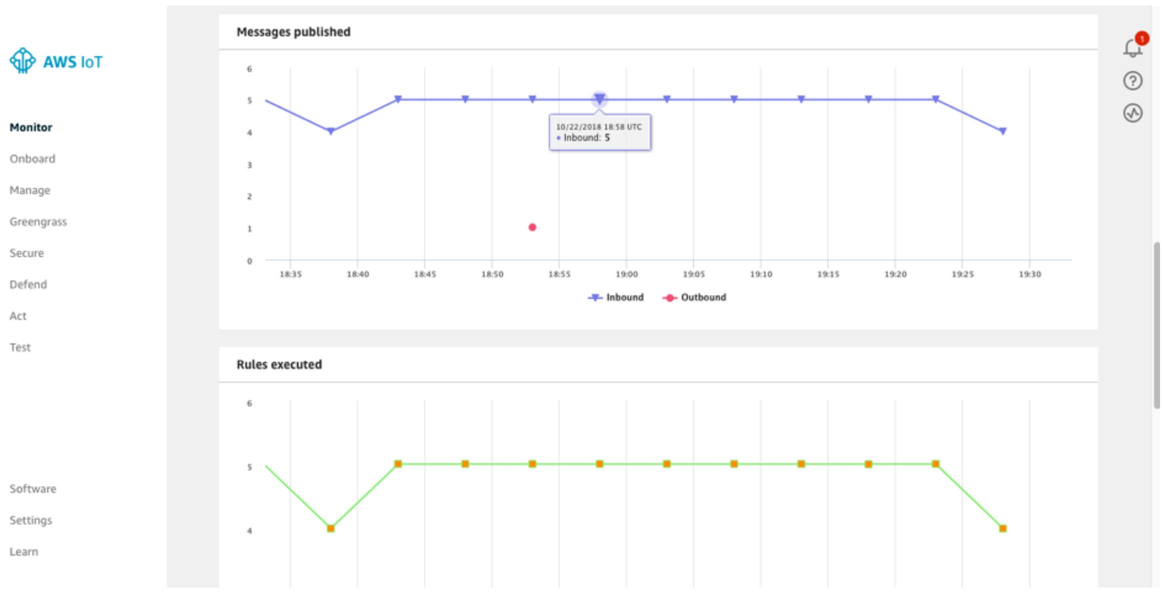
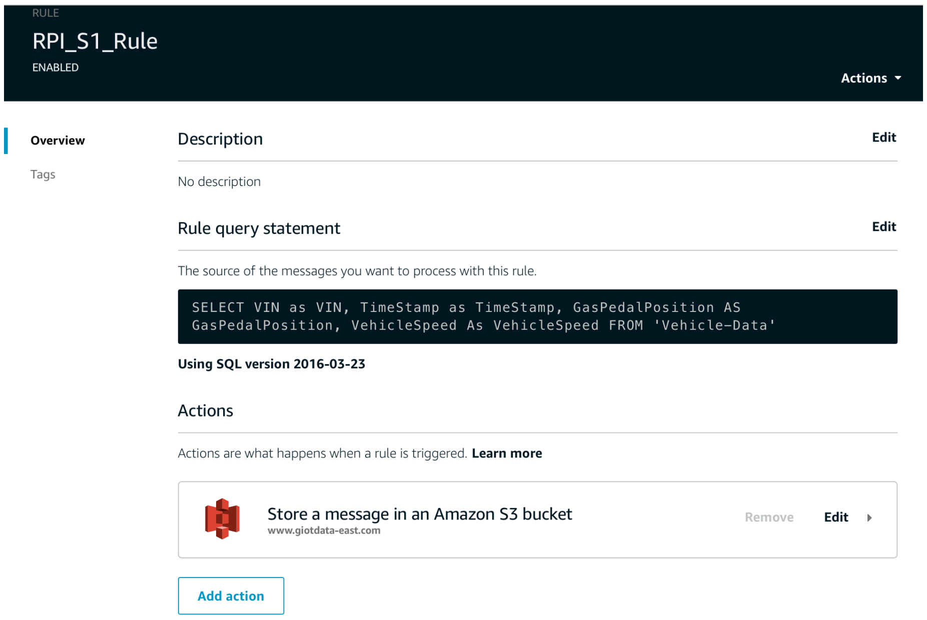


Figure I.6: Monitor on dashboard

I.4 AWS S3 Bucket

To receive the message from AWS IoT Core and configure AWS S3 bucket following steps should be done:

- First thing you should do is to create an action. An action is needed to extract the information from the message. The screenshot of the same is shown in figure I.7.



RPI_S1_Rule
ENABLED

Description
No description

Rule query statement
The source of the messages you want to process with this rule.

```
SELECT VIN as VIN, TimeStamp as TimeStamp, GasPedalPosition AS
GasPedalPosition, VehicleSpeed As VehicleSpeed FROM 'Vehicle-Data'
```

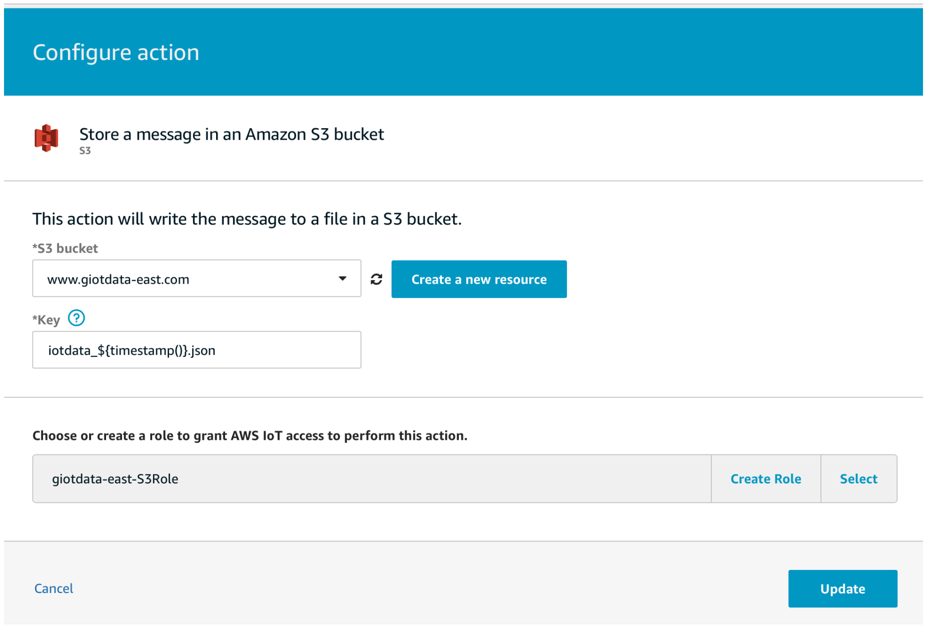
Using SQL version 2016-03-23

Actions
Actions are what happens when a rule is triggered. Learn more

Store a message in an Amazon S3 bucket
www.giotdata-east.com

Figure I.7: Action creation

- Now enter the name of the bucket in this case “www.giotdata-east.com”. Key is the name of the file that will be stored in the S3 bucket. The screenshot of the same is shown in figure I.8.


Configure action

Store a message in an Amazon S3 bucket
S3

This action will write the message to a file in a S3 bucket.

*S3 bucket
www.giotdata-east.com

*Key
iotdata_${timestamp()}.json

Choose or create a role to grant AWS IoT access to perform this action.

giotdata-east-S3Role

Figure I.8: S3 bucket creation

Appendix II

Source Code

II.1 Python Script Related to Black Box

II.1.1 Black Box Script

```
################################################################################
# Author: Gaurav M. Shende
# Description: This script will fetch the vehicle speed and gas pedal position
# and will store it in the 'car_log.txt'. It will then send this data to IoT
# device whenever requested.
################################################################################

import os
import sys
import queue

import can
import time

import RPi.GPIO as GPIO
from threading import Thread
from bluetooth import *

led = 22
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
```

```python
GPIO.setup(led,GPIO.OUT)
GPIO.output(led,True)

gaspp_pid     = 0x49
vspeed_pid    = 0x0D

pid_rq        = 0x7DF
pid_rp        = 0x7E8

gaspp = 0
vspeed = 0

msg_q = None
file_name = 'car_log.txt'

uuid = "94f39d29-7d6d-437d-973b-fba39e49d4ee"
addr = 'B8:27:EB:10:43:0B'

standalone_test_flag = 1
data_display_flag = 0
ble_exc_flag = 0

temp_flag = 0

#Thread to receive vehicle data
def canrx_task():

    global standalone_test_flag
    global ble_exc_flag
    global msg_q

    print("\nIn canrx_task")

    while True:
        if (standalone_test_flag == 0):
            message = bus.recv()
            if message.arbitration_id == pid_rp:
                msg_q.put(message)
        else:
            time.sleep(5)
            ble_exc_flag = 1

#Thread to request vehicle data
def cantx_task():

    global standalone_test_flag
    global ble_exc_flag
    global led

    print("\nIn cantx_task")

    while True:
        if (standalone_test_flag == 0):
            GPIO.output(led,True)

            # Sent a request for gas pedal position
            msg =
             ↪  can.Message(arbitration_id=pid_rq,data=[0x02,0x01,gaspp_pid,0x00,0x00,0x00,0x00,0x00],e
            bus.send(msg)
            time.sleep(0.4)

            #Sent a request for vehicle speed
            msg =
             ↪  can.Message(arbitration_id=pid_rq,data=[0x02,0x01,vspeed_pid,0x00,0x00,0x00,0x00,0x00],
            bus.send(msg)
            time.sleep(0.4)
        else:
            time.sleep(5)

#Function to find near by bluetooth devices and send the latest vehicle data
def bletx_task():

    global uuid
    global addr

    print("Searching for the near by Sheriff's device")

    service_matches = find_service( uuid = uuid, address = addr )

    if len(service_matches) == 0:
        print("Couldn't find the Sheriff's device")
        return None

    first_match = service_matches[0]
    port = first_match["port"]
    name = first_match["name"]
    host = first_match["host"]

    print("Found Sheriff's device: \"%s\" Device Address: %s" % (name, host))
    print("Connecting to the Sheriff's device")

    # Create the client socket
    ble_sock=BluetoothSocket( RFCOMM )
    ble_sock.connect((host, port))

    print("Sending Data to the Sheriff's device")

    i = 0

    for ln in reversed(list(open("car_log.txt"))):
        if (data_display_flag == 1):
            print(ln)
        i+=1
        ble_sock.send(ln)

        if(i == 1220):
            break

    print("Total number of readings sent:", i)

    ble_sock.close()

#Main thread
def main():

    global gaspp_pid
    global vspeed_pid

    global gaspp
    global vspeed

    global file_name
    global standalone_test_flag

    global msg_q

    if (standalone_test_flag == 0):
        print('Connecting to the Vehicle....')

        #Bring up can0 interface at 500kbps
        os.system("sudo /sbin/ip link set can0 up type can bitrate 500000")
        time.sleep(0.1)

        try:
            bus = can.interface.Bus(channel='can0', bustype='socketcan_native')
            print('Connection to the Vehicle established')
        except OSError:
            pprint('Connection to the Vehicle failed')
            GPIO.output(led,False)
            exit()

    msg_q = queue.Queue()

    canrx = Thread(target = canrx_task)
    canrx.start()

    cantx = Thread(target = cantx_task)
    cantx.start()

    try:
        while True:
            if (standalone_test_flag == 0):
                for i in range(2):
                    while(msg_q.empty() == True):   # Wait until there is a message
                        pass
                    message = msg_q.get()

                    if message.arbitration_id == pid_rp and message.data[2] ==
                     ↪  gaspp_pid:
                        gaspp = round(message.data[3]*0.392*100)

                    if message.arbitration_id == pid_rp and message.data[2] ==
                     ↪  vspeed_pid:
                        vspeed = round(message.data[3]*0.621)

                if (vspeed != 0):
                    data = {
                    'VIN' : '1ZVHT84N265118896',
                    'TimeStamp': message.timestamp,
                    'GasPedalPosition': gaspp,
                    'VehicleSpeed': vspeed
                    }

                    with open(file_name, 'a') as outfile:
                        json.dump(data, outfile)
                        outfile.write("\n")
                else:
                    time.sleep(10)
                    if(vspeed != 0):
                        bletx_task()
            else:
                sts = 0
                if (ble_exc_flag == 1):
                    sts = bletx_task()
                    time.sleep(5)

    except KeyboardInterrupt:
        #Catch keyboard interrupt
        GPIO.output(led,False)
        if (standalone_test_flag == 0):
            outfile.close()     # Close logger file
            os.system("sudo /sbin/ip link set can0 down")
        print('\n\rKeyboard interrtupt')

if __name__=="__main__":
    main()
```

II.1.2 Log Data Generator Script

```
##################################################################################
# Author: Gaurav M. Shende
# Description: This script will create a sample mock data from OBD-ii port and
# store it in to 'car_log.txt'. This helps to test the 'car_to_iot_v1.py'
# without actually connecting black box to the car.
##################################################################################

import json
import random
import os

import shutil


if os.path.exists('car_log.txt'):
    os.remove('car_log.txt')
    print("File car_lot.txt already exist")
else:
    print("New file car_lot.txt created")

acpad = 0
vspeed = 0
temp = 1568463179
file_name = 'car_log.txt'

for i in range(1300):

    temp = temp + 1
    if(0 <= i < 120):
        vspeed = 42
        acpad = 47
    elif(121 < i < 240):
        vspeed = 52
        acpad = 57
    elif(241 < i < 480):
        vspeed = 62
        acpad = 67
    elif(481 < i < 600):
        vspeed = 72
        acpad = 77
    elif(601 < i < 720):
        vspeed = 82
        acpad = 87
    elif(721 < i < 840):
        vspeed = 78
        acpad = 77
    elif(841 < i < 960):
        vspeed = 42
        acpad = 47
    elif(961 < i < 1200):
        vspeed = 41
        acpad = 43

    data = {
    'VIN' : '1ZVHT84N265118896',
    'TimeStamp': temp,
    'GasPedalPosition': acpad,
    'VehicleSpeed': vspeed
    }

    with open(file_name, 'a+') as outfile:
        json.dump(data, outfile)
        outfile.write("\n")
```

II.2 Python Script Related to IoT Device

II.2.1 IoT Device Script

```
################################################################################
# Author: Gaurav M. Shende
# Description: This script will fetch the vehicle data from the black box. Then
# send it to the S3 bucket and again download it and then plot it.
################################################################################

import sys
import os
import boto
import json
import time

from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient
from bluetooth import *
from boto.s3.key import Key
from boto.exception import S3ResponseError

#Used to store car readings
car_data = []

```

```
#AWS related vairables
aws_host = "YOUR_HOST_NAME"
cert_path = "/home/pi/Desktop/iot_to_cloud/key_cert/"
client_id = "RaspberryPi"
vdata_topic = "Vehicle-Data"

my_aws_client = None

uuid = "94f39d29-7d6d-437d-973b-fba39e49d4ee"

local_cloud_location = "/home/pi/Desktop/iot_to_cloud/data_from_cloud/"

#Variable used for debugging
data_print_flag = 0

#Function to fetch data from S3 bucket
def fetch_s3_data():
    bucket_name = "BUCKET_NAME"
    aws_access_key_id = os.getenv("ACCESS_KEY")
    aws_access_secret_key = os.getenv("SECRET_KEY")

    conn = boto.s3.connect_to_region('us-east-1',
       aws_access_key_id = aws_access_key_id,
       aws_secret_access_key = aws_access_secret_key,
       is_secure = True,
       calling_format = 'boto.s3.connection.OrdinaryCallingFormat'
       )

    bucket = conn.get_bucket(bucket_name)

    bucket_list = bucket.list()

    cnt = 0

    for l in bucket_list:
        key_string = str(l.key)
        s3_path = local_cloud_location + key_string
        try:
            cnt += 1
            if (data_print_flag == 1):
                print ("Current File is ", s3_path)
            l.get_contents_to_filename(s3_path)
        except (OSError,S3ResponseError) as e:
            pass
            if not os.path.exists(s3_path):
                try:
                    os.makedirs(s3_path)
                except OSError as exc:
                    import errno
                    if exc.errno != errno.EEXIST:
                        raise


    print("Total number of files downloaded: ", cnt)

#Function to fetch data from black box over bluetooth
def ble_fetch_data():

    global uuid
    global car_data

    ble_srv_sock = BluetoothSocket( RFCOMM )
    ble_srv_sock.bind(("",PORT_ANY))
    ble_srv_sock.listen(1)

    port = ble_srv_sock.getsockname()[1]


    advertise_service( ble_srv_sock, "Sheriffs_Device",
                       service_id = uuid,
                       service_classes = [ uuid, SERIAL_PORT_CLASS ],
                       profiles = [ SERIAL_PORT_PROFILE ])

    print("Waiting for connection on RFCOMM channel %d" % port)

    ble_clnt_sock, ble_clnt_info = ble_srv_sock.accept()
    print("Accepted connection from ", ble_clnt_info)

    co_data = []
    n_data = []

    i = 0

    try:
        while True:
            data_holder = ble_clnt_sock.recv(1024)
            data_holder = data_holder.decode('utf8').replace("'", '"')

            l = len(data_holder)

            if (l > 98):
                co_data.append(data_holder)
            elif (l == 98):
                car_data.append(data_holder)

                if len(data_holder) == 0:
                    break
                i += 1
            else:
                n_data.append(data_holder)

    except IOError:
        pass

    ble_clnt_sock.close()
    ble_srv_sock.close()

    l = len(car_data)
    for m in range(l):
        if (data_print_flag == 1):
            print("Data received from Car: %s" % car_data[m])

    if (data_print_flag == 1):
        print(len(co_data))
        print("Co", co_data)

        print(len(n_data))
        print("N", n_data)

        print("i",i)

    temp = []
    l1 = len(co_data)
    for j in range(l1):
        temp = co_data[j].split("\n")
        l2 = len(temp)
        for k in range(l2):
            if (temp[k] != ""):
                if (data_print_flag == 1):
                    print("temp",k, temp[k])
                car_data.append(temp[k])
                i += 1

    if(i < 1200):
        print("Reinitiate data transfer")
        sys.exit(0)
    else:
        print("Data reception from car done!")
        print("Total number of readings received: 1200")
        print("BLE disconnected")

#Function to configure AWS S3 bucket on IoT device
def aws_config():

    global aws_host
    global cert_path
    global client_id
    global my_aws_client

    my_aws_client = AWSIoTMQTTClient(client_id)
    my_aws_client.configureEndpoint(aws_host, 8883)
    my_aws_client.configureCredentials("{}AmazonRootCA1.pem".format(cert_path), \

                                          ↪  "{}38ea5c7a2c-private.pem.key".format(cert_path),
                                          ↪  \

                                          ↪  "{}38ea5c7a2c-certificate.pem.crt".format(cert_path))

    # AWSIoTMQTTClient connection configuration
    my_aws_client.configureAutoReconnectBackoffTime(1, 32, 20)
    my_aws_client.configureOfflinePublishQueueing(-1)  # Infinite offline Publish
     ↪  queueing
    my_aws_client.configureDrainingFrequency(2)  # Draining: 2 Hz
    my_aws_client.configureConnectDisconnectTimeout(10)  # 10 sec
    my_aws_client.configureMQTTOperationTimeout(5)  # 5 sec


temp = None

#Function to send data from IoT device to S3 bucket
def send_data_to_aws():

    global vdata_topic
    global my_aws_client
    global temp

    my_aws_client.connect()

    num_car_data = len(car_data)

    cnt = 0
    for i in range(1200):
        try:
            cnt += 1
            temp = json.loads(car_data[i])

            vdata = {}
            vdata['VIN'] = temp['VIN']
            vdata['TimeStamp'] = temp['TimeStamp']
            vdata['GasPedalPosition'] = temp['GasPedalPosition']
            vdata['VehicleSpeed'] = temp['VehicleSpeed']

            vdata_json = json.dumps(vdata)

            my_aws_client.publish(vdata_topic, vdata_json, 1)

            if (data_print_flag == 1):
                print('Data sent to Cloud: %s\n' % vdata_json)

#              time.sleep(0.1)

        except ValueError:
            #print("Error for reading: ",temp)
            pass
    my_aws_client.disconnect()
    print("Total number of readings sent to cloud: ", cnt)

#Function to clear user data
def clr_usr_data():
    global car_data

    car_data.clear()

def main():

    aws_config()
    ble_fetch_data()
    send_data_to_aws()
    clr_usr_data()
    fetch_s3_data()
    os.system("sudo Python3 /home/pi/Desktop/iot_to_cloud/data_plot_v4.py")

if __name__== "__main__":
    main()
```

II.2.2 Data Plotting Script

```
################################################################################
# Author: Gaurav M. Shende
# Description: This script will plot the vehicle data vs time
################################################################################

import os
import datetime
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
from matplotlib.ticker import (MultipleLocator, FormatStrFormatter,
                               AutoMinorLocator)

#Global variables related to files
```

```
file_path = []
file_list = []
num_file = []

#Global Variables related to vehicle parameters
vin_num = []
tstamp = []
acpad_d = []
vspeed = []

#Global Variables related to plot
tstamp_p = []
acpad_d_p = []
vspeed_p = []

#Scaling variable
scale_param = 1    #Make this variable 1 for 120 sample points and 0 for 1200 for a
 ↪  plot
x_maj_loc = 0
x_min_loc = 0

#Function to fetch the data from the files
def fetch_data():
    global file_path
    global file_list
    global num_file

    global vin_num
    global tstamp
    global acpad_d
    global vspeed

    for i in range(num_file):
        file_list[i] = file_path + "/" + file_list[i]
        with open(file_list[i],'r') as json_file:
            data = (json_file.readline()).split(',')

            #Extract vin infromation
            temp = data[0].split(':')
            vin_num.append(temp[1])

            #Extract time stamp
            temp = data[1].split(':')
            temp[1] = temp[1].replace('{','')
            tstamp.append(temp[1])
            tstamp = list(map(float, tstamp))

            #Extract accelarator pedal position
            temp = data[2].split(':')
```

```python
            acpad_d.append(temp[1])
            acpad_d = list(map(int, acpad_d))

            #Extract vehicle speed infromation
            temp = data[3].split(':')
            temp[1] = temp[1].replace('}','')
            vspeed.append(temp[1])
            vspeed = list(map(int, vspeed))

#Function to average the fetch data for 10 points
def avg_data():

    global tstamp
    global acpad_d
    global vspeed

    global tstamp_p
    global acpad_d_p
    global vspeed_p

    global scale_param
    global x_maj_loc
    global x_min_loc

    #Scale the data i.e. reduce the sample data points by averaging
    #the speed for 10 seconds
    if(scale_param == 1):
        for i in range(0,num_file,10):
            tstamp_p.append(tstamp[i])

        temp = []
        for i in range(0,num_file,10):
            for j in range(10):
                temp.append(acpad_d[i+j])
            acpad_d_p.append(round(sum(temp)/10))
            temp.clear()

        temp = []
        for i in range(0,num_file,10):
            for j in range(10):
                temp.append(vspeed[i+j])
            vspeed_p.append(round(sum(temp)/10))
            temp.clear()
        x_maj_loc = 6
        x_min_loc = 1
    #Do not scale
    else:
        for i in range(0,num_file):
            tstamp_p.append(tstamp[i])
            acpad_d_p.append(acpad_d[i])
            vspeed_p.append(vspeed[i])
        x_maj_loc = 60
        x_min_loc = 10

#Function to plot the data
def plot_data():

    global vin_num
    global tstamp_p
    global acpad_d_p
    global vspeed_p

    t_len = len(tstamp_p)

    for i in range(t_len):
        tstamp_p[i] = datetime.datetime.fromtimestamp(tstamp_p[i]).strftime('%H:%M:%S')

    tdate = datetime.datetime.today()

    fig,ax = plt.subplots()

    #Title, and label for x and y axis
    plt.title("" + r"$\bf{" + "VIN:" + "}$"  + vin_num[0] + "\n" + r"$\bf{" + "Date: "
    ↪  + "}$" + str(tdate.month) + "-" + \
              str(tdate.day) + "-" + str(tdate.year) + "    " + r"$\bf{" + "Time: " +
              ↪  "}$" + \
              tstamp_p[0] + " to " + tstamp_p[t_len-1], fontsize = 12)

    plt.xlabel('Time Stamp')
    plt.ylabel('Vehicle Speed')

    #Convert string to integer in order to plot the graph
    for i in range(num_file):
        acpad_d[i] = int(acpad_d[i])
        vspeed[i] = int(vspeed[i])

    #Plot the graph for vehicle speed and accelerator pedal position
    ax.plot(tstamp_p,vspeed_p, label = 'Vehicle Speed', color = 'brown', alpha = 0.6)
    ax.plot(tstamp_p,acpad_d_p, label = 'Gas Pedal Position', color = 'blue', alpha =
    ↪  0.6)

    #Set parameters for the x and y axis label
    ax.tick_params(axis="x", labelsize=5, labelrotation=30, labelcolor="black")
    ax.tick_params(axis="y", labelsize=5, labelrotation=0, labelcolor="black")

    #Set the resolution for x and y axis
    ax.xaxis.set_major_locator(MultipleLocator(x_maj_loc))
    ax.xaxis.set_minor_locator(MultipleLocator(x_min_loc))

    ax.yaxis.set_major_locator(MultipleLocator(5))
    ax.yaxis.set_minor_locator(MultipleLocator(1))

    #Set the color code for different speed range
    plt.axhspan(35, 55, color='green', alpha=0.3)
    plt.axhspan(55, 65, color='violet', alpha=0.3)
    plt.axhspan(65, 75, color='yellow', alpha=0.3)
    plt.axhspan(75, 85, color='blue', alpha=0.3)
    plt.axhspan(85, 100, color='red', alpha=0.3)
    plt.axhspan(100, 120, color='brown', alpha=0.3)

    #Write speed limit text on graph
    plt.text(0,40,"Speed 35-55 mph", color = 'black', alpha=0.2, fontsize = 20)
    plt.text(0,57,"Speed 56-65 mph", color = 'black', alpha=0.2, fontsize = 20)
    plt.text(0,67,"Speed 66-75 mph", color = 'black', alpha=0.2, fontsize = 20)
    plt.text(0,77,"Speed 76-85 mph", color = 'black', alpha=0.2, fontsize = 20)
    plt.text(0,89,"Speed 86-100 mph", color = 'black', alpha=0.2, fontsize = 20)
    plt.text(0,106.5,"Speed 101-120 mph", color = 'black', alpha=0.2, fontsize = 20)

    #Set the remaining parameters and save the plot
    fig1 = plt.gcf()
    plt.legend(loc='upper center', bbox_to_anchor=(0.9, 1), shadow=True, ncol=1)#,
     ↪  bbox_transform=fig.transFigure)
    plt.grid(linestyle='dotted')
    plt.show()
    plt.draw()
    fig1.savefig('vehicle_data.png', dpi=300)

    print("Plot saved in the current directory... \n")

#Main Function
def main():

    global file_path
    global file_list
    global num_file

    print("Python script started... \n")

    #Get the list of all .json files
    file_path = "/home/pi/Desktop/iot_to_cloud/data_from_cloud"
    file_list = os.listdir(file_path)
    num_file = len(file_list)
    if (num_file < 1200):
        print("Not enough data fetched from cloud")
    else:
        num_file = 1200
    #Sort the files by name
    file_list.sort()

    #Print the file names
    #for i in range(num_file):
    #    print(file_list[i])

    print("\nTotal Number of files present in the folder: " + str(num_file))

    fetch_data()
    avg_data()
    plot_data()

    print("Done... \n")

if __name__== "__main__" :
    main()
```

II.2.3 Mock JSON files Generator Script

```
################################################################################
# Author: Gaurav M. Shende
# Description: This script will create mock data to send to S3 bucket
################################################################################

import json
import random
import os

import shutil

file_path = "/Users/gauravshende/Google
↪ Drive/Project/Git_Repo/gms1682_grad_ppr/Code/Data_Presentation/Json_files"

for root, dirs, files in os.walk(file_path):
    for f in files:
        os.unlink(os.path.join(root, f))
    for d in dirs:
        shutil.rmtree(os.path.join(root, d))

temp = 1568463179
fname = temp
#random.seed(0)

vspeed = 0
acpad = 0
```

```

for i in range(1200):
    temp = temp + 1
    fname = fname + 1
    if(0 < i < 120):
        vspeed = 42
        acpad = 47
    elif(121 < i < 240):
        vspeed = 52
        acpad = 57
    elif(241 < i < 480):
        vspeed = 62
        acpad = 67
    elif(481 < i < 600):
        vspeed = 72
        acpad = 77
    elif(601 < i < 720):
        vspeed = 82
        acpad = 87
    elif(721 < i < 840):
        vspeed = 78
        acpad = 77
    elif(841 < i < 960):
        vspeed = 42
        acpad = 47
    elif(961 < i < 1200):
        vspeed = 41
        acpad = 43

    data = {}
    data = {
    'VIN' : '1ZVHT84N265118896',
    'TimeStamp': temp,
    'Accelerator Pedal Position': acpad,
    'Vehicle Speed': vspeed
    }
    filename = file_path + "/giotdata_"+str(fname)+".json"
    with open(filename, 'w') as outfile:
        json.dump(data, outfile)
```