

**ELECTRIC VEHICLE REALTIME MONITORING SYSTEM USING
MACHINE TO MACHINE COMMUNICATION**

Johann Mathias

Bachelor of Engineering
Electronic Engineering



Department of Electronic Engineering
Macquarie University

November 7, 2016

Supervisor: Professor Graham Town



ACKNOWLEDGMENTS

I would like to acknowledge my girlfriend and family for all their support and encouragement throughout this project. I would also like to acknowledge my supervisor Professor Graham Town for his guidance and flexibility throughout this thesis project at Macquarie University.



STATEMENT OF CANDIDATE

I, Johann, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Electronic Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment at any other academic institution.

Student's Name: Johann Mathias

Student's Signature:

A handwritten signature in blue ink, appearing to read 'J Mathias', with a horizontal line underneath.

Date: 07/11/2016



ABSTRACT

Vehicles are cemented in everyday life with most people relying on cars as a means of transport on a daily basis. Even though cars are so vital in everyday functioning their data neglects to be collected and utilised. Vehicle monitoring systems provide access to a range of data that enables logistics to be run on the collected data. This is even more prevalent for electronic vehicles as they possess a greater number of metrics that can be monitored and analysed. Due to high costs and increasingly outdated hardware, vehicle monitoring systems are not common place in most vehicles. The monitoring system in this thesis was designed to meet system requirements, the most crucial being that the system must be a realtime system as well as a machine to machine communication system. This thesis aims to design and develop a vehicle monitoring system prototype targeted at electric vehicles. It also aims to develop a monitoring system that is more technologically up to date and more cost effective than any vehicle monitoring system commercially available.



Contents

Acknowledgments	iii
Abstract	vii
Table of Contents	ix
List of Figures	xi
List of Tables	xiii
Abbreviations	xv
1 Introduction	1
1.1 Electronic Vehicle Monitoring System	1
1.1.1 Monitoring System Requirements	2
1.1.2 My Contribution	2
2 Background and Related Work	3
2.1 Machine to Machine Technology	3
2.2 Internet of things	3
2.3 Wireless data transfer	6
2.4 Collected Data Usages	7
2.5 Python Programming	7
2.5.1 PySerial	7
2.5.2 Networking Python	8
3 Monitoring System Design	11
3.1 Introduction	11
3.2 Related Work	11
3.2.1 ELM327	11
3.2.2 CANlogger2000	12
3.2.3 FleetCarma	12
3.2.4 FLEX	13
3.2.5 Comparison of Existing Hardware	13

3.3	System Model	14
3.3.1	Mitsubishi iMiev	16
4	Results and Future Work	17
4.1	Development	17
4.1.1	Raspberry Pi 4G Internet Connection	17
4.1.2	Hardware Testing	19
4.1.3	Python Coding	20
4.1.4	Shell Scripting	22
4.2	Results	23
4.3	Future Work	25
A	Consultation Form	31
A.1	Overview	31
A.2	Consultation Attendance Form	32
B	Client Python Code	33
B.1	Overview	33
B.2	Client Code	33
C	Server Python Code	35
C.1	Overview	35
C.2	Server Code	35
	Bibliography	36

List of Figures

2.1	IoT Application in the form of a Smart City [11]	4
2.2	Amazon Web Service's IoT Services [9]	5
3.1	Electronic Vehicle Monitoring System Design	15
4.1	The Electronic Vehicle Monitoring System	24
4.2	On Board Diagnostics Two Port	24
4.3	Server data received by the client	26
4.4	Example of the iMiev's CAN data	28



List of Tables

2.1	Wireless Comparison	7
2.2	Socket Vocabulary	9
3.1	Pros and Cons of Existing Hardware	14
3.2	Mitsubishi iMiev OBDII Pinout	16
4.1	J1962 Cable Pinout	19
4.2	Reserved TCP/IP Ports and Protocols	21



Abbreviations

M2M	Machine to Machine
SOC	State of Charge
CAN	Controller Area Network
SSH	Secure SHell
GSM	Global System for Mobile communication
IoT	Internet of Things
OBD	On Board Diagnostics
OBDII	On Board Diagnostics Two
AWS	Amazon Web Services
SDK	Software Development Kit
PPP	Point-to-Point Protocol
APN	Access Point Name
DNS	Domain Name System
MQTT	Message Queuing Telemetry Transport
HTTP	HyperText Transfer Protocol
RTS	Request To Send
CTS	Clear To Send
API	Application Programming Interface



Chapter 1

Introduction

Modes of transportation have become a vital part of life as a human beings. From, cars, buses and trains to planes and boats, all these vehicles require a form of propulsion. At present the most popular version of this for cars is burning fossil fuels to generate a form of energy that is then used to propel a car forward or backward. These fossil fuels however are limited resources, in addition to this they pose additional threats like climate change [17]. As time passes the impact of climate change will grow and become more significant, in order to limit this an alternative to fossil fuel powered cars were developed, in the form of the electric car.

Since it's creation electric vehicles have become more technologically advanced and have been enhanced significantly, this has in turn led to the increase in popularity of electric vehicles. As this demand for electrical vehicles increases so does the demand for energy and more accurate vehicle information. Infrastructure for the charging of these vehicles will need to be put into place, this includes charging stations and means of delivering additional electricity to various locations. Adding to these physical requirements is the user's range anxiety, which is the user worrying about running out of power before reaching their destination, finding parking, and monitoring and managing traffic flows. To solve these issues an electric vehicle monitoring system will be developed.

1.1 Electronic Vehicle Monitoring System

Ideally such a monitoring system would monitor as many aspects of the vehicle as possible, this would allow for tracking of parameters such as location, battery voltage, battery current, battery temperature, speed, SOC, odometer, etc. This realtime monitoring of the various vehicle parameters in addition to allowing the user to have extra information about the vehicle, also allows for manufacturers or third parties to analyse this data and more accurately predict the vehicle's range [13]. The vehicle's range predictor doesn't take the user's route into account, it for instance does not consider changes in elevation, environmental conditions or the driver's behaviour all of which will reduce the vehicle's range [16]. In utilising the additional data a more accurate range can be understood and a prediction of short comings made available, allowing for change.

1.1.1 Monitoring System Requirements

As a result of this the vehicle monitoring system needs to be a realtime systems in order to allow for a timely determination of distance that an electric vehicle can travel at its current SOC, where to find parking, predicting traffic flows and traffic congestion, plus many other vehicle related predictions and analysis. This would also be able to provide data to warn the user if driving the vehicle a certain route would exhaust the vehicle's battery or fuel, resulting in the driver needing to have the vehicle towed. Through the integration with Google maps the system could provide data to display the nearest charging point or petrol station and whether the vehicle will be able to reach it given its current state of charge or available fuel. The benefits of the monitoring system is that it can be used for any vehicle not just the intended electric vehicle purposes.

The monitoring system must be mounted in the vehicle in order to obtain the data required for these calculations to be made. As a result of this, the monitoring system must be wireless, to allow for the realtime data to be transferred from the vehicle wirelessly. The data is intended to be read from a smartphone, making it possible to send the data over both 4G and Bluetooth.

A major aspect of this monitoring system is that it must solely rely on machine to machine communication, meaning that the system must be completely automated with no human intervention needed to run the system other than functioning of the vehicle.

1.1.2 My Contribution

This project had been started by a PhD student who had selected a Raspberry Pi as the device to be used to process data in the car. He had also developed a local network script that could be run using two computers allowing them to communicate with each other on a local network. In addition to this he had purchase a GSM module to be used to acquire GPS coordinates and had written an associated Python script to handle the GSM module and its coordinates. After taking over the project what was required to be done and has been accomplished is:

- Connected the Raspberry Pi to the external Internet via a 4G connection.
- Extended and improved the Raspberry Pi system such that it can obtain data from the vehicle.
- Wrote a client script in Python that reads serial data from the car in real time and can effectively send it to any given IP public address or server.
- Set up a server and wrote a Python script to handle and receive the data being sent from the Raspberry Pi.
- Began on trying to decrypt the data through understanding what the encrypted data meant.
- Automated the entire monitoring system so that it is a machine to machine system.

Chapter 2

Background and Related Work

2.1 Machine to Machine Technology

The interconnection and interoperability of two separate systems, sub-systems and sub-networks is referred to as machine to machine communication or M2M communication. This means that various devices or technologies are communicating without the need for human interference. This is usually implemented through programmed instructions telling the devices what to do. These machines can communicate through wired or wireless methods, most M2M communications are moving to wireless communication leading towards the IoT [7].

M2M communications are required for various parts of the monitoring system to communicate with each other without human interaction. This communication can be done using different forms of wireless communications for example, Bluetooth, UWB, ZigBee and 802.11 protocol (WiFi). These types of wireless communications are useful to know and one of these will need to be incorporated into the monitoring system in order to send the data.

2.2 Internet of things

The Internet of things is the idea of a version of the Internet where everyday objects have a network connection allowing for the sending and receiving of data. ATM's were the first Internet of things objects all the way back in the 1970's, since then more and more machines have become IoT objects and in 2008 it was estimated that there were already more objects connected to the Internet than people [15]. Due to the wide availability of broadband Internet and the decreasing costs associated with connecting to the Internet, more and more devices are being developed with sensors and WiFi connect-ability. This means that an Internet connection is becoming more common on a growing amount of devices such as cars, watches, sound systems, phones, engine and machine parts, lights, fridges, ovens and the list goes on.

The IoT things extends beyond small scale uses and can be applied to large scale ap-

plications like smart power grids, smart roads, smart lighting, etc. These smart networks would ideally all be able to communicate and form an entirely interconnected “smart world”. Libelium is a leader in the field of smart sensors, and are making a significant push to achieve this ideal goal of a smart world. Libelium have a range of smart meters and smart sensors in a wide variety of areas, that range from agriculture to water to health [10]. These sensors are commercially available, and becoming more and more popular as technology develops.

Libelium’s goal is to achieve a smart world where everything is interconnected, a world built on the idea of the Internet of things. A graphical description of this can be seen in the image below, it shows various smart networks and sensor networks that are all connected and send a variety of monitored data to a central hub or its associated data access points.

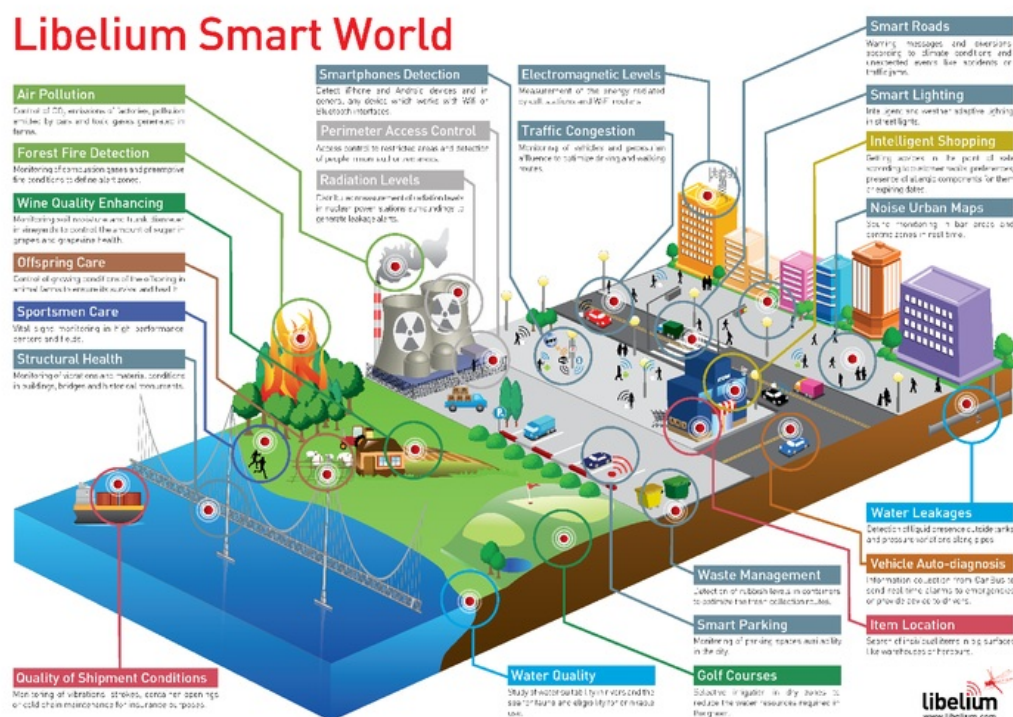


Figure 2.1: IoT Application in the form of a Smart City [11]

Libelium provides the hardware, which is realistically the front end of all these smart networks. The back end of the networks are servers, that handle all the data and communications from object to object and machine to machine. Without all the servers the Internet of things would not be possible, these servers are usually international and controlled by

an external party. Currently the leader in the area of IoT servers is Amazon.com. Amazon had the server and database infrastructure from their book sales website, Amazon then expanded and started renting their servers and server space. Today many companies don't have their own servers or server hardware as they rent server space and server processing power from Amazon.com, DropBox is just one of the companies that do this.

Amazon web services is the platform that allows for devices to connect and use a range of services including, storage, computation, security, analytics and more. Amazon is at the forefront in this aspect of IoT, they are about 10 times the size of their nearest competitor and are continually expanding and becoming increasingly popular.

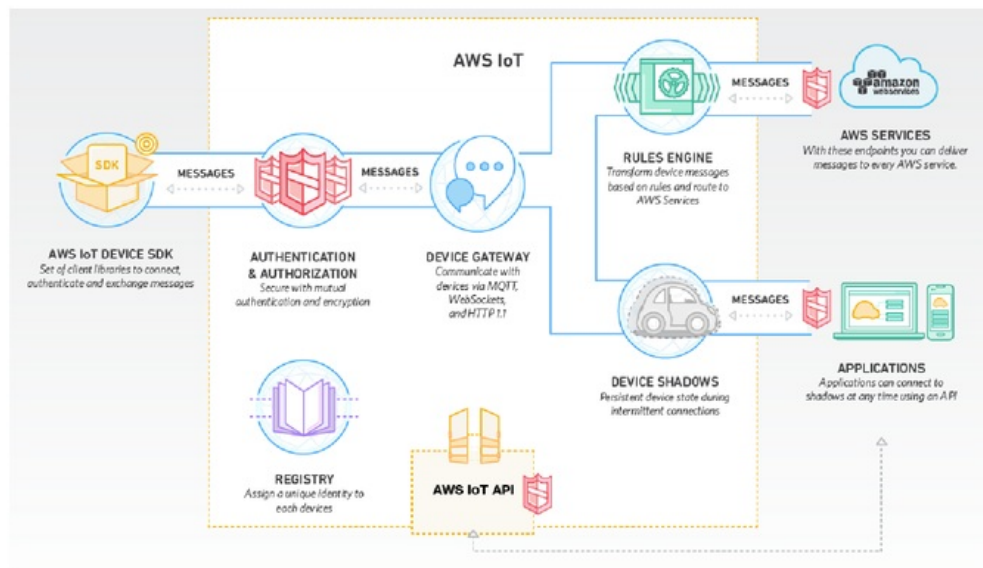


Figure 2.2: Amazon Web Service's IoT Services [9]

AWS IoT Device Software Development Kit

AWS IoT provides its own software development kit, this SDK enables a quick and easy connection for any device or application needing to access to client libraries. The device SDK supports, C, JavaScript, and Arduino and also allows the use of an open source SDK or a user developed SDK. This means that packages, modules and libraries don't need to be installed on the device and can be called using MQTT, HTTP, or WebSockets protocols.

Device Gateway

The AWS IoT device gateway enable a secure and efficient connection allowing for strong and safe communication between the device and AWS IoT. This gateway allows one-to-one

communication and one-to-many communication, this gate way also makes it possible for a device to distribute data to multiple devices or applications using this one-to-many communication method. The Website states that “The Device Gateway scales automatically to support over a billion devices without provisioning infrastructure.” [9]

Device Shadows

AWS IoT allows for the creation of uninterrupted, virtual versions of the device to allow applications or other devices to access the last state of the device. The device shadow and also hold the desired future state of the device, even if the device is offline. The last reported state can be easily accessed and retrieved using the rules engine, this feature is highly useful as a means of a back up.

Rules Engine

Generated data from connected devices can be gathered, processed and analysed using the rules engine, which allows for subsequent IoT applications to be built without the need to manage any of the infrastructure. The rules engine takes incoming messages and data, evaluates and transforms it, and then sends it to another device or cloud based service or wherever the user defines it to go. The rules engine also allows to send to other AWS services like, AWS Lambda, Amazon Machine Learning, Amazon DynamoDB, Amazon Kinesis, Amazon S3, Amazon CloudWatch, and Amazon Elasticsearch Service.

The rule engine allows for the user to create and edit the rules, these rules can trigger certain messages to be sent. For example if a pressure value reaches too high or low it can trigger a rule to send a message or data to a device, this can also be modified to take the average of multiple pressures and then send a message or data if it exceeds a set threshold. The rules engine has inbuilt function that can transform data for analysis and becomes even more powerful when combined with other AWS services like AWS Lambda.

AWS IoT is the future of data processing and analysis and Amazon is leading the way through its cost effective products.

2.3 Wireless data transfer

Wireless data transfer is a major aspect of the monitoring system as the data needs to be communicated to any smartphone in realtime. There are various local are networks that can be easily implemented like ZigBee, Bluetooth and WiFi. A comparison of these wireless connectivity techniques can be easily compared using freely available data [14] as seen in table 2.1.

Depending on the use various types of wireless communications are preferred, smart-grids use a form of M2M communication. Testing has been done on which protocol to use for smart-grids and ZigBee, although slower appears to be the better option as its, low power and has a much larger range than Bluetooth, Wifi uses too much power to be considered for use [5].

Table 2.1: Wireless Comparison

	ZigBee	Bluetooth	Wifi
Range	10-100m	10m	0-100m
Frequency	2.4 GHz	2.4 GHz	2.4 and 5 GHz
Power Consumption	Very low	Low	High
Data Rate	20, 40 and 250 kbps	1Mbps	11 and 54 Mbps

2.4 Collected Data Usages

There are multiple other real world applications of the monitoring system, like the analysis of data from a plug-in hybrid vehicle which has already been conducted [13]. The collected data was gathered during a field test conducted by Volvo and Vattenfall and comparisons were drawn between the electric mode and diesel mode. From this data quantitative results were obtained in the form of various statistics like, average driving speed, driving distance and battery performance in certain weather. This clearly demonstrates that the collected data can be used to provide improved information on the vehicle and its SOC. It furthers the importance of monitoring the vehicle as it allows for the regular collection and analysis of data as it will allow for these statistics to be obtained and collated.

The collected data can be further used to predict traffic conditions and allow for streamlining of traffic. This would be especially beneficial for highly populated areas where traffic is an issue due to it being heavy on a daily basis.

2.5 Python Programming

Python is a popular free to use high-level and dynamic programming language. Its a language recognised and understood by almost any operating system or device. The benefit of this programming language is that it can be used for most purposes, Python also allows for direct execution in the form of scripts and therefore doesn't require a computer to run. Python is also an open source programming language, meaning that anyone can create modules that can be used by anybody provided they download and install it, some of these modules are so frequently used they have been incorporated into the language permanently. As a result of these modules, scripts can be written to automate tasks, like creating clients or servers, connecting to various ports on the device and even sending and receiving data.

2.5.1 PySerial

There are thousands of different modules for all different purposes and objectives, one module in particular pySerial is particularly useful. PySerial allows for the realtime access of serial data from the device's serial port via a Python script. The Justification for this Python script is to allow for the automated reading and handling of the incoming serial data. PySerial has many unique features as shown in the list [12] below:

- Access to the port settings through Python properties.
- Support for different byte sizes, stop bits, parity and flow control with RTS/CTS and/or Xon/Xoff.
- Working with or without receive timeout.
- Same class based interface on all supported platforms.
- File like API with read and write (readline etc. also supported).
- The files in this package are 100% pure Python.
- The port is set up for binary transmission. No NULL byte stripping, CR-LF translation etc. This makes this module universally useful.
- Compatible with io library
- RFC 2217 client

PySerial requires Python 2.7 or later to be run as well as some sort of Java communication, it can be called and then used within a Python script simply by first importing the module by calling `import pyserial` and then the module and all its components are accessible.

2.5.2 Networking Python

Python also already includes networking capabilities, which can be accessed through the importation of the socket module done by calling `import socket`. Python uses socket programming as its type of network communication, these sockets are bidirectional communication channels. Sockets allow for the communication between processes, different local machines, and different machines in different countries. Sockets have a library which contain specific classes for handling various communication, they also have their own vocabulary as set out by table 2.2 [20].

Table 2.2: Socket Vocabulary

Term	Description
domain	The family of protocols, which are used as transport mechanisms, AF_INET, PF_INET, PF_UNIX, PF_X25, etc.
type	The type of communications between the two ends, SOCK_STREAM or SOCK_DGRAM.
protocol	Used to identify a variant of a protocol within a domain and type, usually zero.
hostname	The identifier of the network interface, can be: <ul style="list-style-type: none">• A string, which can be a host name, a dotted-quad address, or an IPV6 address• A string “<broadcast>”, which specifies an INADDR_BROADCAST address.• A zero-length string, which specifies INADDR_ANY• An Integer, interpreted as a binary address in host byte order.
port	The server listens for a client calling on the port number or service name.

Chapter 3

Monitoring System Design

3.1 Introduction

This chapter discusses the existing monitoring systems and the similarities and differences between the monitoring systems, it also overviews the new monitoring system designed and developed for this thesis.

The monitoring system was required to be a realtime system that is universal, so that it can be installed in any electric vehicle. The realtime monitoring allows for in depth analysis of routes and vehicle parameters.

3.2 Related Work

There are pre-existing battery monitoring systems for electric vehicles, however they are only monitoring the current and voltage. The existing design and accompanying documentation contains the implementation and testing of the battery monitoring systems, as well as its components and characteristics [21]. The existing battery monitoring system design is similar to the one that the project intends to create, however it its just monitoring the battery and not other characteristics of the electric vehicle like location and temperature. The existing system is also a wired system that displays the measurements on a screen built into the sun visor and not a wireless system. It is possible to incorporate various aspects of this design into the new one as it is a very similar monitoring system.

3.2.1 ELM327

One of the most popular OBD CAN interface scanners is the ELM327. This scanner plugs into the vehicle's OBDII port and interfaces with a laptop or phone through one of three ways, bluetooth, USB or WiFi. This data can be gathered and analysed by a third party app on a laptop or mobile phone, this means that the device's compatibility becomes wholly reliable on the vehicle's make and if the third party app is capable of decoding the CAN data output by the vehicle.

The advantage is that it is wireless and can communicate to various devices over a range of distances, doing both reading of CAN data and the transfer of the data. Another advantage of this logger is that it gives numerical values for various vehicle parameters, but again this is entirely dependant on the third party app. For electric vehicles this compatibility is very limited, unless the manufacturer has made the CAN information readily available or the online community has developed ways to decode the encrypted CAN data, as is the case with the Nissan Leaf. The device is also extremely inexpensive, and would bring the overall cost of the system down.

This logger is however not compatible with the system requirements as it is not a realtime scanner, this means that the data that is being read and displayed is not current and only being updated periodically or manually after each of the vehicle's trips. There however is no possibility of modifying this system electronically or through software to make it a realtime system that would fit the required system specifications.

3.2.2 CANlogger2000

The CANlogger2000 is another existing data logger, this device is an off the shelf product that from an European based company called CSS Electronics. Due to its overseas location and three hundred and thirty dollar price tag, this device is not particularly popular or mainstream. The CANlogger 2000 is a data logger that allows for realtime logging using a serial interface, this means connecting to the device using a SSH handler like Python or handling it directly using a script and the serial port of the user's device.

This device possessed many advantages, one of them being that it's universal and can read and log the CAN data of absolutely any car that has an OBDII port. This device has two modes, one logs to a file and can be read and accessed only after the logging is done and the second more advantageous mode is the serial mode. This serial mode uses the serial interface to allow for a realtime stream of CAN bus data from the car to the logger and then to a device of the user's choosing. An external client is required to read the data in serial mode CSS Electronics recommends PuTTY as the client, however this can be done multiple ways including using a script to establish the connect and read the data.

The disadvantage of this device is that it has no wireless communication ability and thus can only log the CAN data but cannot transmit the data to an external device like a smart phone or server. However, this can be adapted into a system to allow for wireless communication to a server or mobile phone, making it meet the system requirements.

3.2.3 FleetCarma

FleetCarma is an information and communications technology company that has designed and implemented a monitoring system for electric vehicles. This company sells their product to multiple clients and have reached multiple countries all around the world. FleetCarma has developed electric vehicle modelling technology and an electric vehicle

monitoring system, their scope also extends to electric vehicle research and smart charging systems [19].

FleetCarma opens up its products for a range of purposes, these include research, leasing companies, fleets and electric utilities. The parameters monitored include, electricity usage, battery state of charge, location, temperature and many more parameters, all obtained from a device connected to the vehicle's OBDII port. This company however charges on a subscription basis for the services and products it provides, this price is considerably high and can reach as high as thousands per month depending on the service usage. The service is also dependent on the 3G mobile network, which is already outdated and as time passes will become more and more so. Due to the cost and outdated network with would be more cost effective to build a monitoring system to use than purchase this system from FleetCarma.

3.2.4 FLEX

The FLEX Vehicle Service is a product of Connexion, who have designed and developed a cloud based vehicle monitoring system [1]. This system has been designed and developed to monitor and analyse data from fleets of trucks and cars. Much like FleetCarma the FLEX system can monitor various realtime parameters through the OBDII port, from location to speed and fuel. However unlike FleetCarma, FLEX is not optimised for electric vehicles but rather for fossil fuel vehicles, but the system can still be implemented on electric vehicles.

The FLEX system allows the access of the data through a web-based portal, this allows for graphical and illustrative manipulation of the vehicle's data, meaning it can be easily read, accessed and understood. This system is also requires purchase on a subscription basis, with the hardware included in the first subscription payment. This system is not configured for electric vehicles, this includes both the software and hardware configuration, this means that adaption to electric vehicles would result in additional costs. It would therefore be more efficient and cost effective to design and develop a monitoring system.

3.2.5 Comparison of Existing Hardware

The best choice for the monitoring system needs to be chosen based on the system's merits and its ability to achieve the required specifications. There is also a budget that needs to be taken into account when the monitoring system is being chosen, if the existing monitoring systems can be modified within the budget it is then a viable option. A comparison of these existing monitoring systems has been done and the key points for each of the existing monitoring systems can be seen in table 3.1. The clear choice is the CANlogger2000 as it is reasonably priced and can also log data in realtime, however it will need a communication module in order to communicate the data externally as it doesn't come with inbuilt communications.

Table 3.1: Pros and Cons of Existing Hardware

Hardware	Pros	Cons
ELM327	<ul style="list-style-type: none"> • Inexpensive • Universal • Wireless 	<ul style="list-style-type: none"> • Not realtime logging • Can't be modified • WiFi or bluetooth not 4G
CANlogger2000	<ul style="list-style-type: none"> • Realtime logging available • Can be integrated into a system • Universal 	<ul style="list-style-type: none"> • Expensive • Not wireless communications • Only logs data
FleetCarma	<ul style="list-style-type: none"> • Universal • Wireless communication • Realtime logging 	<ul style="list-style-type: none"> • Very Expensive • Only 3G
FLEX	<ul style="list-style-type: none"> • Realtime logging • Web-based access 	<ul style="list-style-type: none"> • No electric vehicle optimisation • Expensive

3.3 System Model

The design that was chosen uses existing and off-the-self components, that have been integrated together to work as one system. The system mainly centres around the Raspberry Pi 2 module, this is mounted in the car and is powered through the cigarette lighter. This Raspberry Pi module is a single board computer running Debian Linux which is around the size of a credit card, therefore it boots like a computer would with username and password. Due to the constraint that the system must be fully automated, the creation of a script is required, that boots and logs in to the Raspberry Pi module. This script will also start the monitoring programs, this involves running the Python scripts that handle the sending of data to the server and the serial interface that handles the realtime

reading of the CAN data. The Raspberry Pi 2 doesn't have a GSM module, therefore it requires a GSM module to be attached to it in order to obtain the vehicle's location. The Raspberry Pi also has no way of wirelessly connecting to the internet or server, therefore a 4G modem is attached to it allowing it to access the Internet and communicate with the server.

The second major component is the CANlogger2000 [3] it is an off the shelf product that reads the vehicle's CAN data and using a SSH client or handler, the data is displayed through the command line as a serial interface. This serial interface is then accessed using a Python script, this script reads the CAN bus data in realtime via the Raspberry Pi's serial port and then sends it to a dedicated server via a TCP/IP connection. This connection is established through the 4G modem connected to the Raspberry Pi. The modem connection and the Python script to collect and send the CAN bus data are both automatically run when the Raspberry Pi starts up, this is initiated by a shell script that is set to run before login at boot time.

This system model aims to be the most cost effective system, the Raspberry Pi is an inexpensive piece of equipment that can be easily obtained from various places. This means that if the Raspberry Pi has any problem or faults it can easily be fixed or replaced for a relatively low cost. The 4G component is important to the system as there is no system on the market that uses a 4G connection to send or receive any data. The CANlogger2000 can be integrated with the Raspberry Pi through a Python script thus making it a good option to log and read the data. For these reasons the system design had been decided.

Figure 3.1 shows the system broken down into its components so as to visualise the system and its flow.

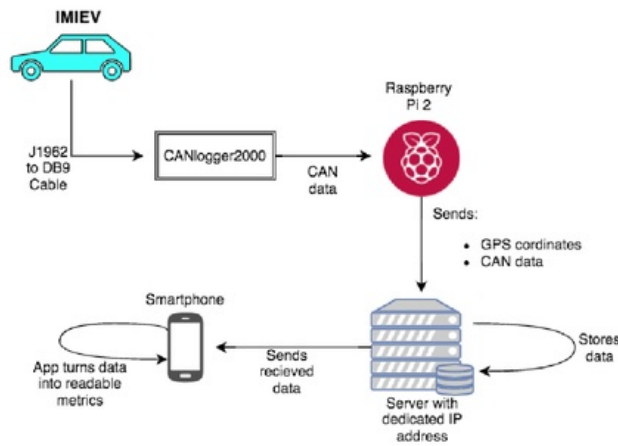


Figure 3.1: Electronic Vehicle Monitoring System Design

3.3.1 Mitsubishi iMiev

The vehicle purchased for the project and alongside projects is a Mitsubishi iMiev, this is a five door electronic vehicle that is produced and sold by Mitsubishi Motors. The iMiev is the world's first modern mass production electronic vehicle that is highway capable, and thus more practical. The monitoring system will need to connect to the CAN bus of this electric vehicle, this means that the pinouts of this OBDII port must be known in order to successfully connect and read the data. This was done by means of the service and user manuals of the vehicle, the pinout of the vehicle can be seen in table 3.2.

Table 3.2: Mitsubishi iMiev OBDII Pinout

Signal	Pin Number
CAN High	6
CAN Low	14
Signal Ground	5
Chassis Ground	4
Power	16

Chapter 4

Results and Future Work

4.1 Development

The development stage involved taking the design and making it into a physical prototype, testing each individual component to make sure no errors arose, and then completing a system test on the prototype to ensure that the correct and expected results were obtained and no errors arise when all the components are integrated. These various stages are explained in detail throughout this chapter.

4.1.1 Raspberry Pi 4G Internet Connection

The development stage began with connecting the Raspberry Pi to the Internet. The Raspberry Pi machine runs Debian Linux as its operating system, which is an older and simpler version of Linux. The reason for running this operating system is that it requires less processing power than the likes of Ubuntu, and other Linux systems. In Debian Linux and some other Linux operating systems when USB modems are physically connected they are detected by the operating not as modems but as mass storage devices. This needs to be rectified before the modem can be connected, this has to be done using a Linux open source program called `usb_modeswitch`.

The package needs to be installed from the Internet, so an ethernet connection needed to be established and the command `sudo apt-get install usb-modeswitch usb-modeswitch-c` used to download all the necessary packages and libraries. `usb_modeswitch` is a program that switches the USB modem from mass storage mode to modem mode, this is done by first locating the configuration file `usb_modeswitch.conf` located in the `/etc` directory and adding the correct configuration, which is set by the modem type and model [4]. Finding the correct modem type and model was significant problem that arose, although it seems like trivial task, the target vendor and product vendor identification numbers needed to exactly match that of the model or the USB wouldn't switch modes. This problem was eventually resolved and the correct configuration file was established and the machine was restarted, which causes `usb_modeswitch` to run automatically and put the USB modem into modem mode.

From this point two more packages are need to be installed, WvDial and PPP Daemon, this is done using the command line and the command `sudo apt-get install ppp wvdial`. The PPP interface needs to be enabled and configured, editing the `/etc/network/interfaces` file and modifying it to read:

```
auto gprs
iface gprs inet ppp
provider gprs
```

The corresponding peer configuration, located in `/etc/ppp/peers` file needs to be edited to include:

```
user 'pi'
connect '/usr/sbin/chat -v -f /etc/chatscripts/gprs -T
preconnect'
/dev/ttyUSB0
noipdefault
defaultroute
replacedefaultroute
hide-password
noauth
persist
usepeerdns
```

The final configuration to be done to connect the Raspberry Pi to the Internet is the configuration of WvDial. WvDial is a PPP dialer that dials the modem and starts PPP daemon that then establishes an Internet connection. The dialer's configuration file is located in `/etc/wvdial.conf`, this file then needs to be edited to include the following:

```
[Dialer Default]
Init1 = ATZ
Init2 = ATQ0 V1 E1 S0=0
Init3 = AT+CGDCONT=1,'IP', 'connect'
Stupid Mode = 1
Modem Type = Analog Modem
ISDN = 0
Phone = *99#
Modem = /dev/ttyUSB0
Username = ''
Password = ''
Baud = 460800
```

The contents of this configuration file, vary from service provider to service provider. Optus was the provider used, Optus has certain set access point names for mobile broadband that depend on the SIM's service type and the plan it is on. The SIM used was a prepaid mobile broadband plan and using the guide provided by Optus [18] the appropriate access point name can be found. Optus' phone is always the standard *99 and it

doesn't require username or password. The init strings change according to the service provider and the type of network connection, which is 4G in this case. The dialer is called using the command `sudo wvdial` in the command line and it dials out obtaining a local and public IP address as well a primary and secondary DNS addresses.

4.1.2 Hardware Testing

After receiving the logger and being able to get access to the electronic vehicle, the CAN logger needed to be tested to see if it had any faults so that they could be rectified or fixed. Before the testing of the logger was even possible a problem arose, the electric vehicle was dead and needed to be charged. Normally a simple problem to be solved, however the charging cable required 15 amps from the wall socket which only provided 10 amps. As a result of this an adaptor was required to be purchased to charge the vehicle, once the adaptor arrived and the vehicle could be charged and the testing of the logger could then begin. During the test however, a fault was detected in which the logger was disrupting the car's electronic dashboard. The lights on the logger showed that it wasn't a fault of the logger as it powered on and automatically configured itself and was waiting to receive data from the vehicle. This meant it would either be a fault in the cable or the disastrous possibility that it was incompatible with electric vehicle.

The first thing that was tested was the cable, this was done by belling out the cable or continuity testing it. This simply just tests which pins are connected on either end of the cable, this was done on the J1962 cable which had the OBDII connector one end and a DB9 plug on the other end. To do these tests a multimeter was connected to the pins on the OBDII end and the pin that it was connected to on the DB9 end was located when the multimeter gave a reading. On the completion of this test it revealed that the pinout diagram for the J1962 cable was incorrect as the continuity test revealed that pinouts of the actual cable were not the same as the cable described. The cable was then spliced open and the correct wires were found by continuity testing the spliced cable and then re-soldering it correctly to a new a DB9 connector. The cable was belled out again to check that the soldering connected properly, the cable was then retested on the vehicle and it worked as it was initially intended. The pinout of the new cable seen in table 4.1.

Table 4.1: J1962 Cable Pinout

Signal	OBDII Pin Number	CAN Logger Pin Number	Cable Colour
CAN High	6	7	Violet
CAN Low	14	2	Grey
Signal Ground	5	3	Black
Chassis Ground	4	3	Black
Power	16	9	Red

4.1.3 Python Coding

The core of the system revolves around the Python code, this python code has been written to enable the Raspberry Pi to read the CAN data from the electric vehicle and then an associated client script to send the obtained data from the Raspberry Pi to an external server. The client script is the script that is on the Raspberry Pi which is located in the vehicle. When the code was written, the code to read the device was written separate to the networking Python code and then later combined, this was done so as to make it easier to debug any problems that came up when writing the Python script. The reader script was written first, but however couldn't be tested until it was connected to the logger which needed to be connected to the car in order to read data.

The reader uses the pySerial package as the logger connects to Raspberry Pi as a serial device, it sends one bit at a time so the reader needed to be able to handle this type of communication. The original idea was to use the SSH handler PuTTY to deal with the serial communications as that's what the recommended handler for the CAN logger was. However due to the script being written in Python there needed to be an additional package installed to enable Python and PuTTY to communicate with each other, this package is called Paramiko [6]. As a result of the limitation of not having a serial device to test with, the code had to be run line by line to make sure that the code and its syntax was correct and bug free.

During the programming process it was found that paramiko was not necessary as Python allows for direct access to the device's serial ports, through another open source package, pySerial. This meant that instead of the script connecting via a SSH client the script will now connect directly to the Raspberry Pi's serial port. This mean't finding the device's physical hardware address, this can be done though plugging in the device and detecting changes in `/dev`, it can vary from device to device but for the Raspberry Pi it was `ttyACM0`. The serial port connection requires a baud rate or bit rate, which determines how much information is transferred per second. For this system the baud rate is set to 115200 bits per second, which is the same baud rate that the CAN logger sends the data, this is done so that none of the messages are missed and not sent or received.

The client and server scripts were next to be created, however due to the additional knowledge required online tutorials were used to assist the creation of these Python scripts [20]. The script requires a host address and a port number, the host is either a public IP address or a URL and the port number can be any number other than the ones in table 4.2 [22].

These ports are all reserved for the own assigned functions, any other port can be chosen the number has no significant baring on it. Initially the client script was written and sent a simple line of text on a local network, this was done to check that the code had no bugs. If bugs were present this made it possible to debug the code and correct it. When both the server and client code were written and running correct they needed to be tested externally and modified such that they could run externally. For this to be accomplished a public IP address is needed as the host address, the reason for this is when

Table 4.2: Reserved TCP/IP Ports and Protocols

Protocol	Port Number
File Transfer Protocol (FTP)	20/21
Secure Shell (SSH)	22
Telnet	23
Simple Mail Transfer Protocol (SMTP)	25
Domain Name System (DNS)	53
Dynamic Host Configuration Protocol (DHCP)	67/68
Trivial File Transfer Protocol (TFTP)	69
Hypertext Transfer Protocol (HTTP)	80
Post Office Protocol (POP)	110
Network Time Protocol (NTP)	123
NetBIOS	137/138/139
Internet Message Access Protocol (IMAP)	143
Simple Network Management Protocol (SNMP)	161/162
Border Gateway Protocol (BGP)	179
Lightweight Directory Access Protocol (LDAP)	389
Hypertext Transfer Protocol over SSL/TLS (HTTPS)	443
Lightweight Directory Access Protocol over TLS/SSL (LDAPS)	636
FTP over TLS/SSL	989/990

on a network the IP address given to a computer or device is dictated by a router and is not the IP address used to receive the incoming data. For this reason the server end needed to be set up such that the public IP address displayed is the one directly from the service provider. When connecting to the internet using a 4G modem the device's public IP address is direct from the service provider, thus attempts were made to start a server on this IP address.

Starting a server on the 4G modem's IP address wouldn't work despite many attempts with different modems and service providers, however none proved fruitful as the server was denied the ability to start on the modem's IP address. This was due to the service providers not allowing access to this IP address as they own the IP addresses and don't allow any other incoming connections. This meant that a server had to be purchased or acquired so that a dedicated IP address can be obtained, this is exactly what is required as the host address. With this dedicated IP address the server can be started on the dedicated address using the developed Python script. This would mean the client would then seek a connection from the dedicated IP address and call on the port number, the server would in turn be listening on its port number and in turn accept the connection

from the client. After acquiring a dedicated IP address from Nectar [2] the code was tested, debugged and modified to work externally.

The two separate scripts, the reader and the client scripts both had to be combined, in order for the system to send the data being read. This involved embedding the reader script inside the client script such that the data being sent was not a simple line of text but instead the serial data being read from the serial port of the Raspberry Pi.

4.1.4 Shell Scripting

Shell scripts are scripts designed to be run by Unix shell in the command line. They're usually used to print text or execute files, the shell script written was written to firstly connect to the Raspberry Pi to the Internet and to run the Python client script. The shell script was written in the bash shell and looks as follows:

```
#!/bin/bash

sleep 15s

sudo wvdial &

sleep 15s

python3 /home/pi/Desktop/client.py &
```

The reason for this shell script is there needs to be an Internet connection established before the Python client script is run. Internet connection is done by the `sudo wvdial` command, however when creating the script this command wouldn't exit by itself and this resulted in the following commands in the shell script not being able to run. To try and prevent this the `&` was added as it pushes the command to the background to run, however this didn't work as the connection command requires more time to fully complete its execution process. To hold the execution of the Python script and stop it from executing before the Internet connection is established, a delay was added. This delay `sleep 15s` was added after the Internet connection command to prevent the Python client file from running before the Internet connection has been established, this delay of 15 seconds was tested multiple times and found to be an acceptable duration. The shell script was later appended to include a delay before the Internet connection command as when the shell script was set to run automatically, the USB modem needed to initialise before the Internet connection command is executed.

Due to the machine to machine system requirement, the Raspberry Pi needs to automatically connect the Internet and run the client Python script, this means setting the shell script to run automatically. The shell script needs to be given the permissions to run automatically, this is done through the command `chmod 755 my_script` which gives the file read, write and execute permissions or 700 will give read and execute permissions. To then run the file automatically, changes have to be made to the root files of the Rasp-

berry Pi. The file that needs to be modified is in `/etc/init.d/rc.local`, this `rc.local` file contains a section in which files can be set to boot before login. In the `rc.local` file, the path and location of the shell script needs to be added in the start section in order for it to run on start up as seen in the code below.

```
case "$1" in
    start)
        do_start
        #shell script goes here
        /home/pi/Desktop/3g.sh

        ;;
    restart|reload|force-reload)
        echo "Error: argument '$1' not supported" >&2
        exit 3
        ;;
    stop)
        ;;
    *)
        echo "Usage: $0 start|stop" >&2
        exit 3
        ;;
esac
```

In order for this shell script to run automatically the Raspberry Pi needs to be logged in, there were two options to do this, write a script to run in boot time to log in automatically or just disable the log in altogether. Disabling the login was the most efficient approach, this was done by first editing the `/etc/inittab`. Disabling the `getty` login program can be done by commenting out the line `1:2345:respawn:/sbin/getty 115200 tty1` in `/etc/inittab` with a `#` and then replacing it with `1:2345:respawn:/bin/login -f pi tty1 </dev/tty1 >/dev/tty1 2>1`, saving the file and the Raspberry Pi boots without requiring a login.

The Raspberry Pi will now start up and automatically, connect to the Internet and run the client Python file, which will automatically read the data, connect to the server and send the data to the server making it a machine to machine communication.

4.2 Results

The final designed system produced the results that were expected. The below diagram shows the system on its own, the OBDII plug end of the cable needs to be then plugged into the OBDII port of the vehicle and the USB into the cigarette lighter USB adaptor to power the Raspberry Pi.

The OBDII port for each car is located in a different place, the user manual or service manual will usually have the location of the OBDII port along with a diagram. For the



Figure 4.1: The Electronic Vehicle Monitoring System

Mitsubishi iMiEV the OBDII port is located on the driver's side, under the panel below the steering wheel and to the left hand side of that panel.



Figure 4.2: On Board Diagnostics Two Port

To receive and read the data the server needed to be set up, this server was set up using the nectar website. This site allows for the creation of a virtual server, that has a dedicated IP address and can be accessed at any point of time. During the creation of the server the first thing that needed to be done was select the server's image, a Ubuntu server was chosen as the image, the server also needed to allow SSH connections in its

security and authentication settings. A key pair then needed to be created in order to access the server at a later point in time, the command to do this is `ssh-keygen -t rsa -f cloud.key` and a password was also set to allow for access to the server. The keygen command generates both a public key and a private key however only the private key is required to log in to the server along with the password.

Once the server had been set up new security and authentication settings needed to be added in order to allow incoming connections to the server on the designated port. This was done through adding a new security group that accepted the incoming connections on the specified port from the client script. As the server now accepts connections on the designated port then it is possible to log on and read the data. The login process requires, the key pair, password and using SSH, the command `ssh -i cloud.key -l ubuntu 115.146.86.77` needs to be executed through the command line. Once the command has been executed the password set for the server will be requested, once this has been entered it and accepted the server will log on and the server Python script can be run and the data can be received.


Figure 4.3 shows the data sent from the monitoring system in the vehicle to the server, the data is being received by the server and displayed.

4.3 Future Work

Improvements and further developments can be made to the system to enhance both its performance and usability. Due to time constraints several things were unable to be undertaken as a result the future work to be done on the system include things like developing a mobile app to improve usability, decrypting the CAN data, and a server alternative to improve the functionality.

The current server has certain limitations, the server is a viable option for research purposes but not for commercial uses. The current server has the benefit of being free to use for researchers, students and professors, however it wouldn't be a viable option for important task like tracking ambulances for example. The server host states that they allow constant anytime access to the server, however due to its large client base it requires constant upkeep and the server could potentially go down at any point in time. There is one bug that resides on the server end that has already been detected. The bug results in the connection being refused when the client tries to connect, this means that the server can not be found by the client as the server has not started correctly. The only fix for this is to restart the server connection, run the Python server script again and then reconnect by means of the client.

This is problematic as the only possible way to tell the connection has been refused is through the time it takes to connect, too long and the connection has been refused. This is due to the fact that the system in the car has no screen therefore there is no way to see that the connection to the client has been refused, the other option is to try and connect to the server via a computer thus testing the connection. For practical and commercial applications a improved and more reliable server should be obtained, this will come at a



```

johannmathias — ubuntu@logger: ~ — ssh — 90x53
Last login: Thu Oct 27 14:14:42 on ttys000
Johann-MacBook-Pro:~ johannmathias$ ssh -i cloud.key -l ubuntu 115.146.86.77
Saving password to keychain failed
Enter passphrase for key 'cloud.key':

-----
NeCTAR Ubuntu 16.04.1 LTS x86_64
Image details and information is available at
https://support.nectar.org.au/support/solutions/articles/5000106269-image-catalog
-----

* Documentation: https://help.ubuntu.com
* Management:    https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage
Last login: Thu Oct 27 03:15:20 2016 from 114.75.194.103
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@logger:~$ python3 server.py
Socket created.

Starting up Server on 115.146.86.77 port 5568
Socket bind complete.
The Server is listening for a client
Connected to: 49.183.130.166:48406
1;0;0;36;

27T042213399;0;356;30;0;0;60;102;0;0;152;
27T042213400;0;660;8;21;0;1;119;42;0;31;
27T042213403;0;57;0;12;
27T042213406;0;229;0;0;0;28;
27T042213406;0;310;0;0;0;0;0;29;
27T042213407;0;314;0;0;0;0;0;25;
27T042213407;0;319;255;246;255;246;0;20;0;25;
27T042213407;0;344;0;0;2;162;0;0;27;
27T042213410;0;420;0;0;0;0;0;24;
27T042213410;0;426;127;255;0;127;255;0;104;28;
27T042213410;0;432;0;15;255;255;0;0;16;
27T042213410;0;464;0;0;0;0;0;10;
27T042213414;0;157;0;27;
27T042213416;0;229;0;0;0;35;
27T042213417;0;310;0;0;0;0;0;44;
27T042213417;0;314;0;0;0;0;0;40;
27T042213417;0;319;255;250;255;246;0;0;41;
27T042213417;0;344;0;0;2;162;0;0;42;
27T042213417;0;300;0;0;2;195;0;0;33;
27T042213418;0;392;0;0;0;1;0;21;
27T042213418;0;149;128;64;0;0;0;0;29;
27T042213419;0;884;74;96;0;10;0;0;42;
27T042213420;0;887;4;23;68;44;4;0;50;42;
27T042213420;0;880;7;144;230;33;45;96;0;32;
27T042213420;0;1029;0;0;36;0;64;0;0;5;
27T042213421;0;1064;0;0;0;1;119;42;15;
27T042213426;0;229;0;0;0;50;

```

Figure 4.3: Server data received by the client

monetary cost. The alternative to just dealing with the server issue is to obtain a server from Amazon.com. Amazon rent out their hardware and charge per hour of usage or per gigabyte processed, prices vary depending on the location of the server, American

servers being the cheapest and the servers get more expensive for less popular countries like Australia and the Asia Pacific region.

For this system Amazon VPC is the most suitable of the Amazon Web Services available as it allows for both storage and computation. Amazon has a proven computing environment and networking capabilities, Amazon VPC also allows for quick scaling that enables the server to be scaled up or down depending on the user's needs while only requiring the user to pay for what has been used [8]. The pricing for Amazon VPC is approximately \$0.06 per hour or \$0.06 per gigabyte. The advantage of using an Amazon.com server is that it can easily be integrated with other Amazon products like Amazon DynamoDB which is a database that can store the data or AWS IoT which is the future of data management and a central hub for data collection and analysis. AWS IoT however slightly more expensive at \$6 per million messages, although as IoT becomes more common the price will decline. After the server receives the data it needs to be decrypted, this can be done using an Amazon computational service or via an external application, an Internet application or even a mobile application.

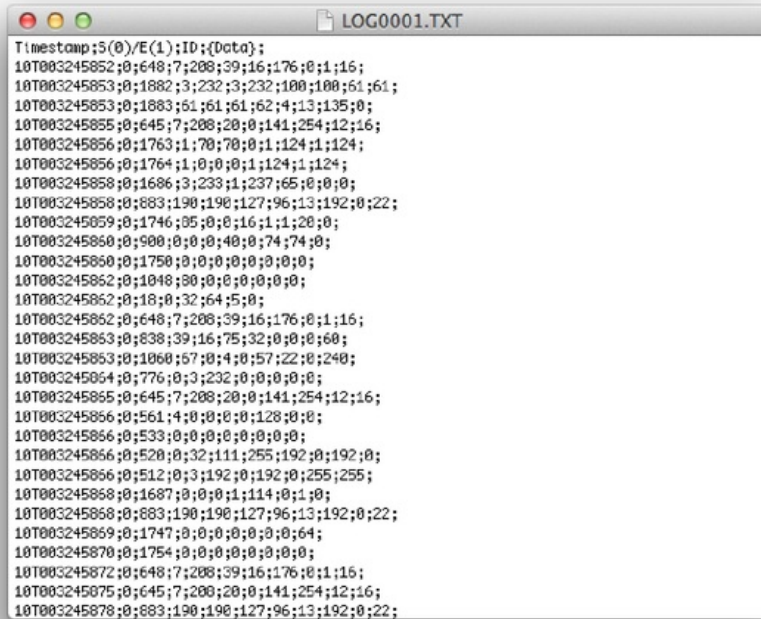
A mobile application to convert the data seems like the most efficient and cost effective method, however if the data required to be processed is too much for a smart phone then alternatives are available thanks to Amazon Web Services. The data needs to be viewed by a user in realtime therefore a mobile application will be needed regardless, therefore it makes sense that the computation and decryption of the CAN data is done by the smartphone application. The computation can be limited such that it only updates every 10 or 30 seconds, or even longer depending on the parameter that needs to be displayed. Parameters like temperature, change a considerably slower than other parameter thus then can be computed with longer time intervals, this can be programmed to be done by the mobile application itself, however if the data needs to be computed as fast as it is being output then Amazon EC2 or AWS Lambda would be an appropriate choice.

Amazon EC2 and AWS Lambda are computational engines that make cloud computing easier for developers to use. AWS Lambda is overall the more expensive option of the two but it is also the more functional of the two, it allows the running of the code without the need to manage servers to access it. AWS Lambda also has its own code that runs in parallel with the user's code to enable the smooth operation of the required functions set out by the user. The user's code can be uploaded to AWS Lambda and can be triggered to run by other AWS services, websites or even smart phone application activity. AWS Lambda runs the code using only the required resources and charges the user for the computational time used, which is on a subsecond metering basis.

A major advantage of using AWS Lambda is that it can call any other Amazon service, this includes the Amazon DynamoDB database, Amazon EC2 computation or Amazon VPC networking services. This allows a dynamic handling of the data as AWS Lambda already has inbuilt integration for various backends like web applications, mobile applications and IoT. This means the data can be easily extracted, transformed and displayed through graphical analytics. The data can then be read and analysed relatively easily, it also allows for logistics to be run on the data both externally and within Amazon Web Services. When a substantial amount of data has been collected through a substantial

amount of cars being monitored, the data can then be used for its intended purposes of managing congestion, finding parking and the additional monitoring of electric vehicles in order to build and implement charging infrastructure. However before this can be done the data needs to be decrypted so that the data can be read quickly and easily.

The CAN bus data is encrypted by the manufacturer so as to ensure that its private and not being misused by anyone. For the Mitsubishi iMiev the data is encoded hexadecimal or decimal, an example of this can be seen in figure 4.4.



```

Timestamp;5(0)/E(1);ID:{Data};
10T003245852;0;648;7;208;39;16;176;0;1;16;
10T003245853;0;1882;3;232;3;232;100;100;61;61;
10T003245853;0;1883;61;61;61;62;4;13;135;0;
10T003245855;0;645;7;208;20;0;141;254;12;16;
10T003245856;0;1763;1;70;70;0;1;124;1;124;
10T003245856;0;1764;1;0;0;0;1;124;1;124;
10T003245858;0;1686;3;233;1;237;65;0;0;0;
10T003245858;0;883;190;190;127;96;13;192;0;22;
10T003245859;0;1746;05;0;0;16;1;1;20;0;
10T003245860;0;900;0;0;0;40;0;74;74;0;
10T003245860;0;1750;0;0;0;0;0;0;0;0;
10T003245862;0;1048;0;0;0;0;0;0;0;
10T003245862;0;18;0;32;64;5;0;
10T003245862;0;648;7;208;39;16;176;0;1;16;
10T003245863;0;838;39;16;75;32;0;0;0;60;
10T003245863;0;1060;67;0;4;0;57;22;0;240;
10T003245864;0;776;0;3;232;0;0;0;0;0;
10T003245865;0;645;7;208;20;0;141;254;12;16;
10T003245866;0;561;4;0;0;0;128;0;0;
10T003245866;0;533;0;0;0;0;0;0;0;
10T003245866;0;520;0;32;111;255;192;0;192;0;
10T003245866;0;512;0;3;192;0;192;0;255;255;
10T003245868;0;1687;0;0;0;1;114;0;1;0;
10T003245868;0;883;190;190;127;96;13;192;0;22;
10T003245869;0;1747;0;0;0;0;0;0;0;64;
10T003245870;0;1754;0;0;0;0;0;0;0;0;
10T003245872;0;648;7;208;39;16;176;0;1;16;
10T003245875;0;645;7;208;20;0;141;254;12;16;
10T003245878;0;883;190;190;127;96;13;192;0;22;

```

Figure 4.4: Example of the iMiev's CAN data

The figure shows a the data as output by the CAN logger, each of the values are separated by a semicolon, this can be seen in the figure. The logger outputs the time stamp first, then the base data of the log file, which is set to decimal, following that is the CAN data identification usually followed by eight data bits. The identification value is key in the decryption process as it identifies what data is encoded in the following eight bits, to decrypt the data it must be known what data is stored in the eight bits for this to be known the identification key must be known. Once the identification key is known then an equation can be reverse engineered using the 8 data bits and the known

value being sought out. This can be quite time consuming and difficult due to this fact, Mitsubishi had been contacted and the ability to decrypt the data had been requested however Mitsubishi are yet to comeback with a response. This decryption can be done on multiple platforms but would be most efficiently it is done on a mobile app or AWS Lambda or a combination of the two.

Appendix A

Consultation Form

A.1 Overview

This section contains the supervisor consultation attendance form as required by the department.

A.2 Consultation Attendance Form

Consultation Meetings Attendance Form

Week	Date	Comments (if applicable)	Student's Signature	Supervisor's Signature
0	28/07	group only	J. Math	PS.
1	04/08	OK.	J. Math	PS.
2	09/08	OK ^{not} use...	J. Math	PS.
3	16/08		J. Math	PS.
4	23/08		J. Math	PS.
5	30/08		J. Math	PS.
6	12/09		J. Math	PS.
8	26/09		J. Math	PS.
10	18/10	OK.	J. Math	PS.
11	25/10		J. Math	PS.
12	1/11	OK.	J. Math	PS.

Appendix B

Client Python Code

B.1 Overview

This section contains the Python code for the client communication end of the system.

B.2 Client Code

```
import socket
import serial
import time
import sys
import select
import csv

import http.client
import json
from urllib.parse import quote_plus

base = '/maps/api/geocode/json'

PORT_NAME = '/dev/ttyACM0' #port address of the serial device
BAUDRATE = 115200

def refresh_serial_port(serial_port):
#clears the serial port to avoid any errors
    serial_port.setDTR(False)
    time.sleep(1)
    serial_port.flushInput()
    serial_port.setDTR(True)
```

```

def Main():
    host = '115.146.86.77'
    port = 5560
    s = socket.socket()
    s.connect((host, port))

    print('Connected to the Server')
    print('Opening serial port ' + PORT_NAME + ' at '
          + str(BAUDRATE) + ' baudrate...')

    ser = serial.Serial(port=PORT_NAME, baudrate=BAUDRATE, timeout=1)
    refresh_serial_port(ser)

    print('CAN data is being transmitted')

    while True:
        # f = open('dataFile.txt', 'a')
        data = ser.readline().decode('utf-8')
        s.send(data.encode('utf-8'))
        # f.write(str(ser.readline().decode('utf-8')))
        # f.close()
        # f = open('dataFile.txt', 'a')
        # geocode('Macquarie University, Sydney, NSW')

    print('Closing serial port...')
    ser.close()
    s.close()

def geocode(address):
    #taken from previous work done by PhD student
    path = '{}?address={}&sensor=false'.format(base, quote_plus(address))
    connection = http.client.HTTPConnection('maps.google.com')
    connection.request('GET', path)
    rawreply = connection.getresponse().read()
    reply = json.loads(rawreply.decode('utf-8'))
    print(''\n \n \t The longitude and latitude is:')
    print('\t')
    print(reply['results'][0]['geometry']['location'])

if __name__ == '__main__':
    Main()

```

Appendix C

Server Python Code

C.1 Overview

This section contains the Python code for the communication at server end of the system.

C.2 Server Code

```
import socket

host = '115.146.86.77' # server IP address
port = 5560

server_address = (host,port)

def Connect():
    s.listen(2) # 2 connections
    print('The Server is listening for a client')
    c, address = s.accept() # accepts x
    print('Connected to: ' + address[0] + ':' + str(address[1]))
    return c

def Transfer(c):
    while True:
        f = open('dataFile.txt', 'a')
        data = c.recv(1500) # receive the data
        print(data.decode('utf-8'))
        f.write(str(c.recv(1500).decode('utf-8')))
        f.close()
        f = open('dataFile.txt', 'a')
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('Socket created.')
s.setblocking(1)
s.settimeout(10000)
print('Starting up Server on %s port %s ' % server_address)
try:
    s.bind((host, port))
except socket.error as msg:
    print('Socket failed to bind')
print('Socket bind complete.')

while True:
    try:
        c = Connect()
        Transfer(c)
    except:
        break
```


Bibliography

- [1] Connexion. (2016, October) Flex vehicle system. [Online]. Available: <http://www.flexvs.com/>
- [2] N. Directorate. (2016, October) Nectar server. [Online]. Available: <http://nectar.org.au/>
- [3] C. Electronics. (2016, August) Canlogger2000. [Online]. Available: <http://www.csselectronics.com/screen/product/can-bus-logger-canlogger2000>
- [4] emr. (2015, June) Guide how to use raspberry pi with 3g usb stick. [Online]. Available: <http://copyndone.com/2015/06/27/guide-how-to-use-raspberry-pi-with-3g-usb-stick/>
- [5] Z. M. Fadlullah, M. Fouda, and N. Kato., "Toward intelligent machine-to-machine communications in smart grid," *IEEE Communications Magazine*, vol. 49(4), no. 53-58, 2011.
- [6] J. Forcier. (2016, August) Paramiko. [Online]. Available: <http://www.paramiko.org/>
- [7] I. GENERAL. (2013, May) What is the difference between m2m and iot? [Online]. Available: <https://iot.telefonica.com/blog/what-is-the-difference-between-m2m-and-iot>
- [8] A. W. S. Inc. (2016, October) Amazon virtual private cloud (vpc). [Online]. Available: https://aws.amazon.com/vpc/?nc2=h_m1
- [9] ——. (2016, November) Aws iot features. [Online]. Available: <https://aws.amazon.com/iot/how-it-works/>
- [10] Libelium. (2016, October) 50 sensor applications for a smarter world. [Online]. Available: http://www.libelium.com/resources/top_50_iot_sensor_applications_ranking/#show_infographic
- [11] ——. (2016, October) Smart world image. [Online]. Available: http://www.libelium.com/wp-content/themes/libelium/images/content/applications/libelium_smart_world_infographic_big.png

- [12] C. Liechti. (2001-2015) Pyserial. [Online]. Available: <https://pythonhosted.org/pyserial/pyserial.html#overview>
- [13] P. Lin, "Statistical analysis of lithium-ion battery data collected on-board electric vehicles," Master's thesis, Royal Institute of Technology, Stockholm Sweden, 2013.
- [14] A. Lowne and H. Doughty, "Bluetooth moves beyond the earpiece to rule other applications," June 2010. [Online]. Available: <http://mobiledevdesign.com/learning-resources/bluetooth-moves-beyond-earpiece-rule-other-applications>
- [15] J. Morgan. (2014, May) A simple explanation of 'the internet of things'. [Online]. Available: <http://www.forbes.com/sites/jacobmorgan/2014/05/13/simple-explanation-internet-things-that-anyone-can-understand/#7815359f6828>
- [16] K. M. Muttaqi, A. D. T. Le, J. Aghaei, E. Mahboubi-Moghaddam, M. Negnevitsky, and G. Ledwich, "Optimizing distributed generation parameters through economic feasibility assessment," *Applied Energy*, vol. 165, pp. 893–903,, 2016.
- [17] U. of Concerned Scientists. (2016, September) Cars and global warming. [Online]. Available: <http://www.ucsusa.org/clean-vehicles/car-emissions-and-global-warming#.WB78FWR94y4>
- [18] Optus. (2016, September) Our mobile broadband/home wireless broadband apns & technical settings. [Online]. Available: <http://www.optus.com.au/business/support/answer/our-mobile-broadband-or-home-wireless-broadband-apns-technical-settings?requestType=NormalRequest&id=1379&typeId=5>
- [19] C. Technologies. (2016, October) Fleetcarma. [Online]. Available: <https://www.fleetcarma.com/>
- [20] Tutorialspoint. (2016, August) Python network programming. [Online]. Available: https://www.tutorialspoint.com/python/python_networking.htm
- [21] J. I. Vial and J. W. Dixon, "Monitoring battery system for electric vehicle, based on "one wire."
- [22] S. Wilkins. (2012, April) Tcp/ip ports and protocols. [Online]. Available: <http://www.pearsonitcertification.com/articles/article.aspx?p=1868080>