



Edge-driven Real-time Vehicular Services CM3203

Samuil VELICHKOV
C1418948

BSc Computer Science with Year in Industry

Supervisor: Prof. Omer Rana
Moderator: Dr. Bailin Deng

11 May, 2018

Abstract

With the ever-increasing popularity of the current cloud-based Internet of Things (IoT) model there has been a tremendous shift towards smart, connected devices which constantly relay data in an attempt to provide better, more insightful services, ranging from smart home automation to industrial applications. Resulting from this data deluge is the notion of Fog and Edge Computing which attempts to bridge the gap between the benefits of cloud computing services and the demand for high bandwidth, low latency, location aware, mobility applications. As defined [14] the Fog nodes in the Fog computing notion, are located as close to the source of information as possible, usually one hop away from Edge nodes which come in the form of sensors, mobile devices, end users, vehicles etc.

Time-sensitive services are hosted at the frontier of the network subsequently improving quality of service, reducing latency and increasing end-user satisfaction. This new paradigm is ideal for becoming an enabling technology for emerging applications such as autonomous vehicles, wearable technologies and augmented reality that demand real-time, near-fixed latency response times. The project at hand explores the implications and advantages of this new paradigm and puts them into practice by devising a prototype simulated connected vehicle in a Fog and Edge environment that aims to improve road safety and collision avoidance in the ever-more connected world.

Acknowledgements

I would like to personally thank my supervisor, Prof. Omer Rana, for providing me and the project with guidance, ideas and feedback from the very start until its submission.

I would also like to give credit to Open Source software and technologies as I am a strong believer that software and knowledge should be shared and made available for everyone to benefit. Without software companies such as the Qt Company and the Linux Foundation, this simulator would not have been possible.

Most of all I would like to thank my family and close friends for supporting me throughout the entire duration of my degree.

Table of Contents

Abstract	1
Acknowledgements	2
Table of Figures	5
1. Introduction	6
1.1. Project Outline	6
1.2. Aims and Objectives	7
1.3. Scope of Project	8
1.4. Project Importance	9
1.5. Audience	10
1.6. Approach	10
1.7. Assumptions	12
1.8. Project Outcomes	12
2. Background	13
2.1. Problem Context	13
2.2. Constraints	14
2.3. Theory	14
2.4. Existing Solutions	15
2.4.1. IEEE 802.11p and Dedicated Short-range Communications (DSRC)	16
2.4.2. Cellular Networks and 5G	17
2.5. Methods and Tools Used	18
2.5.1. Controller Area Network (CAN) Bus and SocketCan	18
2.5.2. Qt and QML	20
2.6. Hardware	22
2.7. Research Questions	23
3. Specification and Design	24
3.1. System Overview	24
3.1.1. Requirements	24
3.1.2. Design and Architecture	25
3.1.3. Use Case	29
3.2. CAN Bus Simulator	31
3.2.1. Requirements	31
3.2.2. Design	32
3.3. Instrument Cluster	33
3.3.1. Requirements	33
	3

3.3.2. Design	35
3.4. Fog Network	37
3.4.1. Requirements	37
3.4.2. Design	38
4. Implementation	40
4.1. Changes to Initial Plan	40
4.2. CAN Bus Simulator Implementation	43
4.2.1. Configuration	43
4.2.2. Implementation	44
4.3. Instrument Cluster Implementation	48
4.3.1. Configuration	48
4.3.2. Implementation	50
4.4. Fog Network Implementation	52
4.4.1. Configuration	52
4.4.2. Implementation	53
4.5. Overall System	55
4.6. Successes and Failures	56
5. Evaluation and Results	57
5.1. Testing	57
5.2. Progress against Objectives	61
6. Future Development	64
7. Conclusions	66
8. Reflection on Learning	67
8.1. Personal Skills	67
8.2. Progress against SFIA	68
Table of Abbreviations	70
Appendices	72
References	74

Table of Figures

Figure 1: ‘Feature Driven Development’ cycle (FDD)	10
Figure 2: Fog Network Overview	15
Figure 3: Future V2V and V2I Communications	16
Figure 4: Future 5G V2X connected network	17
Figure 5: CAN Bus Frame structure	19
Figure 6: CAN Bus virtual interface in a Linux environment	19
Figure 7: SocketCAN communication layers	19
Figure 8: Visual Representation of the Instrument Cluster	20
Figure 9: Hardware layout of the simulated system	22
Figure 10: System design iteration steps	26
Figure 11: Flowchart of the whole system (Fog Node, Connected Vehicle and Collision Vehicle)	27
Figure 12: Entity-relationship diagram of the components	28
Figure 13: Diagram of a collision use-case scenario	29
Figure 14: Use Case Diagram #1, collision notification (V2V)	30
Figure 15: Use Case Diagram #2, internal vehicle signal	30
Figure 16: CAN Bus flowchart and data propagation	32
Figure 17: Instrument Cluster flowchart and data propagation	35
Figure 18: Instrument cluster design representation	36
Figure 19: Fog node flowchart and data propagation	39
Figure 20: CAN Bus: Overridden keyboard input class	45
Figure 21: CAN Bus: Key press iteration	45
Figure 22: CAN Bus: Key pressed handling	46
Figure 23: CAN Bus: Message sending	46
Figure 24: CAN Bus: SSL configuration	47
Figure 25: CAN Bus: Server threading	48
Figure 26: CAN Bus: Relaying message	48
Figure 27: Qt Creator IDE	49
Figure 28: Instrument Cluster running in Qt Creator IDE	49
Figure 29: Instrument Cluster: listener for CAN Bus and Fog Node connections	50
Figure 30: Instrument Cluster: On connection with CAN Bus	51
Figure 31: Instrument Cluster: Checking message types	51
Figure 32: Instrument Cluster: Collision data checking	52
Figure 33: Fog Node: Listening for vehicle connections	53
Figure 34: Instrument Cluster: Handling connections and their data	53
Figure 35: Fog Node: Collision Checking and Notification sending	54
Figure 36: Vehicle Collision notification script	54
Figure 37: Hardware Fog Node and Connected vehicle Simulator representation	55

1. Introduction

The influx of data usage in the field of information and communication technology, largely attributed to the recent popularity of the Internet of Things (IoT), poses a challenge for established cloud computing services. For latency-sensitive applications, network congestion and the distance between end devices and the cloud are increasingly becoming an issue which is especially true for content delivery applications. It is also fast becoming difficult for service providers to keep up with their Service level agreements (SLA) which forces building of new data centers at new locations in order to keep up with demand[1].

Another particularly relevant problem is the gaining popularity of connected, semi-autonomous and in the near future, fully autonomous vehicles, which are predicted to generate and consume roughly 4 terabytes of data, every eight hours of driving[2]. Supporting the vast number of vehicles predicted to be on the roads of the future requires a novel, more decentralised approach at networking and it is “Fog and Edge Computing” which will be in the forefront of solving this data deluge.

In addition, automotive manufacturers are trying to pack more and more features, sensors and devices into their connected and autonomous vehicles that production prices are rapidly increasing with the more specialised hardware and software they have to use. This becomes a major problem in prototyping and developing vehicles as the complexity increases[3], new, more flexible approaches must be taken. This is where a revolutionary internal network vehicle simulator can help to facilitate rapid conceptualising and prototyping of vehicular features and services.

1.1. Project Outline

The overall focus of this project is to study and explore the implications of Fog and Edge computing in the field of vehicular services in order to improve existing cloud-centric services. By creating a prototype simulation of a vehicle in a Fog network environment, the implications of the Fog theory can be evaluated using a real example.

One of the prime use case examples for showcasing the potential of the technology would be collision avoidance. The project proposes having Fog based safety service, to which a connected car is subscribed and sends a notification of a collision if one has occurred in proximity of the vehicle. The service receives the initial notification from other connected participants on the road that have been involved in the incident. The use case is especially valid if the collision has happened in front of the vehicle where it has not been detected by the car’s onboard sensors, because of obstructions or limitations in visibility. The connected car driver, after being notified can approach driving with higher cautiousness and trigger

application of the brakes to slow their speed down and prevent further multi-vehicle incidents. This use-case can be mainly applied to high speed driving, on a motorway or speed road, where cars are normally driving with a speed upwards of 50Mp/h and braking distances are significant, especially in bad weather and road conditions and driver visibility is affected.

Shorter range collision avoidance is already handled by on-board car sensors. This fusion of information for short and long range collision avoidance will lead to safer road conditions.

Bringing this use case into a functioning system requires three major components to be developed:

- A Connected Car Simulator, to allow for the evaluation and testing of the Fog and Edge services in simulated target vehicle environment.
- A Vehicle Digital Instrument Cluster Head Unit, to visualise the results of the simulation and secure communication between the vehicle and the network.
- A Fog Network, a simulated network of Fog nodes which facilitate real-time vehicular services that communicate with edge devices (vehicles).

Developers would benefit from this project by not having to test the service on a real, physical platform. Current automakers have a deliberate process in developing new platforms for vehicles and that takes time and resources. Providing the automakers with an integration of some of the basic building blocks would help them add additional features and rapidly prototype their own integrations and shorten the time from concept to a fully working product.

1.2. Aims and Objectives

The aim of this project is to design and develop a functional prototype simulator of a connected car and a simulated fog environment in which it can operate. This simulator will be used in the context of a very relevant use case of collision avoidance in vehicular systems. The environment would try to mimic as closely as possible a use case scenario in which there are multiple cars on a busy public road or motorway and one of the geo-located connected vehicles would get into an incident. The main objectives of the project are:

Objective 1: Create a connected vehicle simulator prototype that mimics as closely as possible a connected car's internal network

- Research, devise and develop a virtualized internal car network to send vehicular data and read from.
- Secure the inter-vehicle communication to prevent packet sniffing or spoofing.

Objective 2: Visualise the simulated connected car's functionality using an automotive instrument cluster

- Design and develop an instrument cluster to visualise the data on the internal car network.

Objective 3: Create a Fog Network in supporting vehicular services

- Design and implement a Fog network of nodes for receiving and processing location-based information sent by a fleet of connected vehicles.
- Simulate a use case scenario of alerting the network participants of a vehicle involved in an accident.

Another key problem, solving of which would make the project more accessible to the audience, is hardware inaccessibility and this project aims to address that by building the vehicular edge network of internal Electronic Control Units (ECUs) using 'Raspberry Pi' System on a chip boards. Unlike other automotive projects, this project does not require specialist hardware to develop for. This involves:

- Assembly of a low-power, high-availability, prototype hardware internal car network of ECUs.
- Configuration of hardware and software modules to support the appropriate technologies used to simulate the fog environment.

By having all of the participants connected to the Fog network, a Fog service would be able to process transmitted information from the vehicles on the road and act upon it by sending location-agnostic notifications to the vehicles in the area of the accident, alerting them of a possible danger to avoid further multi-vehicle incidents.

1.3. Scope of Project

Creating a Fog network is a broad research area that can be explored to a great extent and degree of complexity. With respect to the project time constraints, it is vital to limit and focus the scope of the project towards solving the major problems.

- The outcome of the project does not aim to create a fully functioning, production-ready application, but the focus will be mainly on devising and developing the foundation of the main features in an extensible and modular architecture to enable car manufacturers, developers, automotive enthusiasts and researchers alike to build on top of this basis and simulate and test different environments and scenarios.

- In order to meet the project demands a connected car simulator will be developed with respect to the most relevant technologies in connected and semi-autonomous cars of today.
- The project does not make use or have access to any physical vehicle hardware and will be based on research into the workings of such technologies and simulating them using readily available hardware devices for the purposes of demonstrations.

There are currently no existing solutions to the problem of complete software vehicle simulation for the internal car network. This is mainly attributed to the fact that each automotive manufacturers keep their internal car network development in-house and there are almost no attempts at outsourcing the technology. For vehicle developers and enthusiasts this is a problem because it leaves them no other choice but to reverse-engineer proprietary car networks to understand how they work.

The core component of the project would be the connected car simulator used for simulating an internal vehicle network and instrument cluster and communicate with a Fog node to receive valuable traffic and road information. The purpose of the simulator is to create a platform with which automotive manufacturers or developers can simulate and test the behaviour of a connected car influenced by external events. They can use the platform to quickly prototype and test various connected car use cases in the field of car safety such as collision avoidance and any other future developments in the vehicle services space.

1.4. Project Importance

Having said that there are currently no existing vehicle network simulators and all current technology is proprietary, leaving developers with no options, which results in stagnation and an unused potential for the technology and the industry in general. This is a major factor and the automotive industry can be of benefit from such a tool. Open source vehicle software projects such as Automotive Grade Linux (AGL)[4], which aim to standardise automotive technology, have gained major adoption and are currently being deployed in newly developed vehicles by some automotive manufacturers such as Toyota[5].

Another major importance that this project has is demonstrating the unrealised potential of road safety developments. By notifying and keeping track of participants' locations on the road, and notifying vehicle drivers of possible incidents, this project plays a role in the work towards "Vision Zero"[6], to eliminate all accidents on the roads and consequently save lives.

Furthermore the project aims to help set a basis for future advancements in the field of Fog and Edge computing by providing a practical example for the implications and possibilities of this newly emerged technology.

1.5. Audience

The intended users of the simulator and implementation are connected car manufacturers, automotive developers and hobbyists that require a tool to simulate an internal car network, visualise car parameters and create new fog network scenarios and use cases. An assumption is made that the users have technical abilities and are familiar with car and network topologies in general, having experience in deploying custom software environments.

Whilst there is no age restriction as to whom can use the prototype, it is necessary for them to have the technical abilities and understanding of the programming languages, the software is written in.

The benefit for the target user is that this simulator will provide a platform which will make it easier to rapidly develop and prototype any new connected car services and features.

1.6. Approach

The project can be split into three major parts: ‘Connected Car simulator’, ‘Instrument Cluster’ and ‘Fog Network’. By doing this, it becomes easier to define the problems each of the parts poses and separating the features the system needs to support. Consequently, I have found that the software methodology that best fits the development of this project is Feature Driven Development (FDD).

FDD is an ‘Agile’ methodology that can be represented into five stages, beginning with developing a base-overall model, setting a features list, planning the project by the separate features and ending with incrementally designing and building feature by feature[Fig. 1].

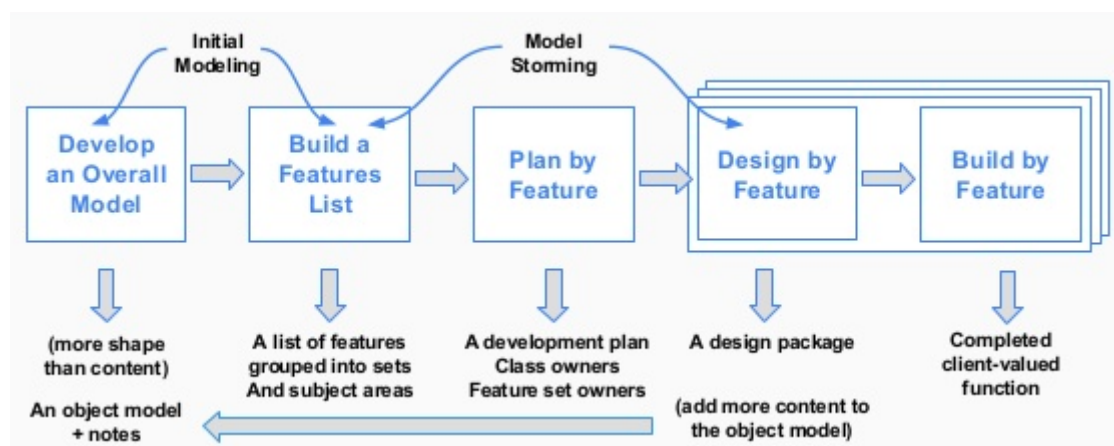


Figure 1: ‘Feature Driven Development’ cycle (FDD)[7]

This methodology best compliments the aim of the project: to enable an extensible architecture and application, by making use of multiple short development stages, it allows

for reducing costly and lengthy changes in the requirements during project development which is usually a pitfall in other development methodologies. It is essential, given the short time frame, for any changes in requirements to not majorly affect the development process and consequently, the outcome of the project.

Being an 'Agile' methodology FDD usually encourages that there are multiple members in a team that collaborate, each with their own domain of expertise, to structure a high level scope of the application. However, being an individual project, these aspects of the model could not be followed and the overall model was formed using the specific requirements and use case of the system at hand.

By separating the project into separate, individual parts, a high-level overall model and features list of the system was formed. This was done by creating sequence and flow diagrams of the system, describing what the basic parts are, participants, end points, processing nodes and listing their interactions. This provided a comprehensive feature list grouped into subject areas and subsets. A development plan was then carried out to assign a time frame in which every individual feature would need to be completed and with respect to the available project time span.

Following from there, the order in which the features of the implementation would be conducted was chosen and was influenced by the difficulty of the task to solve. The next step was designing and implementing each separate part of the feature set from the aforementioned stage to create an overall system. Testing of what was developed was also necessary to assure that other components of the system would not need changing and to flow any new content and ideas back to the overall model. During development there was a lot of new content that could be added but the time frame of the project could not allow for their development which was put into the Future Developments section. In a team, the feature development load would be spread out and would be more flexible to any similar changes.

To track the development progress, the 'Trello' task organization and management platform was used which allows for creating the tasks required by the feature list, assigning them to the main sections, setting percentage completion and time-frames which, as a whole, allowed to better judge the workload. By having regular meetings with the supervisor throughout the project and keeping them updated on the progress, receiving feedback and getting guidance, keeping on top of the project was possible.

1.7. Assumptions

The project adheres to the following assumptions:

- That the target audience have technical abilities and are familiar with vehicle and network topologies.
- Fog and Edge Computing is a new area of study and there is no existing conclusive architecture for supporting vehicular communications and the work in this paper is focused on the implementation research from various fields such as cellular communication.
- The project implementation assumes there is a GPS unit to provide location data to the simulator.
- There is an active wireless connection between the Fog node and Edge nodes.
- There is an active bridged ethernet connection between the car Instrument Cluster ECU and the internal network ECU.
- The project assumes that the system is configured correctly on implementation testing.

1.8. Project Outcomes

The main outcome of the project is most notably the implementation, as it is one of the few demonstrations of how this new arising technology might be put into practice. With powerful processors and hardware becoming ever so ubiquitous and accessible, to even fit in our pockets, being constantly connected, sending and receiving data, begs the question, why the same is not yet done for vehicles.

With some joint effort and standardisation of technologies, and automotive manufacturers being more willing to cooperate with each other, there can be true connectedness in the automotive industry. Smart services similar to the one prototyped in the implementation of this project can become part of our everyday lives and improve our living conditions.

2. Background

2.1. Problem Context

The current state of cloud computing is facing many challenges to keep up with demand and ensuring a good quality of service, so cloud service providers are forced into searching for better solutions to support the future connected society. There is much research on improving cloud computing performance and throughput but in the near future it is predicted for a normal user to consume gigabytes of information per day [8] and with the increasing number of connected devices, the current network infrastructure will be facing difficulties. It is estimated that there are over 1.2 billion motor vehicles in use worldwide. By 2018, connected car shipments are expected to hit 48 million units[9] and will continue to increase in an alarming pace every year. The network infrastructure and roads of the future need to be equipped to support this fast-approaching problem.

Instead of moving big data for the cloud to process, Fog and Edge computing provides a novel approach where computation is brought closer to the source, in micro data centres located right at the edge of the network. It is expected such architecture to be embedded into the already well-established communication infrastructures, in road-side units and cellular towers as well as in vehicles themselves[10]. This method allows for a drastic reduction in communication latency particularly in time-sensitive services such as navigation-related services, car sharing, smart city applications, traffic monitoring, vehicle safety systems, emergency assistance, autonomous driving and any other current and future capabilities which would otherwise heavily rely on cloud or back-office connectivity.

Furthermore, British road traffic related death and casualty rates have been in stagnation for the last 7 years as reported by the ‘Department of Transport’ [11]. This is largely due to the fact that no new technologies have been introduced as mandatory in vehicles. With the introduction and mandate of airbags and safety belts in vehicles, public roads have seen a tremendous decline in casualties. A new technology is necessary to bring the fatality rates further down or even eradicate them.

This project aims to evaluate the advantages and limitations of using Fog and Edge computing in tackling the problem of sending real-time traffic and incident information to support the safer roads of the future by creating a connected car simulator in a Fog and Edge environment and address the different complexity and security challenges associated with the approach.

2.2. Constraints

While we might imagine and dream of a world without road incidents, this will not be easy to achieve and comes with some constraints:

- The time required to develop this novel approach to a production-ready solution can and most likely take much more time and effort than is available in the scope of this project. This is why the main focus will be on establishing the basis in an extensible manner and working towards building on top of it in the future.
- Specialised hardware that is currently being developed and offered by Hewlett Packard to support enterprise Edge network systems[12] is available and would be best to simulate and test fleets of vehicle connections, however this is not within reach or accessibility of the personal financial investment and budget for the project and will not be necessary to test the conceptual features of the system.
- Fog and Edge computing, being a new and arising technology has not yet been widely adopted and is only recently becoming more known. It does not have a vast amount of research available and scientists are just now conducting comprehensive surveys on the technology.[44]
- For the past few years new cars come equipped with the capabilities of communicating to cell towers via popular 3G/4G networks[13] However, to deploy the implementation to existing cell tower and road-side units, would require a costly investment by governments in order to make this technology available to the public. This includes installing micro-data centers or small servers in base stations near all public roads.

2.3. Theory

The concept of Fog computing suggests that Fog nodes represent micro-data centers or servers installed near the source of information (edge of the network)[14] such as cell towers and road-side units and have the ability to process large quantities of sensor and device data in real-time environments. The Fog nodes serve as an intermediary between edge nodes (in this case connected vehicles) and backbone, cloud infrastructure, receiving, processing and forwarding only concise, insightful information to the cloud systems for further processing. Cloud and Fog computing provide the same storage, applications and data to end-users, however Fog computing differentiates with its better geographical distribution and proximity to end-users.

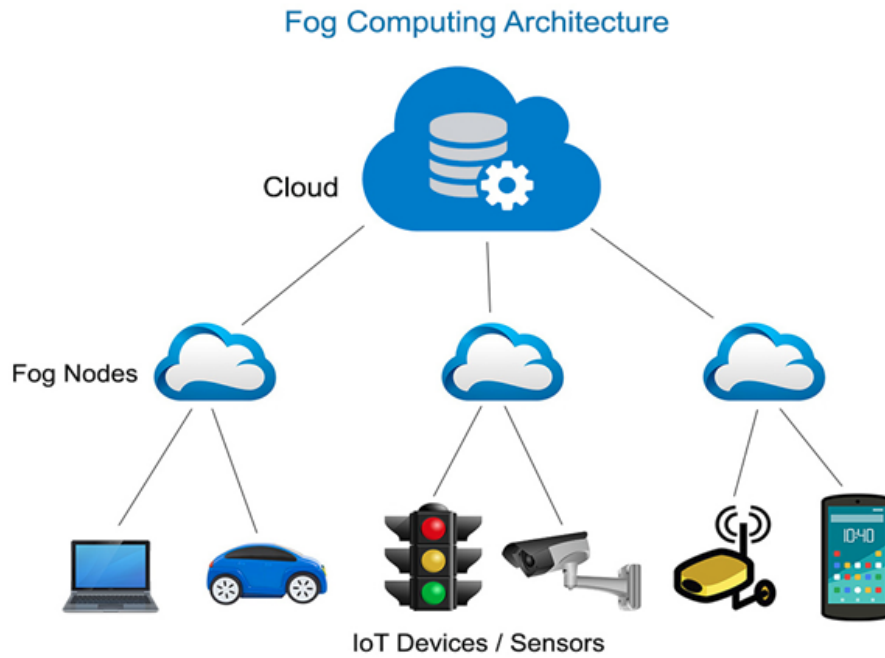


Figure 2: Fog Network Overview[15]

Compared to the cloud, Fog computing provides an intermediate level of computing power, in a less dense and medium-weight package. This new coined method alleviates the network load from congestion down the enterprise infrastructure lines and provides a novel approach at solving the problem of big data.

2.4. Existing Solutions

As of writing this report, there currently exist no commercial nor open-source solutions that are trying to tackle the problem of simulating future connected car services in a Fog and Edge environment. However, there exist enabling technologies that aim towards establishing Vehicle-to-Vehicle (V2V), Vehicle-to-Infrastructure (V2I), Vehicle-to-Everything (V2X) communications that lie under the common banner of Vehicular Communications Systems (VCS) and try to solve the problem of eliminating road traffic incidents, which is also relevant for the scope of the project. They are not commercial products but rather, well established development standards, however they do not completely solve the problem of road safety on their own due to their design specifications or usage and more about that will be delved into below.

Vehicular mesh networks are a key technology to cater the needs of transportation data sharing by connecting road participants and neighbouring infrastructure in transient, ad-hoc networks[16].

2.4.1. IEEE 802.11p and Dedicated Short-range Communications (DSRC)

The de-facto standard for V2X is Dedicated Short Range Communications (DSRC) wireless technology, which is based on the IEEE 802.11p standard, the Wireless Access in Vehicular Environments (WAVE) protocol in the U.S., and the European Telecommunications Standards Institute (ETSI) Intelligent Transportation Systems (ITS) European standards [17]. Moving objects have always posed a problem for any type of wireless connection and this is especially true for vehicles that travel at high speeds, as the communication link between them and roadside infrastructure might exist only for a short period of time. 802.11p does not need to establish a basic service set (association and authentication) before beginning to transmit and exchange data. It operates in the licensed ITS band of 5.85-5.925 GHz and is an enabling wireless technology for short-range data exchange in V2X communication.



Figure 3: Future V2V and V2I Communications[18]

- The benefits of the standard are that a specific spectrum of radio frequency has been assigned for it to operate without signal interference from other wireless communications.
- The downsides are that there is trouble in achieving consensus for the spectrum available for this technology and thus the range of the communication varies. For instance, communication range is 30 meters in Japan, 15-20 meters in Europe and, 1km (max) in USA and the throughput is simply not enough for any other applications apart from safety-critical ones[19].
- The other immediate limitation of this technology is apparent, it requires specialised hardware to operate and thus increases the cost to produce.

2.4.2. Cellular Networks and 5G

Widespread connectivity demands have made cellular networks commonplace and has pushed carriers to improve hardware and increase bandwidth. This makes the technology a primary enabler for vehicular communications. Cellular networking facilitates parallel data streams that would allow a vehicle or user not only to consume content but also have free, additional bandwidth for use in time-sensitive, safety-critical applications.

5G (5th generation wireless systems) is the next step in mobile communications, and the supposed bandwidth that 5G might offer could be upwards of 1Gb/s up to tens of Gb/s. 5G not only brings improvements to bandwidth, but also a suite of new technologies: ‘small cells’, ‘millimeter waves’, Massive Multiple-input Multiple-output (MIMO), beamforming and full duplex[20].

The combined implications of these technologies make 5G a prime enabler in the future V2X connectivity and will become a key growth driver. Future connected cars will have the ability to exchange sensor data, traffic information and multimedia content with each other using the mentioned low-latency technologies as well as communicating with the existing infrastructure. The aim of the technology is to provide complete connectivity between devices, vehicles, infrastructure and open the path for previously unseen and beneficial use-cases.

The downsides of this new communication method are that deployment is just beginning around various cities in the USA and has just recently passed tests in the UK. It is predicted it will not see widespread adoption until early 2020’s[21]. It will not be until the middle of 2025 that we see a high percentage of vehicles and users with 5G enabled device transceivers. Another pressing issue, similar to IEEE 802.11p, is establishing the consensus around the frequency bands available to 5G in different regions around the world[22].

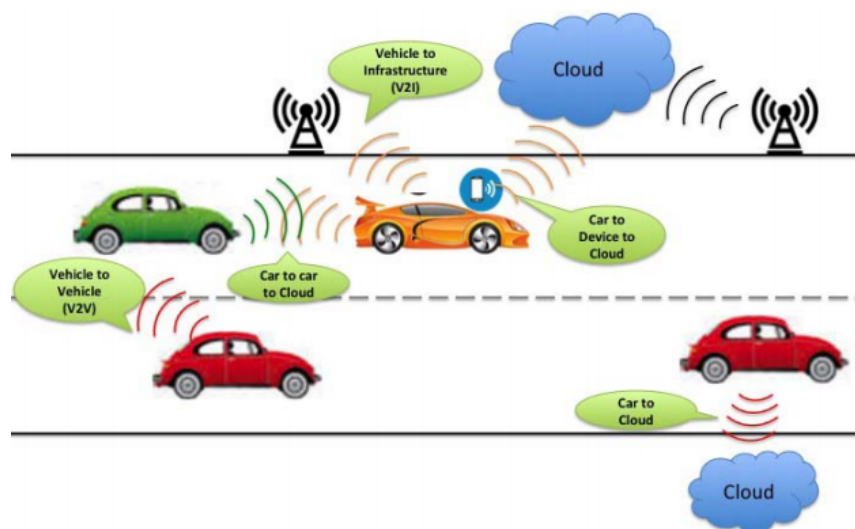


Figure 4: Future 5G V2X connected network[15]

This project assumes making use of the existing cell tower infrastructure which is 4G enabled (See Assumptions section). With the evolution of the technology into 5G, the use case of the project has the possibility of performing even better, having the ability to not only rely on cellular infrastructure but also send signals in an ad-hoc manner to other road participants, acting as a relay of information. The future implications of 5G are impressive, however this does not mean that existing cloud-based services will be able to cope with the load carried over consistent gigabit connections and data used by every vehicle and user. As can be seen later, use case examples do not always require a back-office connection in a Fog and Edge environment.

2.5. Methods and Tools Used

In order to create a functional system, several tools, methods and technologies were used in the development of the system which were selected from the requirements and design specification section for creating a simulator of a connected car in a Fog and Edge computing environment and the following section will provide some background information about those technologies, necessary for understanding the implementation.

2.5.1. Controller Area Network (CAN) Bus and SocketCan

All modern vehicles use an internal network to facilitate communications between microcontrollers and sensors without the necessity of a central host computer (SCV), namely the Controller Area Network (CAN) Bus. Communication inside the vehicle, between control units, sensors and electronics alike is critical for the operation of the vehicle.

The modern automobile could contain as many as 100 Electronic Control units (ECU) in order to manage various subsystems which can include the drivetrain, airbags, power steering, windows, mirrors, anti-lock braking and the list goes on[22]. In some cases independent subsystems are formed but the majority need to communicate amongst each other and may need to receive feedback from sensors or send signals to actuators. This networking of the car has allowed for a wide range of vehicle systems to be developed in software that would otherwise be complex and costly to implement if they were ‘hard-wired’ using traditional automotive electrics[22].

The bus is a simple Data Link layer, part of the OSI (Open Systems Interconnection) model[23], there are no MAC or Node addresses, ARP, Routing etc. The CAN follows the simple bus network topology where all participants are connected to the same wire and they can be either both listeners and/or senders. There is no security implemented or specific protocols used and CAN frames (messages) are simply broadcasted, can operate in loop mode making them cyclic and are also multiplex (containing an ID for specifying different data payload types)[24].

The CAN frame has a very basic structure. A single packet mainly contains a CAN identifier (CAN-ID, 11/29 Bit) used for content addressing, Data Length Code (DLC, 4 Bit) that specifies the number of bytes of payload data transmitted, and a Data (0 to 8 Bytes long) field containing the payload as seen in Fig. 5.

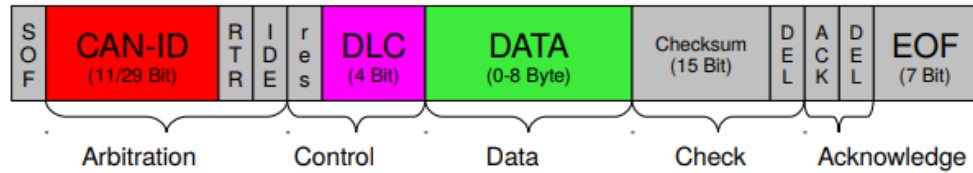


Figure 5: CAN Bus Frame structure [24]

As of Linux kernel version 2.6.25 there exists a virtual CAN network device driver: SocketCan[25] which eliminates the need for real CAN hardware and allows for creation of virtual CAN interfaces [Fig. 6].

```
vcan0: flags=193<UP,RUNNING,NOARP> mtu 72
unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 txqueuelen 1000 (UNSPEC)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

pi@raspberrypi:~ $ ifconfig
```

Figure 6: CAN Bus virtual interface in a Linux environment

SocketCan uses the PF_CAN protocol family[25] specifically defined for creating virtual CAN interfaces and is built on top of the Linux Socket Layer[26] which allows for using of the well-established socket programming interfaces in the operating system, the existing network driver model for networking hardware (e.g Ethernet cards) and protocols as well as routing inside the OS e.g. TCP/IP protocols[24]. The project later shows in the implementation section, bridging of a CAN socket with a regular TCP/IP socket. Raw packets can then be directly captured and processed at the application level.

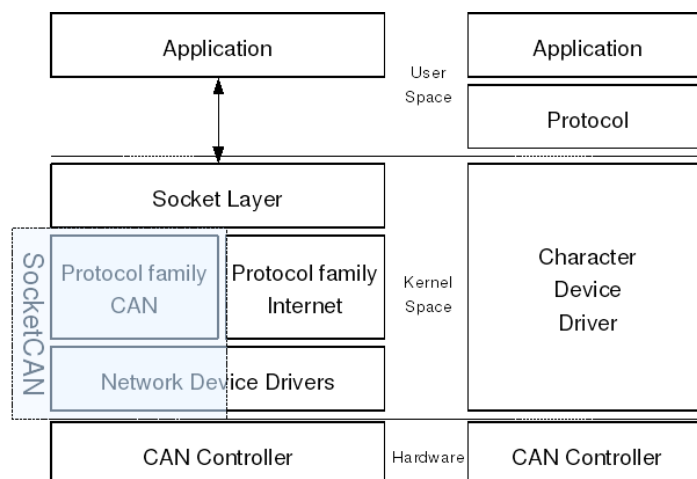


Figure 7: SocketCAN communication layers [25]

By having access to raw CAN frames, programming becomes similar to regular, known programming interfaces, using the CAN identifier to filter frames, working with their timestamps and allows for easy software deployment and migration, asynchronous reading and writing for multiple independent applications.

To make programming and development even more convenient many tools and libraries have been developed with the official suite being from the Automotive Linux Foundation: 'can-utils'[27]. The CAN Bus Simulator, described in Section 3.2 was built using Python3 and a library called 'python-can'[28] for simpler and more readable code than using raw frame filtering and parsing.

2.5.2. Qt and QML

Another one of the tasks set out to be completed in the scope of the project was to visualise the problem being solved. This meant extending the data the connected car produced from the CAN bus and Fog network into an automotive and familiar graphical interface which is the Instrument Cluster as described in Section 3.3. Cars these days make use of many embedded devices for all types of purposes, this includes speed, temperature, oil, tire pressure, other sensor information and in general, the state of the vehicle. The device which displays mission-critical car information to the driver in a nice, visual way is called the Instrument Cluster as seen in Fig.8.



Figure 8: Visual Representation of the Instrument Cluster

In more recent years car manufacturers have started using digital display technology for their instrument clusters [45] which allows for infinite configuration in design and functionality and is only limited to the features the car supports. This information is usually concise and is displayed in a visually appealing and understandable manner. However, using and sourcing specialised embedded devices and controllers is difficult for developers such as small car manufacturing companies or general hobbyists willing to develop for vehicles. These devices

come with proprietary drivers, kernels and are not ready to develop for out of the box, which makes it a time consuming and cost ineffective process.

In the search for the right language to develop the initial version of the Instrument Cluster, I found existing Python-based GUI libraries such as PyGame to be cumbersome. Web visualisation languages such as jQuery were promising, however when it came to working with native sockets and listening to the constant flood of data and parsing it in real time crashed the target hardware's browser. Another issue was that web technologies are not readily supported by embedded devices. C++ gave promising results but the visualisation libraries lacked in quality and were difficult to work with. Continuing the search gave fruit and Qt and QML addressed exactly these issues.

Qt is a cross-platform application framework and widget toolkit for creating embedded graphical user interfaces and applications that can run on any software and hardware platforms with little to no changes on the underlying codebase, while still maintaining a native application with native capabilities and performance[29]. This is especially useful for developing applications on low-power devices such as the Raspberry Pi this project is based around which do not have a typical desktop system architecture. Qt also supports cross-compilation for target platforms such as the Raspberry Pi, so compilation can be done on more powerful, external hardware. For the development of this project it would have taken a lot more time to compile the software on the low-voltage processor of the Raspberry Pi.

Qt is written in C++ and the programming languages it officially supports are C++ and QML as well as third party language bindings for Python, Rust, C#, NodeJS and others. Qt provides a C++ class library with rich application building blocks for C++ and is specialised in giving the developer the flexibility and communication with advanced GUI developments. QML is a declarative, JavaScript-based language for describing user interfaces of programs. A UI is built almost in the same fashion as how a JSON object is defined, with objects and properties. The benefits come in the usability of the language as HTML, CSS and JavaScript knowledge is enough to create complete applications. The language can be extended with C++ to allow for more advanced developments.

2.6. Hardware

The hardware that will be used in the making of this project, as stated in the Constraints section, has to be cost-effective, readily accessible and adhere to the requirements the problem. Two ‘Raspberry Pi’ SoC development boards will be used for replicating the connected car internal network. A generic laptop with a Linux operating system will be used for creating the Fog network to which the ‘car’, represented by the Raspberry Pis, will connect to. The hardware has been purchased from Adafruit[30], a popular computer and IoT hobbyist online shop and has been configured using the documentation available on the official Raspberry Pi website [31].

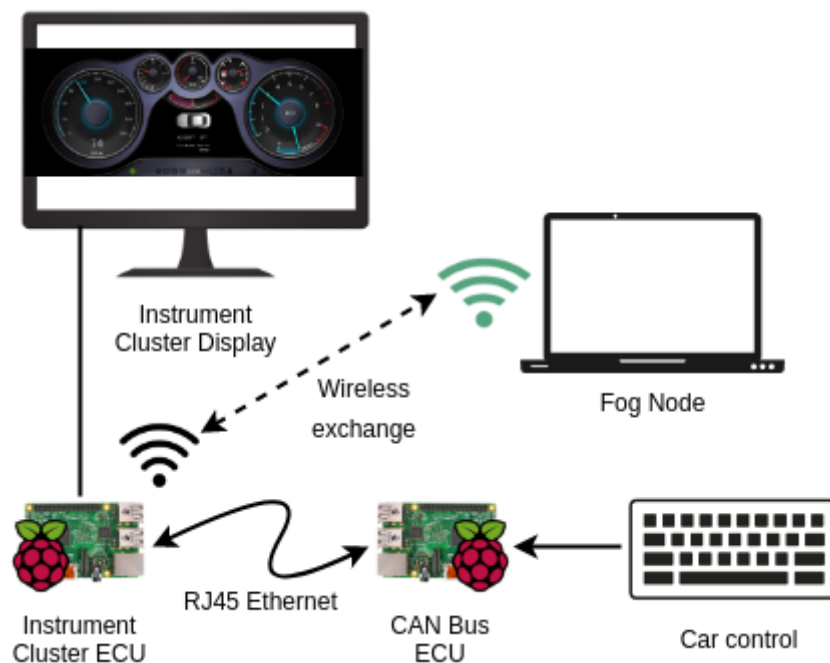


Figure 9: Hardware layout of the simulated system

The list of hardware equipment is as follows:

- A Raspberry Pi 3 Board (WiFi enabled) (Represents the Instrument Cluster ECU)
- A Raspberry Pi 2 B+ Board (no WiFi, Ethernet bridge) (Represents the CAN Bus ECU)
- A Laptop
- 2x MicroSD cards for the Raspberry Pi's
- 2x MicroUSB power supplies for the Raspberry Pi's
- An Ethernet cable to bridge the Raspberry Pi's
- An HDMI enabled Monitor to visualise the Instrument Cluster ECU
- A keyboard to send commands to the CAN Bus ECU

In a real connected car, there is usually more than one CAN Bus but for the purpose of this prototype there are no inherent advantages or disadvantages to using more than one.

2.7. Research Questions

The overall aim of this project is to understand the concept of Fog and Edge computing and provide a prototype recommendation for a network architecture in which vehicular edge nodes can operate and receive application specific information in the field of vehicle safety, without the need of a cloud service and evaluate the feasibility of this approach. The research questions this project is attempting to answer mirror the aim of the project.

The first immediate question is, what sort of data will be exchanged within the system?

- It is necessary to research the types of parameters and information that would be found in an internal vehicle network to answer this question. Having an appropriate idea of the data exchanged within the environment makes the selection of components and requirements for the system possible.

The next question is a direct consequence from the first: How can this data be simulated as closely as possible?

- To answer this, insight into the participants of the system is needed. By knowing that the target end-devices are connected vehicles, we can best simulate the parameters and information by creating a connected car simulator. This project answers the two questions by doing exactly that.

The last question comes from the general requirement, to create a fog network: How do we evaluate the prototype network architecture?

- By not only simulating the participants, but also prototyping the Fog network architecture itself we can run tests to evaluate the performance and implications of a possible similar, real system.

3. Specification and Design

Prior to beginning development towards the implementation of the project, a comprehensive specification and architecture design is needed. With the intent of being extensible and modular, the development approach needs to be carefully planned. The following section addresses the requirements, design and architecture of the system and additionally discusses and justifies the decisions taken during this process.

Apart from looking at the system as a whole, in order to more thoroughly assess the capabilities and design aspects required, a closer look at the individual component requirements for the project must also be carried out. For ordering and prioritisation of the system requirements, the MoSCoW methodology[32] was used which splits requirements into four categories: Must have, Should have, Could have and Won't have.

3.1. System Overview

As initially said in Section 1.1, the system can be split into three Major components. Each individual component has its characteristics and distinctive features and can function as their own separate entities. In order to solve the problem of Fog networking and vehicle notifications, these components must function as a whole.

3.1.1. Requirements

To define the purpose of this system and what it does we first need to specify the overall requirements and give supporting justifications.

Functional Requirements

Must Have:

- ❖ Facilitate a Fog network environment
 - This comes from the aims of the project, to have a Fog & Edge computing network.
- ❖ Simulate a connected vehicle in a Fog network
 - Again the requirement comes from the use case and aim, to emulate a connected vehicle as an edge node in a fog environment.
- ❖ A method to simulate a collision
 - In order to notify existing participants of the network, a collision has to be simulated.

Could Have:

- ❖ Sending vehicle diagnostics information to a cloud service

- In a typical fog and edge environment, the fog nodes can aggregate data and send it to cloud services for further assessment.

Should Have:

- ❖ Encryption and authentication methods in place
 - The consequences of a breach in a system which transports people could be catastrophic, hence security measures should be put in place.
- ❖ A visual representation of the vehicle information and status
 - It would be beneficial to have a display of the internal vehicle information so that actions and changes can be visually perceived.

Won't Have:

- ❖ A fully working, deployable system
 - The constraints (See Section 2.2.) of the system give a clear justification as to why this is not feasible.

Non-functional Requirements

Must Have

- ❖ High throughput
 - The system needs to function even at high load. This is needed in order to satisfy the use case.
- ❖ High availability
 - The use case does not permit components having less than 99.9% downtime.
- ❖ Fault tolerant connection
 - The connection between the fog network and the CAN Bus must continue even though there is a problem with a component.
- ❖ Seamless handover
 - Upon disconnection from a Fog network node, the Instrument Cluster head unit must reconnect to a subsequent network node.

Could Have

- ❖ Modular and extensible design
 - To allow future expansion of the system.

Should Have

- ❖ Language agnostic interfaces
 - So as to interface the system from any language without using proprietary and special interfaces.

3.1.2. Design and Architecture

Following from the above mentioned requirements and especially the modularity aspects of the system, it can be clearly seen that the system, at a very high level, would consist of at least two major components: the Connected Car Simulator and the Fog network. Throughout the project timeline, various design decisions were made which led to the eventual split of the

Connected Car Simulator component into further two more: the CAN Bus Simulator and the Instrument Cluster, which allows for even greater modularity of the system. This way, the individual components could function on their own, without hindering the rest of the system, however, in order to implement the use case, they must function together as a whole, linked system. Future expandability is also possible for the three individual components, again noting the independence of each component from the rest.

The initial design of the system was first envisioned to have a single Fog node and a single Connected Vehicle in the system. Over the course of the project, it went through several revisions and iterations to come to the final network structure.

To briefly summarize the structure and its iterations, UML diagrams have been used. Initially it was designed for the CAN Bus to be connected directly to the Fog Node as seen on Iteration #1 in Fig. 10, however this approach is not technically sufficient to support the use case of simulating a connected vehicle. The vehicle driver would also not have a visual reference of the inputs they are making such as pressing the throttle.

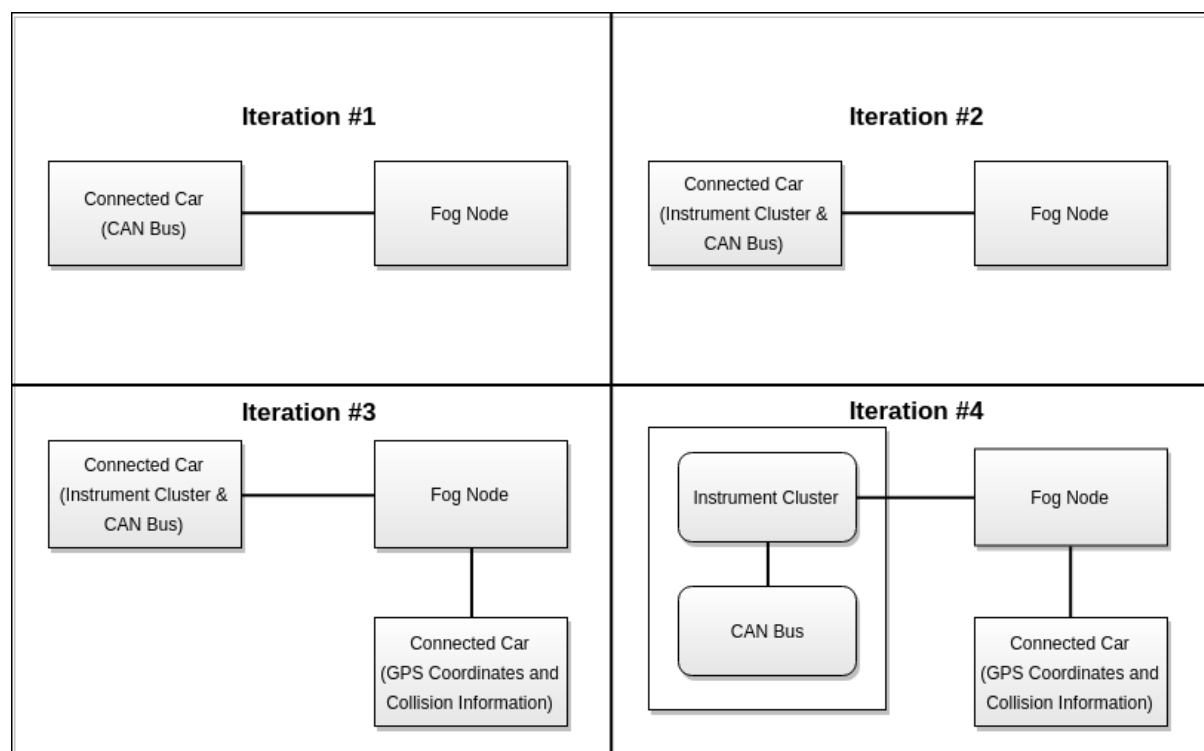


Figure 10: System design iteration steps

This led to revising the design and implementation for having an Instrument Cluster act as the intermediary between the CAN Bus and the Fog node which would also aid in visualising the information sent both from the CAN bus and the Fog node which can be seen in the second iteration of Fig.10. Further to improve the design of the system, in order to have an evaluation and testing method to see whether the vehicle receives notifications from other

participants in the Fog network, another connected car was added which sends location data as well as a collision alerts to the Fog Node so that the instrument cluster can be tested whether it receives the notification and displays it. This was addressed in iteration #3 of the same figure.

The final design of the overall system was influenced by all the factors mentioned in the previous iterations in addition to the security, integrity, modularity and extensibility requirements of the system. This meant decoupling the CAN Bus and Instrument Cluster integrations and making them separate entities, continuing operation even if one fails. Ultimately, if a component fails, the system will not serve its purpose as intended by the use case scenario, however other components will not seize operation. By separating the CAN Bus and Instrument cluster we also achieve the intent of simulating as closely as possible how a connected vehicle works internally, which, having made the research is exactly how an internal vehicle network functions. The Instrument Cluster is the only point of communication to the Fog Node and can only read information from the CAN. This way, it becomes impossible for an attacker from an outside network such as the Fog network, to reach and send malicious commands to the CAN Bus to override the vehicle functions and do any harm.

From the viewpoint of how the data flows through the system, a further diagram explains this below.

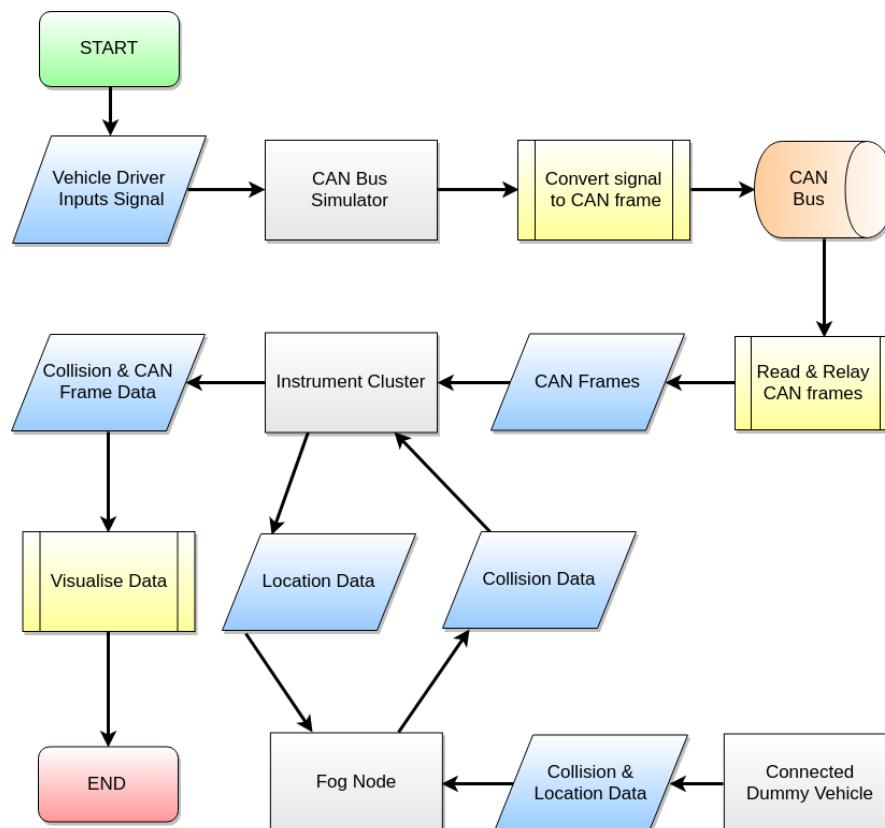


Figure 11: Flowchart of the whole system (Fog Node, Connected Vehicle and Collision Vehicle)

The general flow of the system starts with the Vehicle driver sending a signal to the CAN Bus simulator, this can be in the form of a pressing the pedal or sending a turn left/right signal. The CAN Bus simulator, listening for these signals converts them into CAN frames and writes them to the bus. A separate process listens for CAN bus frames and relays them to the Instrument Cluster. Once the message data arrives, it gets visualised to the vehicle driver. The Fog node part of the system receives location and collision data from another vehicle and having also received location data also from the Instrument Cluster (assuming there is a GPS Unit to provide it) determines based on these parameters whether a notification of a collision should be sent to the Instrument Cluster. If one is received, the Instrument Cluster will show the given notification on the vehicle display.

From an entity-relationship standpoint the system is designed as follows:

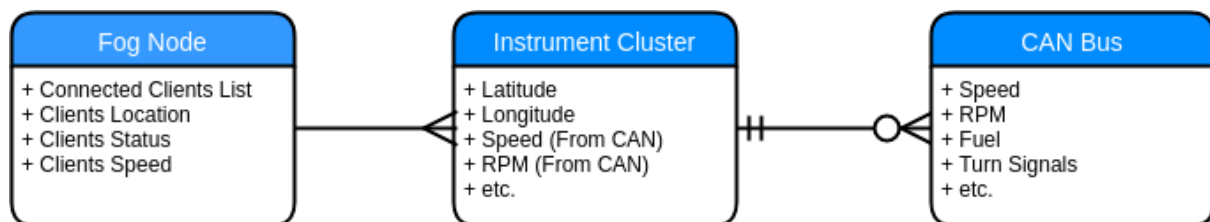


Figure 12: Entity-relationship diagram of the components

There is a one to many relationship between the Fog node and the Instrument Cluster which means that there could be many vehicles connected to one Fog Node. The Instrument Cluster has a mandatory one to one connection and an optional one to many connection with the CAN Bus. This means that there can be optionally many CAN buses used in the vehicle, while this is not a must requirement, the system should technically support it. In terms of what Data is facilitated in the different components of the system, the Fog Node contains a list of active vehicle connections and their properties such as location, status and speed where status can be whether the vehicle has been involved in a collision. On the Instrument Cluster, the Latitude and Longitude data points come from a GPS signal and are relayed to the Fog node. The instrument cluster also receives speed, RPM and other vehicle data from the CAN Bus, the CAN bus generates this data when signals are input by the vehicle driver. The list of vehicle, CAN Bus parameters, simulated is given in Appendix A.

3.1.3. Use Case

As mentioned in Section 1.2., the goal of the project is to create a use case of managing vehicle collision notifications in a Fog and Edge environment. The Following Use Case Diagram gives an overview of the events that the system has to support:

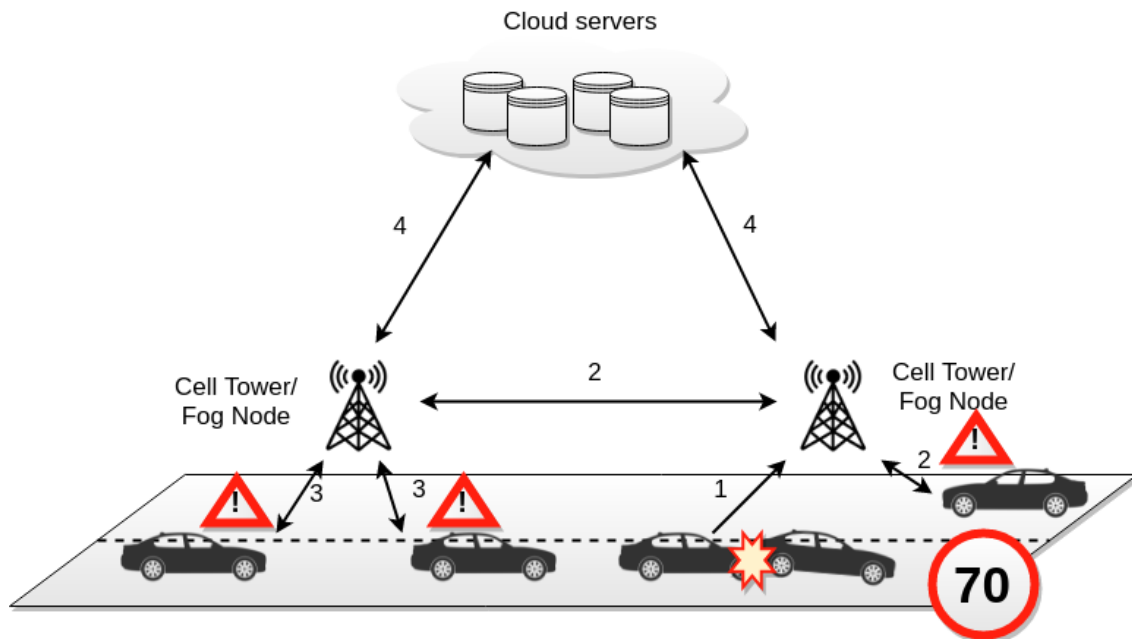


Figure 13: Diagram of a collision use-case scenario

When vehicles are travelling on a motorway at ~70Mph speed, and there is fog, rain, driving conditions are poor and visibility is limited, there are other vehicles upfront and braking distances are not short, multi-vehicle collisions can unfortunately occur. By notifying incoming vehicles of a potential incident, they can react and slow down their vehicles to avoid further accidents. In Fig. 13 when a collision has happened, the connected vehicles sensors send a signal to the Head Unit which subsequently notifies the nearby Fog Node of the incident and its location (1). The notification is then processed and propagated through the Fog network by the Fog node to alert nearby drivers of the accidents and sends a signal to their Head Units which subsequently display a warning message (2,3). After that is done, Cloud, back-office response teams can be notified of the incident and immediately send assistance (4). This way lives can be saved and the use case gives us the implications of the Fog and Edge computing notion.

The use case can be split into further two sub-use cases which would make it more clear as to what functionalities have to be later implemented in the solution.

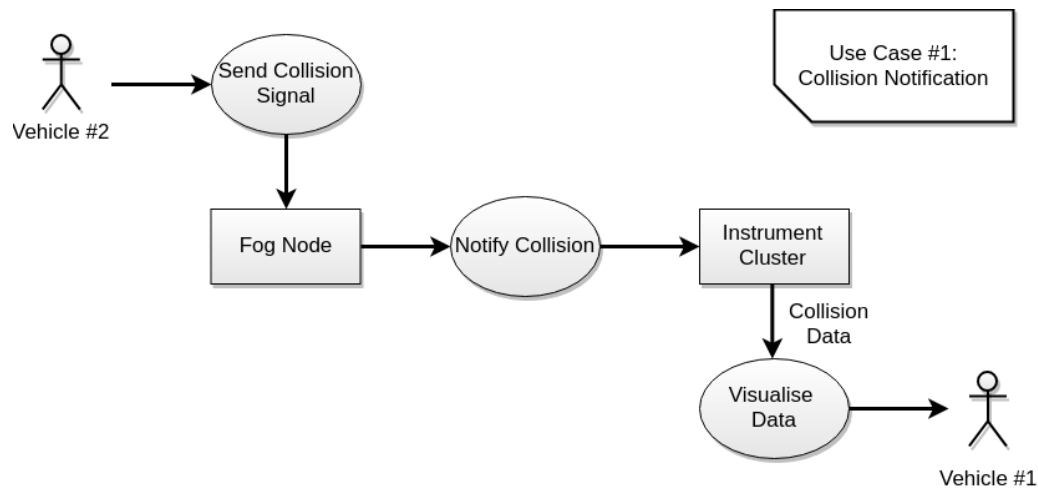


Figure 14: Use Case Diagram #1, collision notification (V2V)

The above use case diagram, Fig. 14, illustrates the collision notification functionality of the system. Vehicle #2 Sends a Collision signal to the Fog Node which then sends a notification to Vehicle #1's Instrument Cluster and subsequently visualises the data on the display for the vehicle driver to see.

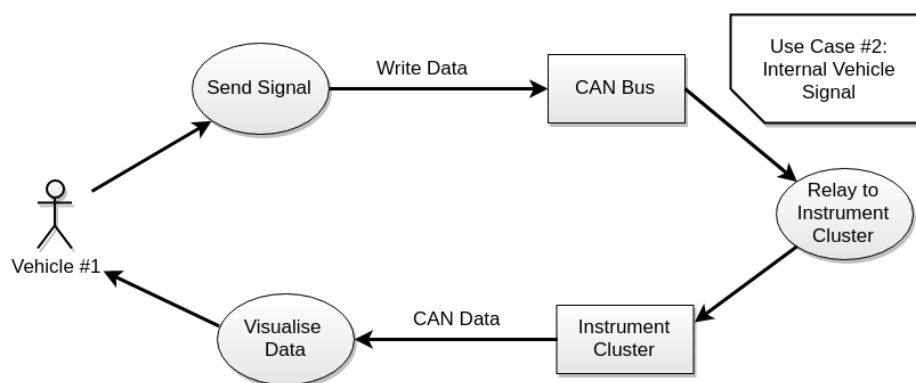


Figure 15: Use Case Diagram #2, internal vehicle signal

Use case diagram #2 demonstrates the event sequence, when a vehicle driver inputs a signal such as throttle, turn signal etc. The CAN Bus Receives the signal and relays it to the Instrument Cluster. That same CAN data is then visualised by the Instrument Cluster to the user. This use case comes from the general idea to simulate a connected vehicle and has been based on research made in the Background section as to how to structure the components in the system.

3.2. CAN Bus Simulator

The CAN Bus Simulator is a crucial component of the system as it emulates the internal network in the connected car, namely the CAN Bus. Below are the functional and non-functional requirements that specify what features and behaviour the Simulator has to support.

3.2.1. Requirements

Functional Requirements

Must Have:

- ❖ A virtual CAN interface
 - This comes from the intent to simulate the internal network of the car.
- ❖ Ability to send and read information from a virtual CAN interface
 - In order to be able to work with the CAN interface, we need to have read and write methods.
- ❖ Method to access information on the CAN Bus from an outside source
 - Justifying this is the overall system need for a visualisation method which should be able to connect to the CAN Bus Simulator.
- ❖ Authenticate outside connections
 - No other parties should be able to connect to the CAN Bus apart from the permissioned ones.

Could Have:

- ❖ Ability to read and write data from/to the CAN Bus asynchronously
 - This could be beneficial as we will be working with kilobytes per second data that can lead to problems if being overwritten multiple times.
- ❖ Have an On-board Diagnostics (OBD) port
 - This could be used for linking straight to the CAN bus but not necessarily writing to it.

Should Have:

- ❖ Encrypt data in transit being accessed from an outside source
 - The data should be encrypted in transit to prevent malicious attacks.
- ❖ Support up to 10 connections at any given time
 - In a real vehicle there are usually multiple ECUs listening to the same CAN Bus.

Won't Have:

- ❖ A direct visual interface for displaying the information or connections
 - In reality this is not necessary as the CAN Bus is 'headless'.
- ❖ The ability to communicate with the Fog network directly
 - This is a measure for security and to simulate a connected car as closely as possible.

Non-functional Requirements

Must Have

- ❖ High availability and no dropped connections
 - Whilst driving it is dangerous to suddenly have critical systems disconnect from the CAN bus.
- ❖ High throughput
 - The CAN bus must not be flooded with too much information that would make it inoperable.

Should Have

- ❖ Modular, allowing for future extensibility.
 - This is necessary to allow for future development.

Referring back to the Initial report (See Appendix C), some requirements have not been included as during the project design and development phases they have become redundant and have been listed in the Implementation Section 4.1.

3.2.2. Design

Having made the requirements for the CAN Bus Simulator clear, we can design the structure and define the functionalities it needs to support. The initial plan is to have every single component in the system function as a standalone entity. This means that components have to be loosely coupled and function independently.

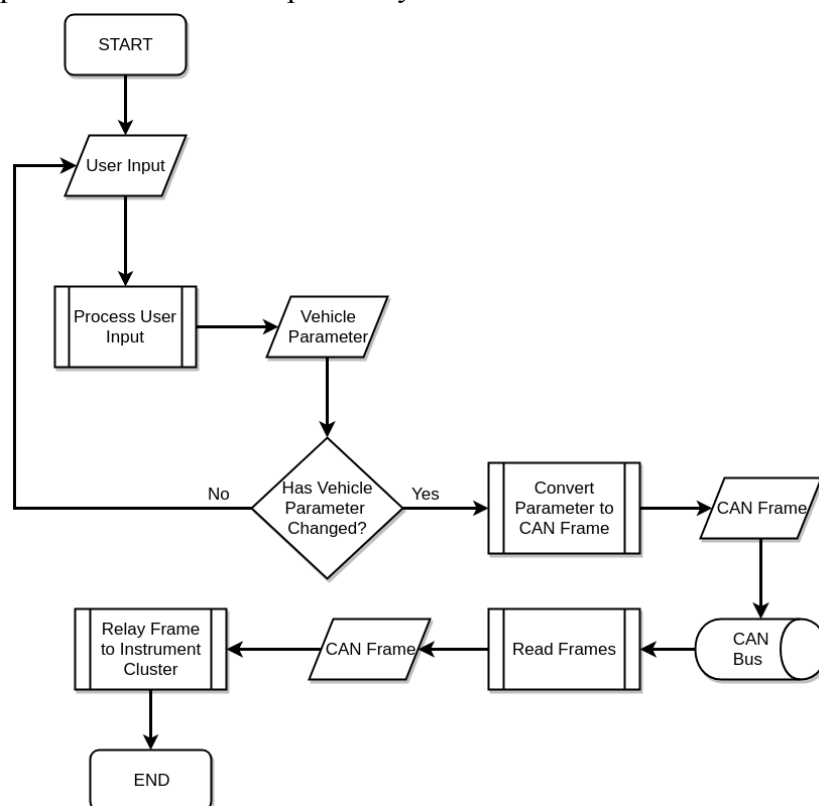


Figure 16: CAN Bus flowchart and data propagation

To best describe the design and functionality of the CAN Bus Simulator component, a flowchart has been made showing the different data, processes and decisions that it needs to make in order to fulfill the needs of the project seen. This diagram can be seen in Fig.16.

As defined by the Overall System design and the above mentioned requirements, the CAN Bus Simulator has to communicate with the Instrument Cluster and have an interface for the user in order for them to input data. As well as this, the Simulator must be able to process the user input against a predefined set of parameters that represent the internal functionalities the vehicle supports, which is also defined in the component requirements.

The flow begins by taking in user input which can be throttle, turn signals and any supported vehicle signals, which are then converted to their appropriate parameter counterparts. A check then runs to see whether the value of the given parameter is different. This is because inputs from the vehicle driver might be repeated without their value changing such as turn signal indication and the CAN bus can then become flooded with signals to above of its specified capacity. After the check, the parameter and its value are converted into a CAN specific frame and sent to the Bus. A listener inside the simulator for signals on the CAN bus will read the frames and subsequently relay them to the Instrument Cluster. The design decisions have been made in line with the component requirements as well as from how the CAN Bus physically works.

3.3. Instrument Cluster

The Instrument Cluster is a necessary component for the system in order to provide a visual representation of the information that is propagated through the network. Below is a list of functional and non-functional requirements it has to support.

3.3.1. Requirements

Functional Requirements

Must Have

- ❖ The Instrument Cluster must have the ability to connect and receive data from the CAN Bus.
 - This is necessary to support the intentions of having a visualisation method.
- ❖ It must be able to display the CAN Bus data in an appropriate and visually pleasant way.
 - The vehicle driver needs to be visually aware of the useful car information.
- ❖ The Instrument Cluster must be able to receive and send vehicular service information from/to the Fog network and display it nicely.
 - This comes from the use case, to be able to send positioning data and receive notifications of a Fog-enabled collision avoidance service.

Could Have

- ❖ The Instrument Cluster could have a method to specify information to be forwarded to a Cloud endpoint.
 - This could be necessary if the use case demands it.

Should Have

- ❖ The Instrument Cluster should have an encrypted and authenticated connection with the CAN Bus.
 - The data should be encrypted and authenticated in transit to prevent malicious attacks.
- ❖ The Instrument Cluster should have an encrypted and authenticated connection with the Fog Network.
 - The data should be encrypted and authenticated in transit to prevent bogus information from impersonators.

Won't Have

- ❖ The Instrument Cluster won't have the ability to send information to the CAN Bus.
 - The CAN Bus ECU should be the only party that can write to the Bus.

Non-functional requirements

Must Have

- ❖ Modular and extensible design
 - To allow future expansion of the Instrument Cluster.
- ❖ Integrity
 - The Instrument Cluster must not drop connections to the fog network when there are obstructions.
- ❖ High throughput
 - It needs to process all of the data it reads from the CAN Bus.
- ❖ High availability
 - The Instrument Cluster needs to be available as it displays all the vital vehicle information.
- ❖ Fault tolerant connection
 - The connection between the Fog network and the CAN Bus must continue even though there is a problem with a component.
- ❖ Seamless handover
 - Upon disconnection from a Fog network node, the Instrument Cluster head unit must reconnect to a subsequent network node without dropping the connection.

Could Have

- ❖ The Instrument Cluster could receive notifications and CAN data asynchronously
 - This would improve performance and time to display incoming data

Should Have

- ❖ Integrity at speed
 - It should not drop connections when the vehicle is moving

3.3.2. Design

Working towards the non-functional requirement of loose coupledness, the Instrument Cluster can again be defined as a standalone component, however in order to support the use cases it must connect to the CAN Bus Simulator and Fog Nodes. The following flow chart describes that connection and the intended structure and functionality of the Instrument Cluster.

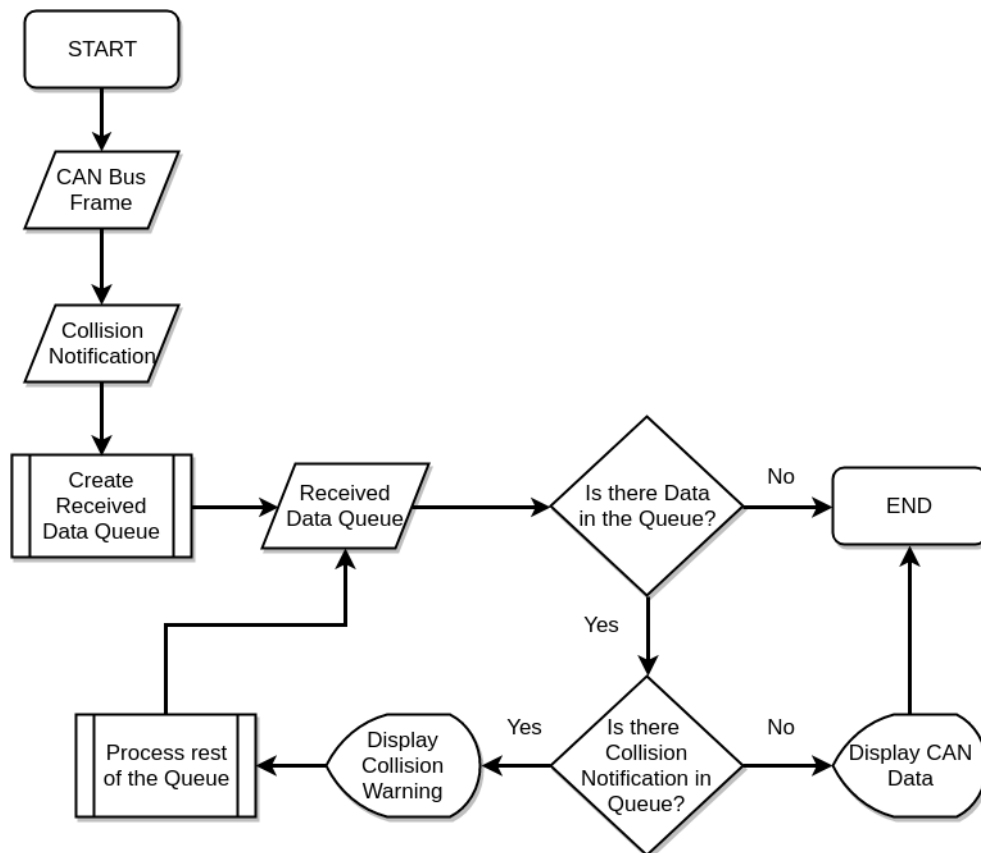


Figure 17: Instrument Cluster flowchart and data propagation

The decisions as to how exactly to depict the Instrument cluster were dictated by both the requirements and the aims of the project. Here, unlike the CAN Bus, the way the Instrument Cluster processes data is standard and does not need conversion. It should store the received CAN Bus frames and if any, collision or incident notifications and put them in a data structure of a queue as can be seen in Fig. 17. This queue of data is then iteratively processed until there is no data left.

The Instrument Cluster design has a decision check to see whether there is data in the queue and appropriately handle it by its type. Higher priority is given to the collision notifications and if they are detected, they get pushed to the front of the queue and visually displayed on the instrument cluster as seen on the diagram. The decision for this comes from the general importance of the incident notification itself. Data related to the CAN Bus is then fed back and rearranged so it is processed immediately after that and displayed appropriately

depending on its function. The approach assumes there is no asynchrony in the processing of the input data as this is a 'could' requirement. This design goes in line with the requirements of the Instrument Cluster, overall system functionality, aims and use case for the project.

Being a user interface, it needed to have visual cues as to what information is being displayed to the user. The approach chosen was to make use of the very familiar and popular vehicular instrument cluster which contains gauges that can be populated with the car data. When the CAN Bus data changes value it is reflected on the digital instrument cluster. The initial render of the digital instrument cluster, Fig 18. depicts less of the information that is actually implemented on the CAN Bus, later seen in the Implementation section. Design cues have been taken from existing vehicle instrument clusters.

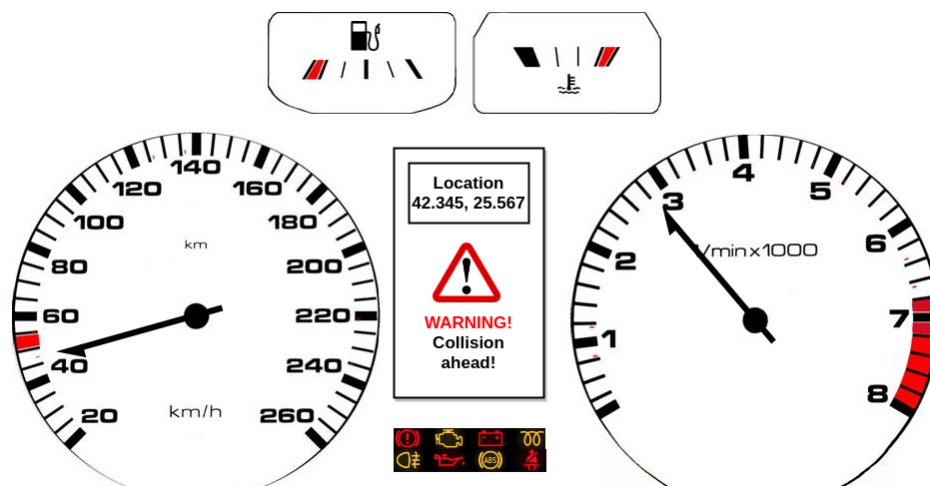


Figure 18: Instrument cluster design representation

The vehicle instrument cluster contains a visual reference about speed, engine RPM, fuel level, location data as well as information about vehicle status such as engine warning lights, ABS warning, oil temperature, light malfunctions, seat belt usage etc. What has been added on top of this design is the Warning Notification section in the centre of the display. This section is populated with information when the Fog Node sends the vehicle a warning about an incident in the vicinity. This tells the driver to slow down or be at alert for incoming dangers. This is especially effective at high speeds and low visibility of the road and other vehicles and can aid in preventing further collisions and casualties.

3.4. Fog Network

The Fog Network is another necessary component for the system in order to facilitate real-time vehicular communications. Below is a list of functional and non-functional requirements it has to support.

3.4.1. Requirements

Functional requirements

Must Have

- ❖ Semi-long range ambidirectional wireless connectivity (up to 500m-1km range)
 - The requirement comes from supporting vehicles that are constantly mobile and can change their proximity to the source.
- ❖ A method to send vehicular service information
 - A Fog node should send information to all edge participants connected to it as per the use case of the project.
- ❖ A method to receive vehicular service information
 - A Fog node should be able to receive information about collisions, and similar types of information as per the use case of the project.
- ❖ The network needs to have encryption and authentication methods in place
 - This is because the network is expected to be protected from snooping or impersonations.
- ❖ A method to store the connected vehicles to the node
 - This requirement comes from the need to know which vehicle to notify in the event of incident data received as defined by the use case.

Could Have

- ❖ Support for up to 1000+ edge connections
 - The network could have at any given time thousands of connections and it needs to be able to process their information.

Should Have

- ❖ Support a unified protocol
 - The network should make use of popular connectivity protocols such as TCP/IP

Won't Have

- ❖ A method to visualise connections to the Fog node
 - This is not necessary as it will be a command line application

Non-functional requirements

Must Have

- ❖ High availability
 - The system must have high availability and no dropped connections.
- ❖ Fault tolerant connection
 - When a fault in a component in the network happens, it should continue transmitting.
- ❖ Seamless handover
 - When a vehicle disconnects from a fog node, reconnection should happen seamlessly.

Could Have

- ❖ Integrity at speed
 - Support connections of fast moving vehicles.
- ❖ High Throughput
 - The network could have at any given time thousands of connections and it needs to be able to handle them.

Should Have

- ❖ Modularity and extensibility
 - The fog network and node should be written in a modular and extensible way.
- ❖ Signal Integrity in obstructions
 - The signal should not be lost when there are obstructions between receiving vehicle and fog node.

3.4.2. Design

As the technology is not yet mature, it is difficult to predict and estimate exactly which methods and tools would best fit the purpose of this new and emerging technology in production scenarios. By comparing and contrasting possible solutions, a few promising methods for supporting the creation of the prototype can be deduced.

To satisfy the requirement of wireless connectivity, a wireless network needs to be established with available and existing technologies. Wi-Fi [33] first comes to mind as it is well-established and there is lots of research made into its implications. It does not suffice at semi-long range connectivity as the technical range is up to 100m in ambidirectional mode[33], integrity at speed and fault tolerant connections. In other cases it is sufficient.

4G connectivity covers all of the problems Wi-Fi did not address however it is costly to operate and requires government authority approval to operate. In addition to this, the technology is not readily available for developers to access and would slow down the process of prototyping and rolling out the implementation. Another option is using Dedicated Short Range Communications (DSRC) however it does not satisfy the range and obstructions requirements and as well as not being readily accessible. This leaves Wi-Fi connectivity as

the only viable option for the sake of this prototype and in future, long term development, use of 4G or even 5G would be necessary.

The design of the Fog Network and more specifically, the Fog Node follows the overall aims, objectives and technology description of Fog and Edge Computing. The Fog node is usually described as an intermediary between the edge devices and the Cloud environment. With all this in mind as well as the requirements and assumptions for the system, the below Design flow chart was created as seen in Fig.19.

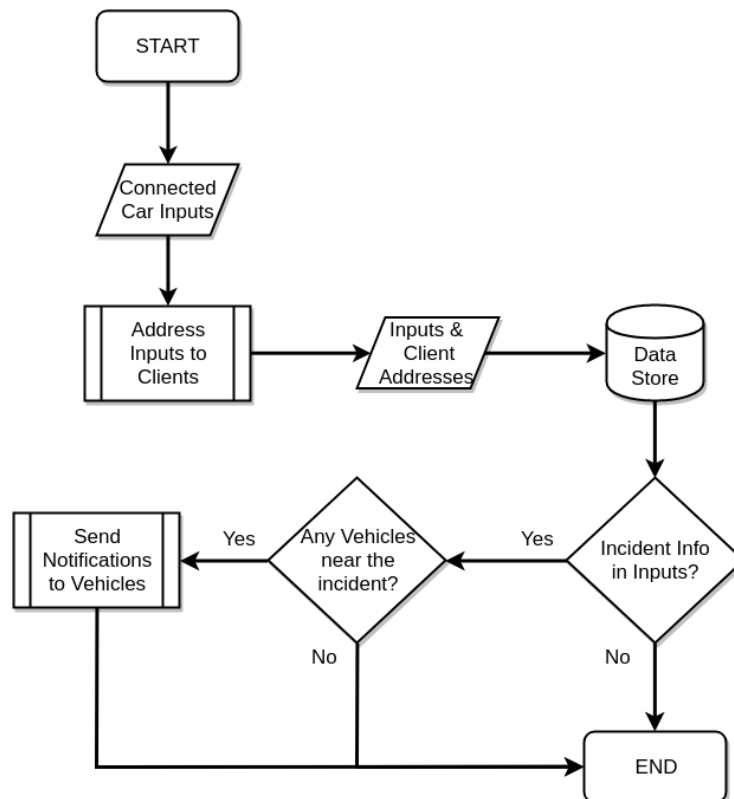


Figure 19: Fog node flowchart and data propagation

The flow chart describes how the Fog node receives inputs from connected vehicles which can be in the form of speed, location data, status notification, incident reporting etc. and uses the client's physical address on the network they are using for communication, mapping and identification to which participant this data belongs to. It then stores the data and checks whether one of the entries contains collision information or incident input. In order to keep track of all the vehicles connected to the Fog Node a data store method is needed as per functional requirements. The Fog node should have a processing method which decides whether and to which connected clients it should send notifications to if there is an incident detected by one of them. This should be based on location and closeness to the originating warning.

4. Implementation

The following section describes the system implementation, delving into a lower level of detail, using code examples to explore the realisation of the concepts and ideas developed in the previous sections as well as attend to the successes and problems in creating the solution. It will go into detail about how the design of the system has been realised and highlight the important and interesting components and aspects of the system. Ultimately, with similar projects, while time management is of utmost importance, stumbles can be encountered along the development path, so this section will also address such time-keeping problems.

4.1. Changes to Initial Plan

Inevitably, during the course of the project design and development there are aspects that are bound to have changes. In this table, a list of the initial plans from Appendix C and final outcomes has been made so that it is clear what is and what is not being implemented.

Initial Plan	Actual Implementation
The system will be designed to collect or emulate numerous sensor information such as speed, acceleration, proximity, location etc.	The vehicle sensory data is emulated through keyboard presses which are then interpreted into the appropriate vehicle signals. There is no hardware GPS module connected and so location data is emulated internally.
Process this (vehicle) data locally using distributed nodes in an attempt to reduce the amount of information relayed to cloud services and simulate as closely as possible a real CAN bus	The vehicle data mentioned is processed both on the vehicle and on the Fog network front which does in turn fulfill the intent of reducing the information relayed to cloud-based services. The CAN Bus is also emulated as closely as the constraints of the project permit, seen later in the implementation.
Issues around security, such as potential attack patterns relevant to vehicles, privacy in the context of this work.	The design and implementation of the vehicular system has made use of popular security methods which are outlined in this section. However, attack surfaces and privacy concerns have not been explored.

Potential integration with Cardiff University “Formula Student” race car [32] will also be explored.	This has not been made possible because the CAN Bus simulator is not fully ready and code-complete for an integration with the Cardiff University “Formula Student” race car, this will be touched on in the Future Work section.
To achieve the aim of the project, “Raspberry Pi” System on a Chip (SoC) boards will be used in conjunction with various types of sensors, in order to mimic a vehicle’s Electronic Control Units (ECUs).	The project does make use of readily-available “Raspberry Pi” boards, however sensors have not been needed and have been replaced by a keyboard in order to mimic vehicle driver inputs. The only sensor that this project might have benefited from is a GPS location tracker, however such hardware was not accessible and is part of the constraints for the project.
Create a complete simulator of the car network, with one “Raspberry Pi” acting as a Central Vehicle Gateway (SCV) and multiple others (ECUs) relaying to it data collected from sensors.	While this has been accomplished, the project has been limited to the use of two “Raspberry Pis”, one acting as the vehicle SCV and one other relaying the drivetrain, and other vehicle functions data. More details explaining what has been accomplished can be found in the following section.
The SCV uses this (sensor) data and processes it locally, obtaining concise, insightful information about the car’s e.g. speed, location, proximity to other vehicles and more, thus acting as an “Edge” node.	This has largely been accomplished, however the internal processing is limited and tied to only a few vehicle functions, such as speed, location and (after having received it from the Fog node) collision/incident data. This will be further explained in the below sections.
Some of this compiled and filtered (sensor) information is then relayed to the appropriate “Fog” nodes.	This has been accomplished and explained in the below sections.
They (Fog nodes) then send the concise data to cloud services (e.g. improving geo-location data, traffic information, accidents etc.).	This objective has not been met because of the time constraints and the lack of implementation of this will not affect the main goal of the project: to fully understand the implications of Fog and Edge

	computing.
Implement a simulator of a car's CAN bus, OBD (On-board diagnostics) and ECU's	Every component apart from the OBD connector has been implemented and can be found explained below. Vehicles typically use OBD as a diagnostics method and is the only way to access and read real hardware CAN Bus data. This project makes use of a virtual CAN Bus, which I was not aware to exist prior to writing of the Initial Plan. Being virtual, the CAN Bus implemented in this project is readily accessible at any point and is just a matter of connecting to the "Raspberry Pi" that has the CAN Bus Simulator running, thus emulating the purpose of the OBD connector.
Test the Automotive Grade Linux (AGL) distribution.	During initial research into the topic of the project, AGL seemed promising, however limitations have been found and in summary, at its current development state, AGL is limited and specialized in vehicular Heating, ventilation and air conditioning systems (HVAC)[46] and development was just being started for the drivetrain and other, more mission-critical systems. Further details of the choice to not use AGL will be explained below in the Successes and Failures section.
Use Java and any other necessary languages/frameworks to develop a framework for simulating the separate ECUs and SCV gateway. Package it appropriately so it can be extended in the future.	As usual with Initial Plans and proposals, the bar is normally set high and this is one of the examples which could not be executed within the time frame of the project. During development, the components of the system have been created with the intent of flexibility and modularity, however they could not be packaged into a framework. While progress has been made towards extensibility, for example, get/set methods and other supporting functions, they can not be imported into popular

	languages such as Python as a package with the syntax “import <i>packagename</i> ”. This will be touched on in the Future development section.
Supplement the framework with a web-based interface to nicely visualise the processed data from the individual nodes.	The programme has a visualisation method, however it does not make use of a web-based interface, but a much better, vehicular specific approach which is explained in the Instrument Cluster sections.

4.2. CAN Bus Simulator Implementation

This section will describe how the CAN Bus Simulator has been made from a configuration, and implementation point of view as well as any issues that have occurred during development are addressed.

4.2.1. Configuration

In order to use the CAN Bus simulator a few dependencies need to be first installed and configured. The first thing that needs to be addressed is to check whether the Linux operating system kernel supports CAN Bus interface drivers.

To check whether the SocketCan driver is loaded by the operating system:

```
$ modprobe vcan
```

To add the vcan0 virtual interface:

```
$ sudo ip link add dev vcan0 type vcan
$ sudo ip link set up vcan0
```

That is all that’s needed to setup the virtual CAN interface and now it is ready to receive CAN frames.

In order to test that CAN Bus is setup correctly, a set of very useful command line tools defined and created by the Linux kernel developers are ‘can-utils’[27] can be installed. They can be used in the same fashion on real CAN busses as well.

‘candump’ prints all the data received on the interface. In a terminal window run:

```
$ candump vcan0
```

From another window we can send a CAN frame with an identifier of 0x1A (which is 26 in decimal) as well as 8 bytes of arbitrary data:

```
$ cansend vcan0 01a#1122334455667788
```

Which will appear on the first window that’s running ‘candump’:

```
vcan0 01A [8] 11 22 33 44 55 66 77 88
```

After having set up the interface we can install the simulator dependencies.

It is assumed Python3 is installed on the target device. To install the required dependencies run:

```
pip3 install pynput  
Pip3 install python-can
```

‘Python-can’ is a simple python interface for the CAN bus which makes the code cleaner when reading or writing to the bus but does not add any particular benefit to functionality. ‘Pynput’ defines a keyboard listener for key presses and registers them.

4.2.2. Implementation

The CAN bus simulator is written in Python3 and can be split into two main modules: Electronic Control Unit and Server.

Electronic Control Unit

The ECU deals with all the information and signals inside the vehicle, this include vehicle driver inputs as well as the static variables affected by other inputs such as tire pressure, seat belt, oil pressure warnings and others.

The ECU contains a keyboard press listener which deals with interpreting the user keyboard input into the appropriate CAN Bus frames. It makes use of the ‘pynput’ keyboard listener library and overrides its ‘on_press’ and ‘on_release’ functions. During development, the default keyboard listener could not interpret multiple key presses at the same time, mimicking multiple user inputs such as using the turn signal and applying throttle at the same time. A

custom class was built to override the functions and store the unique key presses in a 'set' data type of keys which are then parsed by the keyboard handler functions. The keyboard handler functions detect the pressed keys from the keyset and translate them into CAN bus signals. An example of this would be applying the throttle. When pressing the 'up' arrow key on the keyboard connected to the ECU, a speed increase signal is sent to the CAN Bus. This mimics exactly what happens in a real vehicle [35], where the real car ECU detects that the throttle valve is being opened and uses that signal to increase the fuel rate to the engine. Of course, not having a real engine, the controls for increasing airflow or fuel rate have been omitted, however there is no problem adding these down the development line since the code is built in a modular fashion.

The following series of code snippets represent what was just described but in a programmatic way. Fig. 20 represents the overridden 'MyListener' keyboard class.

```
# Listener Class to override keyboard.Listener
# methods on_press and on_release
# and define a keySet which stores the
# actively pressed keys
class MyListener(keyboard.Listener):
    def __init__(self):
        super(MyListener, self).__init__(self.on_press, self.on_release)
        self.key_pressed = None
        self.key = None
        self.keySet = set()
    def on_press(self, key):
        self.key_pressed = True
        self.key = key
        if key not in self.keySet:
            self.keySet.add(key)
    def on_release(self, key):
        self.key = key
        if key in self.keySet:
            self.keySet.remove(key)
        if not self.keySet:
            self.key_pressed = False

# Initialize keyboard listener
listener = MyListener()
listener.start()
```

Figure 20: CAN Bus: Overridden keyboard input class

Every key is then iterated over to check its type in Fig. 21.

```
if listener.key_pressed == True and started_press == False:
    started_press = True
    # create a copy of the keySet to prevent the set changing size during iteration
    for key in keySet.copy():
        try:
            # check whether an alphanumeric character is pressed
            alphanum_key_pressed_handler(key)
        except AttributeError:
            # exception is thrown if not
            key_pressed_handler(key)
```

Figure 21: CAN Bus: Key press iteration

Depending on the type of key pressed (alphanumeric or special key), different vehicle parameters are changed as seen in Fig. 21 and 22.

```
# Handle special key presses
# Depending on the key pressed,
# set the appropriate parameter value and
# send the correct message to the CAN Bus
def key_pressed_handler(key):
    if key == keyboard.Key.enter:
        # start the car, otherwise stop it
        stop_car() if get_engine() else start_car()
        check_car_status(keySet)
    elif key == keyboard.Key.right:
        # set right turn signal to on
        set_right_sign(1)
        msg_sign_right()
        check_car_status(keySet)
```

Figure 22: CAN Bus: Key pressed handling

In the above example in Fig. 22, on pressing the ‘Enter’ key, the car start and stop is toggled, depending on whether the vehicle engine is on or off. The second condition is met if the right arrow key is pressed, which turns on the right turn signal. All the vehicle signals follow the same pattern: 1. Invoke ‘set’ method for the parameter to change its value. 2. Send a message to the CAN Bus, notifying all parties listening to the bus about the change. A function that writes to the bus can be seen in Fig. 23.

```
# Send park brake message
def msg_park_brake():
    park_brake = get_park_brake()
    msgParkBrake = can.Message(arbitration_id=0x1f7, data=[0, 0, 0, 0, 0, park_brake], extended_id=False)
    try:
        bus.send(msgParkBrake)
        print("Park Brake " + str(park_brake) + " sent on {}".format(bus.channel_info))
    except can.CanError:
        print("Message NOT sent")
```

Figure 23: CAN Bus: Message sending

The function is self-explanatory, it sends the park brake status to the CAN bus. It begins by retrieving the current ‘park brake’ hexadecimal value (for this parameter it is either 0x00 or 0x01) and places it in the ‘data’ field of the CAN Frame (which using python-can is defined as Message). The ‘arbitration_id’ is the arbitration id field in the actual CAN Bus frame which has to be unique for the different parameters we are sending on the bus. The ‘extended_id’ flag specifies the possibility for having a longer identifier, the default value is 11 bits, and the extended is 29 bits if the use case has many different types of messages. The CAN Frame is then sent to the configured CAN Bus virtual interface, which in this case is ‘vcan0’. Technically, it is possible to setup a real, physical CAN bus and send the same data to it. This is why the simulator can be used for rapid prototyping of vehicular services as there is no overhead in testing it on real equipment after passing preliminary, virtual CAN Bus tests.

Server

The Server component's purpose in the Simulator is to read data that is being sent on the CAN bus and relay it to the Instrument Cluster using a Secure Sockets Layer (SSL). To implement it, again, Python3 was used for its popularity and ease of readability. The main functionality of the Server can be simply expressed as forwarding CAN Frames to the Instrument Cluster. It does this by creating a secure sockets endpoint as seen in Fig. 24.

```
# Private and public key files
# for SSL auth
KEYFILE = 'privkey.pem'

CERTFILE = 'pubkey.pem'

# Set the hostname, portname
# and buffer size for the server
ic_host = '0.0.0.0'
ic_port = 8082
ic_buf = 1024

# CAN Bus Simulator server initialisation
ic_address = (ic_host, ic_port)
ic_socket = socket.socket(AF_INET, SOCK_STREAM)
ic_socket.bind(ic_address)
ic_socket.listen(1)
ic_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

# Specify SSL socket
ic_socket_ssl = ssl.wrap_socket(ic_socket, keyfile=KEYFILE,
                                certfile=CERTFILE, server_side=True)
```

Figure 24: CAN Bus: SSL configuration

It sets the hostname port numbers and begins listening on the socket which is then wrapped in SSL by the private key and the certificate. This way, encryption is guaranteed on the way and only parties who have the certificate can take part in communication. This goes in line with the requirements for security. The next important part is the actual forwarding method for the CAN bus packets. The server creates a thread which listens for messages on the CAN bus and immediately after that parses them and sends them as JSON format to the Instrument Cluster. This approach alleviates the Instrument Cluster from having to do all the packet parsing which would slow down displaying times, and have the data nicely served. Python provides easier to use tools for parsing of bytes and hexes compared to C++ used in the Instrument Cluster.

The following snippet of code in Fig. 25 shows how the connection between the trusted party and the Server is established

```
# Start the CAN bus server
# and relay the CAN messages to the instrument cluster
def can_server():
    while True:
        try:
            # accept ssl socket connections
            (can_socket, can_address) = ic_socket_ssl.accept()
            print('Got connection', can_socket, can_address)
            # potentially allow receiving messages from the Instrument Cluster head unit
            #thread_ic_receiver = _thread.start_new_thread(ic_receiver, (can_socket, can_address))

            # create CAN bus listener thread
            thread_can_sender = _thread.start_new_thread(can_sender, (can_socket, bus))
        except socket.error as e:
            print('Error: {0}'.format(e))
```

Figure 25: CAN Bus: Server threading

The following snippet in Fig. 26 shows how messages are received, parsed and sent to the Instrument cluster.

```
# Relay CAN Bus messages to the Instrument Cluster
# through an SSL socket
def can_sender(can_socket, bus):
    while True:
        message = bus.recv()
        arr = []
        for byte in message.data:
            arr.append(str(int(byte)))
        can_socket.send(json.dumps({"id": message.arbitration_id, "data": arr}).encode('utf-8'))
```

Figure 26: CAN Bus: Relaying message

4.3. Instrument Cluster Implementation

The purpose of the Instrument Cluster is to establish communication with the CAN Bus Simulator and the Fog Network and visualise the data it receives. The following sections explain how the Instrument Cluster was configured and implemented.

4.3.1. Configuration

Qt5 and QML were used in the creation of the Instrument cluster. The program was written using the Qt5, QtCreator Studio IDE which makes prototyping, development and testing easier as it provides the tools needed to debug Qt applications. The configuration steps will not go into detail as to how to install Qt5 and QtCreator Studio, as the install is platform dependent and documentation can be found on the Qt website [43]. The following steps will go over how to run the Instrument Cluster project.

After having installed Qt5 and QtCreator, the project can be imported into the studio by selecting *File->Open File* or project and selecting the '*qtcluster-base.pro*' file from the

system file explorer. This will import all the project directories into the IDE, ready to be compiled or ran.

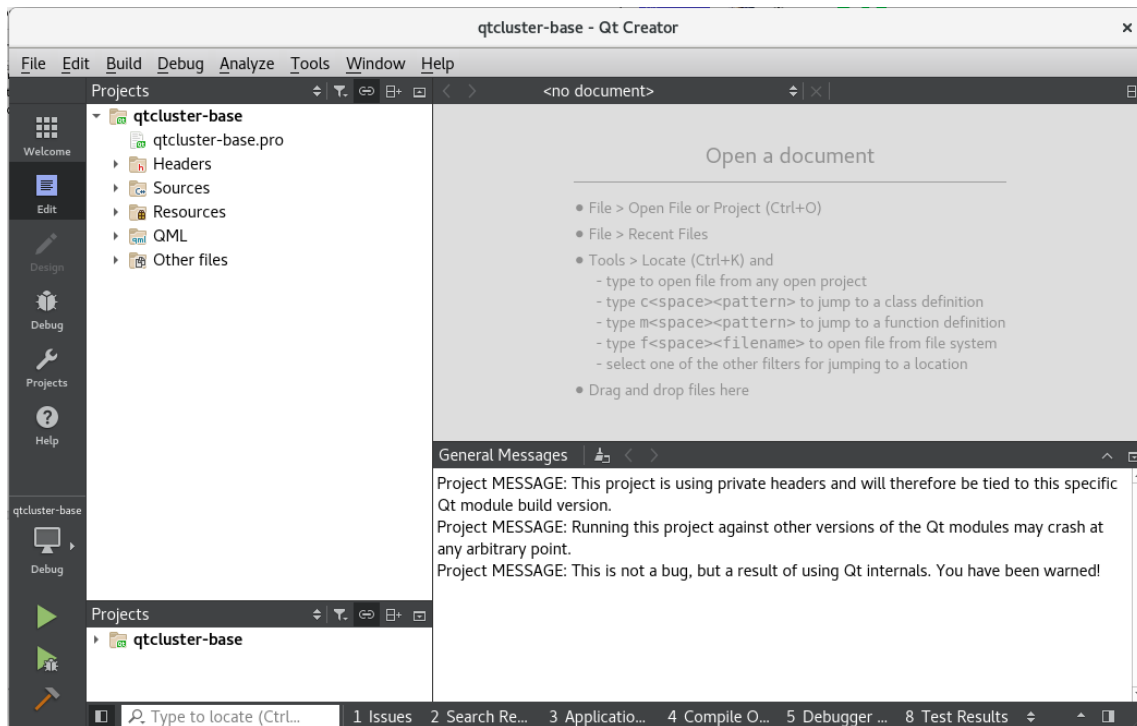


Figure 27: Qt Creator IDE

By pressing *Ctrl+B* or selecting *Build->Build Project* we can compile the project for our platform. After that is done, by pressing *Ctrl+R* or *Build->Run*, the Instrument Cluster display window should open and it should be automatically connected to the CAN Bus Simulator and Fog Node (if they are running as well).

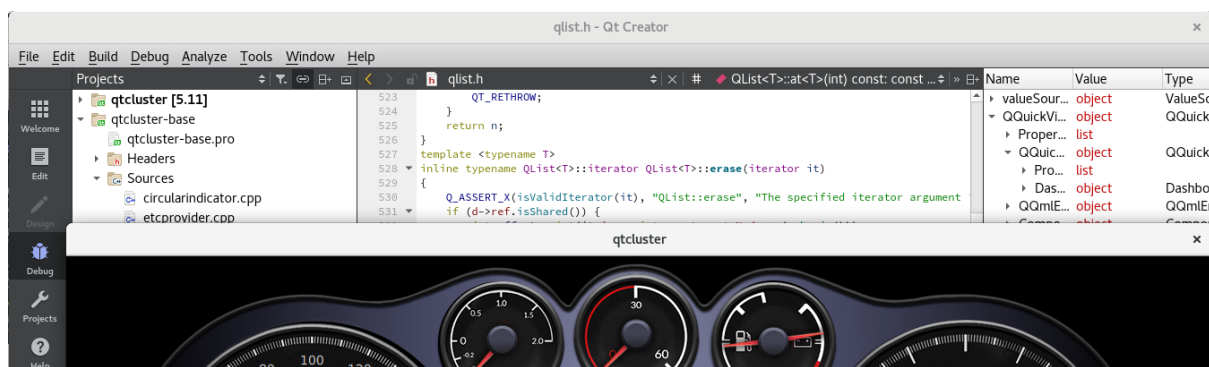


Figure 28: Instrument Cluster running in Qt Creator IDE

4.3.2. Implementation

As stated in the background section Qt5 and QML were identified as best suited for the Implementation of the Instrument Cluster. The Qt company is currently focused at automotive manufacturers like Mercedes-Benz, Tesla and Rimac to enable better vehicle interfaces[36]. This project, having the same target environment, fits perfectly with Qt and their open source automotive tools. The design for the Instrument Cluster happens to be open-sourced and used as a demonstration for the Qt and QML languages potential.

However while the demonstration might be pretty, it uses hard-coded values. For speed rpm and is in no way dynamic or can simulate the connected vehicle. Allowing for modifications and public use, the Qt automotive cluster demo is open-source[37]. By making use of a static design and skeleton for the instrument cluster and implementing the features stated in the design section, the requirements for the visualisation of the vehicle and Fog nodes information were achieved.

The Instrument Cluster functionality implementation is divided into two main sections: CAN Bus Listener and Fog Node Listener.

CAN Bus Listener

The CAN Bus Listener is a client that connects to the CAN Bus Server described in the previous section and accepts incoming CAN Bus frames. The business logic can be seen in the following snippet in Fig. 29:

```
106 // Can bus socket
107 // Check if certificate exists
108 QLatin1String rootCApath = QLatin1String("pubkey.pem");
109 QFileInfo check_file(rootCApath);
110
111 // Load certificate
112 QList<QSslCertificate> cert = QSslCertificate::fromPath(rootCApath, QSsl::Pem, QRegExp::Wildcard);
113
114 // Create an self-signed certificate error
115 // and add it to the SSL ignore list
116 // For a production system this won't be an issue
117 QSslError error(QSslError::SelfSignedCertificate, cert.at(0));
118 QList<QSslError> expectedSslErrors;
119 expectedSslErrors.append(error);
120 m_webSocket.ignoreSslErrors();
121
122 // Attach QSslSocket::encrypted signal
123 // to the onSslConnected method
124 connect(&m_webSocket, &QSslSocket::encrypted, this, &QtIVICClusterData::onSslConnected);
125
126 // Do the same for when there is an error
127 // An call the onSslError function
128 typedef void (QSslSocket:: *sslErrorsSignal)(const QList<QSslError> &);
129 connect(&m_webSocket, static_cast<sslErrorsSignal>(&QSslSocket::sslErrors),
130         this, &QtIVICClusterData::onSslErrors);
131
132 // Connect to the CAN Bus with encryption
133 m_webSocket.connectToHostEncrypted("169.254.22.105", 8082);
```

Figure 29: Instrument Cluster: listener for CAN Bus and Fog Node connections

In short, in the file `'qtiviclusterdata.cpp'`, the certificate created by the CAN Bus is Loaded and using SSL sockets, an encrypted connection is established with the CAN Bus. The `connect` statement on line 124 connects the `QSslSocket::encrypted` signal[38] with the `onSslConnected` method. In this case `m_webSocket` is the object which emits this *encrypted* signal and on line 133, the connection is begun, if it is successful, the `onSslConnected` method is called.

The `onSslConnected` function simply attaches the `readyRead` [39] device signal to the `checkMessages` function which then checks the payload of the CAN Bus data transmitted on the web socket as seen in Fig. 30.

```

141 void QtIVIClusterData::onSslConnected()
142 {
143     qDebug() << "socket connected";
144     connect(&m_webSocket, &QSslSocket::readyRead,
145            this, &QtIVIClusterData::checkMessages);
146 }

```

Figure 30: Instrument Cluster: On connection with CAN Bus

In the `checkMessages` function the JSON object payload transmitted by the CAN Bus server is deserialized and the ID is appropriately checked against the arbitration IDs defined earlier in the CAN Bus converted to decimal, e.g. speed is id #580, gear is #501 etc. as seen in Fig. 31.

```

198 void QtIVIClusterData::checkMessages()
199 {
200     QString message = m_webSocket.readLine();
201
202     QJsonDocument jsonResp = QJsonDocument::fromJson(message.toUtf8());
203     QJsonObject jsonObj = jsonObj.object();
204
205     int arbitration_id = jsonObj.value("id").toInt();
206     QJsonArray payload = jsonObj.value("data").toArray();
207
208     bool ok;
209     if (arbitration_id == 580) {
210         //set speed
211         int speed = payload.at(3).toString().toUInt(&ok, 10);
212         onVehicleSpeedChanged(speed);
213         int nRpm = speed * 150;
214         onRpmChanged(nRpm);
215     } else if (arbitration_id == 501) {
216         int gear = payload.at(3).toString().toUInt(&ok, 10);
217         onGearChanged(gear);

```

Figure 31: Instrument Cluster: Checking message types

The payload data is then converted and sent to the appropriate functions which then visualises it in the instrument cluster appropriately.

Fog Node Listener

The Fog node listener works in the same fashion as the CAN Bus listener with the only difference that the connection is not encrypted. The reason for this is that the target devices which will facilitate a network connection between each other will be in the real life scenario, cell towers and cellular-technology enabled units using well-established encryption methods such as 3DES and others to encrypt data in transit, thus this is not the main focus of the implementation.

```
185 void QtIVIClusterData::checkCollision()
186 {
187     QString message = m_tcpSocket.readLine();
188
189     QJsonDocument jsonResp = QJsonDocument::fromJson(message.toUtf8());
190     QJsonObject jsonObject = jsonResp.object();
191
192     if(jsonObject.value("collision").toString() != ""){
193         onCollisionChanged(true);
194     }
195 }
```

Figure 32: Instrument Cluster: Collision data checking

The *checkCollision* function as seen in Fig. 32, checks whether the Fog node sends collision data, and if so, displays the information on the Instrument Cluster. The Fog Node Listener also periodically sends location data to the Fog node so that it can keep track of the vehicle's position.

4.4. Fog Network Implementation

The Fog network consists of Fog nodes which have the same functions, to receive location and vehicle status information from vehicles connected to them. While Fog nodes should typically be represented by server-grade hardware and cell towers, for the purpose of this prototype, a laptop has been used to allow for similar features such as a semi-powerful processor and wireless interfaces to which other nodes and vehicles can connect.

4.4.1. Configuration

The Fog Node does not have many specific configuration requirements apart from having Python3 installed on the system. In order to allow other devices to connect to the system (in this case a laptop), a shared wireless connection is required. Depending on the platform, this can be done in different ways, but in most Linux systems with a Graphical interface it is a matter of going into 'Network Settings' and turning on WiFi hotspot sharing.

4.4.2. Implementation

The Fog Node represents a simple server that accepts vehicle connections and their JSON payloads which contain location and vehicle status information. In a similar fashion to the CAN Bus Server it creates a server listener. The only difference is that it starts a thread to handle the client connection while it is alive and allows for keeping track and addressing multiple connections asynchronously. The current configuration of the Server accepts location, speed and collision JSON data and saves the parameters in a dictionary assigned per the client's address. This can be seen in Fig. 33.

```
# Begin listening for connections
while True:
    try:
        print("Server is listening for connections\n")
        clientsocket, clientaddr = serversocket.accept()
        clients[clientsocket] = {"location": "", "speed": ""}
        # Create thread when there is a new connection
        thread = _thread.start_new_thread(handler, (clientsocket, clientaddr))
    except KeyboardInterrupt:
        print("Closing server socket...")
        serversocket.close()
```

Figure 33: Fog Node: Listening for vehicle connections

In Fig. 33 '*clients*' is a defined dictionary and the location and speed are assigned as empty when initialising the connection and starting the thread. The next time the vehicle (in this case the Instrument Cluster) sends location and speed data, the parameters will be saved towards this dictionary together with the rest of the vehicles connected to the Fog Node.

```
# Handler thread which checks the payload
# of every received request from a client and
# assigns their parameters such as location and speed
def handler(clientsocket, clientaddr):
    print("Accepted connection from: ", clientaddr)
    while True:
        bin_data = clientsocket.recv(buf)
        data_utf8 = bin_data.decode("utf8").rstrip()
        if "disconnect" in data_utf8 or "" == data_utf8:
            print("Disconnect: " + clientaddr)
            break

        data = json.loads(data_utf8)

        # assign location and speed to the clients
        if "location" in data:
            if clientsocket in clients:
                clients[clientsocket]['location'] = data['location']
                clientsocket.send(json.dumps({"location": data['location']}).encode("utf8"))
```

Figure 34: Instrument Cluster: Handling connections and their data

The '*handler*' function in Fig. 34 checks the data received by the connected client, deserializes it and checks for the data types existent and assigns them appropriately. For the collision data type, however, the process is slightly different as seen in Fig. 35.

```

# Checks if there is collision info in the payload
if "collision" in data:
    if clientsocket in clients:
        clients[clientsocket]['speed'] = data['speed']
        clients[clientsocket]['location'] = data['collision']

    clientsocket.send(json.dumps({"collision": data['collision']}).encode("utf8"))
    latitude, longitude = clients[clientsocket]['location'].split(',')

# Alert all clients in the vicinity about the collision
for client in clients:
    client_lat, client_lon = clients[client]['location'].split(',')
    if client_lat == latitude and client_lon == longitude:
        client.send(json.dumps({"collision": data['collision']}).encode("utf8"))

```

Figure 35: Fog Node: Collision Checking and Notification sending

When a collision is detected, the program iterates over the list of known clients connected and alerts each and everyone, whose position is within 200m of the collision in order to avoid further incidents i.e. is the same within 3 decimal points, the default accepted latitude and longitude format that the clients send uses 4 digits after decimal point which is accurate within 3.6 feet or 1.1 meters [40]. Please consult with Appendix B to see what the approximate accuracy is for latitude and longitude decimal notation. When a client disconnects from the Fog node, the programme removes its entry from the dictionary to prevent iterating over it.

In order to test the vehicle incident use case, a dummy, connected vehicle script was created to simulate a vehicle sending a collision warning to the system. While this could be implemented on another CAN Bus Simulator and Instrument Cluster with the respective Raspberry Pi boards connected to the Fog node, this is unnecessary code duplication and use of physical devices. The Python3 script simply sends its vehicle coordinates and after a couple of seconds, sends a collision warning to the system as seen in Fig. 36.

```

# Send a location and speed message
# to the Fog Node which will be registered
location = {'location': '43.2355,45.2245', 'speed': '75'}
clientsocket.send(json.dumps(location).encode("utf8"))
print("REPLY: " + clientsocket.recv(buf).decode("utf8").rstrip())
time.sleep(2);

# Send a collision notification at
# the same location so vehicles
# in the vicinity are alerted
collision = {'collision': '43.2355,45.2245', 'speed': '75'}
clientsocket.send(json.dumps(collision).encode("utf8"))
print("REPLY: " + clientsocket.recv(buf).decode("utf8").rstrip())

```

Figure 36: Vehicle Collision notification script

This script will be used later in the Evaluation section to test the main use case of the system.

4.5. Overall System

In order for the use case to be executed, the separate components need to be put together and connect with each other to transmit the necessary vehicle/fog information.



Figure 37: Hardware Fog Node and Connected vehicle Simulator representation

The above image in Fig. 37 represents the final implementation of the system and its components. The laptop on the right of the image is running the vehicle collision simulator script and the Fog node, to which the red-coloured Raspberry Pi is wirelessly connected which represents the Instrument cluster. A visual representation of the Instrument cluster can be seen on the monitor to the left, also connected to the same Raspberry Pi, visualising the Fog Node and CAN Bus inputs. The right-most Raspberry Pi represents the CAN Bus Simulator and is connected to the Instrument Cluster via Ethernet. This is due to the fact that a physical CAN Bus could not be created or sourced and so the alternative is to use Ethernet. Connected to the headless CAN Bus Simulator is a keyboard which represents the vehicle driver inputs e.g. throttle, turn signals etc. When an action is initiated through the keyboard, the CAN Bus recognises the action and sends a signal to the Instrument Cluster which then visualises it on the Monitor. Sending a Collision signal from the laptop results in the Instrument Cluster Raspberry Pi to display the warning on the Monitor.

4.6. Successes and Failures

Overall, looking at the system as a whole, the major aspects have been delivered, of course not to the standards of a production-ready application, but as it stands, it can accommodate expansions and patches for the missing pieces in the future. What is surprising is that not many of the initial requirements have changed and really, only minor deviations have been made for the sake of creating a better implementation.

However, the whole process was not as simple as it may seem. The time spent for initial research into which technologies were actually used within cars of today and tomorrow was disproportionate to the rest of the project because of the sheer mass of proprietary technologies manufacturers use and do not disclose. Above all else was the complexity factor, trying to mimic something which is implemented in electronics hardware has a steep research and implementation curve.

Having managed to make a list of possible technologies it was really trial and error, with many iterations on the different components in the system, especially for the Instrument Cluster. Initially using a graphical C++ library, then turning to PyGame and then to Qt (being mostly C++), took a substantial amount of time to implement. As a matter of fact it took a whole week to only setup the Raspberry Pi to compile the Qt application and getting the demonstration up and running. The platform was not officially supported by Qt which led to the difficulties in setting the environment up for building the software. The Qt and QML languages were not entirely new to myself, however it did take time to get back into grips with programming it. Thankfully the documentation provided by the Qt Company was very useful and up-to-date.

Another hurdle that was met was the vast amount of ideas and additions that came up during the project development. These ideas will be covered in the Future developments section, however I personally found them to be distracting and deviating from the originally set out path, which in the long run consumes time available for the project.

As with most projects, initial ambitions are high, which is especially true in this case, however I believe that through careful planning and lots of research it is possible to at least come close to achieving your goals, which I believe I have managed. A further reflection on the project and the learning benefits it has contributed to, will be covered in the next sections.

5. Evaluation and Results

Having completed the system implementation, the next phase of the project is to check to what extent we have achieved the goals of the project. This section will discuss the overall outcome by carrying out tests and examining how well the aims have been realised as well as the project learnings and achievements.

5.1. Testing

During development of the system, many tests were performed to check whether every component fulfills the requirements. Continuous debugging and testing at each stage of the project was needed to ensure the original objectives are met . The below table gives details into the final test cases that were created to check whether the final implementation is in line with the design specifications and requirements. The overall outcomes of the test cases show whether the system implementation was a success. The test cases have been chosen to reflect and demonstrate the main functionalities of the system, assuming it is configured correctly. The below performed use case tests make sure the system works in the scope of the intended functionalities and does not go in-depth into the more rigorous test approaches which will be justified afterwards.

Test Cases

Description	Pre-Conditions	Expected Outcomes	Outcomes
1. Pressing the UP arrow key on the CAN Simulator.	N/A	The instrument Cluster displays an increase in speed.	The speedometer needle moves up on the Instrument Cluster. Success
2. Releasing UP arrow key on the CAN Simulator.	UP arrow key was pressed.	The Instrument Cluster displays the speed lowering.	The speedometer needle moves down on the Instrument Cluster. Success
3. Pressing the Enter Key on the CAN Simulator.	N/A	Parameters Battery level, Fuel Level, Temperature level, RPM should increase	The Instrument Cluster Battery level, Fuel Level, Temperature level,

		on the instrument cluster. To appropriate amounts effectively starting the car	RPM parameters increase from 0 to a preset level effectively starting the car. Success
4. Pressing the Enter Key on the CAN Simulator again	Enter Key was pressed once before.	Parameters Battery level, Fuel Level, Temperature level, RPM, Speed, Gear should go down to 0 effectively stopping the car.	The Battery level, Fuel Level, Temperature level, RPM, Speed, Gear parameters on the dashboard go down to 0 effectively stopping the car Success
5. Pressing the Right/Left arrow Key on the CAN Simulator	N/A	Turn signals on the simulator should start blinking.	The Turn signals light up. Success
6. Releasing the Right/Left Arrow Key on the CAN Simulator	The Right/Left arrow keys were pressed before.	Turn signals should continue blinking.	The Turn signals continue blinking. Success
7. Pressing the 0,1,2,3,4,5,9 keys on the CAN Simulator	N/A	The selected gear would change appropriately. 0 is Neutral and 9 is Reverse gears.	The selected gear changes appropriately. Success
8. Pressing the 'p' button on the CAN Simulator	N/A	The icon on the instrument cluster should light up and show that the parking brake is engaged	The parking brake icon lights up on the Instrument cluster dashboard. Success
9. Pressing the 'p' button again on the CAN Simulator	The button was pressed once before	The instrument cluster parking brake icon should turn off.	The icon turns off on the instrument cluster dashboard. Success
10. Pressing the 'l'	N/A	The 'lights' icon on	The 'lights' icon on

button on the CAN Simulator		the instrument cluster should light up to indicate the car lights are on.	the Instrument cluster dashboard lights up. Success
11. Pressing the 'l' button on the CAN Simulator again	The button was pressed once before	The 'lights' icon on the instrument cluster should turn off indicate the car lights are on.	The 'lights' icon on the Instrument cluster dashboard turns off. Success
12. Pressing the 's' button on the CAN Bus Simulator	N/A	The seat belt icon would turn on the instrument cluster.	The seat belt icon lights up on the instrument cluster. Success
13. Pressing the 's' button on the CAN Bus Simulator again	The button was pressed once before	The seat belt icon would turn off the instrument cluster.	The seat belt icon turns off on the instrument cluster. Success
14. Pressing the 'f' button on the CAN Bus Simulator	N/A	The fuel level should begin to increase in 1% increments on the Instrument Cluster	The fuel level increases on the dashboard every time the button is pushed. Success
15. Pressing the 't' button on the CAN Bus Simulator	N/A	The temperature level should begin to increase in 1% increments on the Instrument Cluster	The temperature level increases on the dashboard when the button is pressed. Success
16. Pressing the 'b' button on the CAN Bus Simulator	N/A	The battery level should begin to increase in 1% increments on the Instrument Cluster	The battery level increases on the dashboard when the button is pressed. Success
17. Pressing the left 'shift' button on the CAN Bus	N/A	A door icon on the instrument cluster should indicate the doors are locked	A door icon was not implemented in the Instrument cluster. Failure.

18. Pressing the right 'shift' button on the CAN Bus	N/A	A door icon on the instrument cluster should indicate the doors are unlocked	A door icon was not implemented in the Instrument cluster. Failure.
19. Start the vehicle collision simulator	The Fog node and CAN Bus Simulators must be active.	The vehicle instrument cluster shows a collision notification.	A collision notification is shown on the instrument cluster display. Success
20. Disconnect the Instrument cluster from the Fog node connection	The Instrument Cluster was connected to the Fog node.	The Fog node should continue accepting connections.	The Fog node is still active and can continue operation. Success.
21. Disconnecting the CAN Bus from the Instrument Cluster	The CAN Bus was connected to the Instrument Cluster.	The Instrument cluster should continue running.	The Instrument cluster continues operation. Success.
22. Disconnecting the Instrument Cluster from the CAN Bus.	The Instrument Cluster was connected to the CAN Bus.	The CAN Bus should continue accepting inputs.	The CAN Bus is still active and continues operation. Success.
23. Reconnecting the Instrument Cluster to the CAN Bus.	The Instrument cluster was disconnected to the CAN Bus.	The Instrument cluster should begin to receive CAN Bus inputs.	The Instrument cluster receives CAN Bus frames after reconnecting and displays them. Success.

To gain further confidence in the completeness and robustness of the system, many different classifications of tests can be executed which would be more rigorous, such as unit testing, integration testing, component interface testing, acceptance testing, performance testing, security testing etc. and would definitely be advantageous in identifying the weak points in the system.

However, this being a project aimed at being flexible and extensible, allowing for other developers to pick it up and further extend it i.e. it will continually be in development, the above mentioned tests would be more suitable for a production-ready environment, which this project does not have as a short-term aim. On top of this, such rigorous testing usually

takes up half of the available development time, which is not a possibility for a feature-driven project like this one. The further testing requirements have been considered for the Future Development section of the project, and will definitely be necessary if the project takes a turn for a production-ready environment such as automotive manufacturing companies.

5.2. Progress against Objectives

In order to critically evaluate the results of the system and the overall progress we have to look into what was created and how it meets the expectations of the main project objectives.

Objective 1: Create a CAN Bus Simulator prototype that mimics as closely as possible a connected car's internal network

While a complete production-ready system has not been achieved, the intended implementation of the prototype CAN Bus Simulator has been fulfilled and all of the 'must have' requirements met. This can be reaffirmed from the above use case tests. The prototype CAN Bus simulator provides, to a great extent, a look into the internal workings of a vehicle, how parameters are being processed and exchanged between the different components. It can definitely, with more time and effort be extended into a fully-featured production-ready application which will be touched on in the Future Development section. In comparison to the initial plan, what has been implemented, only has small details omitted or changed which are not crucial or that completely change the functionality. Examples such as the lack of OBD connector, for diagnostics, limitation to only two ECU's and not using the AGL distribution as outlined in the Changes to Initial Plan section are minor requirements that do not deviate from the initial aims, but only complement them. As it stands at the moment it provides a useful basis for other developers to simulate a multitude of different parameters, the ability to add more functionality and features and flexibility and extensibility that would cater any developments they might want to create. Overall, for the available time frame of the project, the simulator has the complete feature set and is tested to work as intended in the use case scenario.

Objective 2: Visualise the simulated connected car's functionality using an automotive Instrument Cluster

With one of the major aims of the project being, visualising the internal car networks' parameters, the Instrument Cluster was not easy to conceptualise and implement. Having said in the Initial Plan that there needs to be a form of a visualisation method, noting a web-based interface, during the course of the project development this became not viable due to the technical requirements for low-latency in the displaying of the inputs. This objective became a major hurdle when researching and testing the different methods it could be implemented with as stated in the Implementation section. Because of the necessity and diligence in

completing this requirement it took a substantial amount of the project development time to configure and implement the Instrument Cluster. The end result however was worth that persistency. Showing from the use case testing above is that the requirements are all met for the purpose of the Instrument Cluster and it displays both CAN Bus and Fog data without issues. In fact, owing to the open-source Qt instrument cluster design and demonstration, I believe that this component of the system is more feature-complete than the rest and does not have as much room for improvement as the rest of the components. Despite this, ideas and features have been listed for the Instrument Cluster in the Future Developments section. At its current state, the visualisation component provides a useful method for seeing the real advantages of the use case that is being simulated: collision avoidance. It is definitely more ‘production-ready’ than the rest of the system components, however it still needs further rigorous component interface testing and many other testing approaches to be applied before naming it that.

Objective 3: Create a Fog Network in supporting vehicular services

The Fog and Edge computing theory has not yet been explored and is just beginning to show its full potential. This project merely scratches the surface of the possibilities that this technology may bring to the world of communications and computing. The attempt here was to create and simulate a Fog Network in the field of vehicular services. This proved to be a challenge from the start as it would be for most people that explore a new, unknown to them before notions and technologies. What was beneficial to the project and the implementation was the availability of very up-to-date research articles done by professors, knowledge groups and scientists alike, that really gave this project a good direction into this novel approach. Having done the research, implementing the Fog network was still not an easy task to achieve as there are no technology standards that have been selected as of yet. Being a new technology, it is bound to be changing requirements in the future, as the public sees what works and what does not. And this possible future change has also been taken into account in the implementation, making it flexible, modular and extensible. Reflecting on the current state of the Fog network component, there is much room to improve, especially in establishing connections (at the moment it is a fixed port and IP address), as well as processing more than the specified initial vehicle parameters. The possibilities of extending the Fog network further have been mentioned in the Future Development section. In retrospect I believe that the Fog network nodes support the main aim and use case of the project well, showing the advantages of using this new and enabling technology. In my opinion it would take time before the technology matures and this project could be a setting stone in other Fog and Edge-driven vehicular services that could be realised in the coming years.

Overall Project Progress

Having used the Feature Driven Development methodology proved beneficial in that it allowed to clearly define and develop the components in a step-by-step manner. Being loosely coupled, the system made it easier to see and improve parts which did not support the overall goal of the system. The choice to use popular programming languages such as Python make the system more accessible for future expansion by other developers who are unfamiliar with vehicular systems. Qt is not the most known of programming frameworks, however it has incredible potential in the automotive industry, proven by the wide adoption by industry leaders such as Tesla and Mercedes-Benz. Using the well-established network ‘sockets’ communication link, contributes to the already good accessibility of the project.

Apart from the above mentioned facts, there are strengths and weaknesses to using Fog & Edge computing for this project. The immediate strong point is that, the project is part of a growing research library of the implications of the technology. By doing more exploration into the capabilities we can find new or existing use cases to implement this approach to. Another strong point is the actual technology, by having vehicles and participants one network hop away from the service, latency and throughput limitations are basically eradicated.

In today’s world when computing power is becoming ever so cheaper, accessible and more compact, the financial implications of the technology for deployment in enterprise environments are not that huge. However the limitations come in with the socio-economic adoption and installation of such computing Fog nodes on cell towers around the public roads. The project has not explored the possible connection between the cell tower interface and the actual Fog node link that is based at the cell tower. This would require more research into how cellular networks work and would go too far off the topic of the project. This is something that the companies which support existing cellular infrastructure need to address and there probably will be many limitations until a possible solution is found.

Another downside to the project is that there is the possibility of vehicular Ethernet, to phase out CAN Busses in cars. Ethernet technology is becoming cheaper and easier to implement and allows gigabit speeds of throughput and with autonomous vehicle sensors such as cameras and radar constantly sending data, the CAN bus will probably be used only for the drivetrain of the vehicle. However there are limitations to every technology and vehicle Ethernet is no exception as manufacturers are currently working on setting specific vehicular protocols which would make the technology very proprietary and inaccessible, as can be seen with Tesla[47].

6. Future Development

The following sections will look into any ideas that have arisen during project development and could not be explored in the given time frame as well as longer-term plans for the future of the project.

Initially having a broad scope, the project could not deliver a production-ready application, however a prototype implementation on which future developments can be made has been firmly established.

Packaging the CAN Bus simulator

While progress has been made towards extensibility of the CAN Bus Simulator, for example get/set methods and other supporting functions, they should be able to be imported into popular languages such as Python as a package with the syntax ‘import *packagename*’ so that individual functions and classes can be called. This will accelerate development and improve developer adoption of the project. The long-term goal is to have this simulator as a useful tool with which different internal vehicle functions can be prototyped, simulated and used in various connected vehicle or transportation projects alike. By making the simulator this flexible, it might not become automotive specific but could also be used in the prototyping and simulation of projects in different fields such as food processing, nuclear physics, spacecrafts, oil platforms, wind energy farms and in general, sensor-enabled environments.

Simulating more vehicular features and AGL integration

It is obvious that this prototype has not provided a simulation of all of the vehicular features and it would be beneficial if more are added, to make a more detailed picture and simulation of a vehicle’s internal network. The potential is there, to simulate more CAN Buses and link them together, systems such as the Heating, ventilation and air conditioning (HVAC) system can be operated in unicycle. Automotive Grade Linux has already made a good simulation of the HVAC systems[4] so a potential collaboration with AGL is possible, to bridge the two projects together and make a uniform system. AGL foundation is currently experimenting with providing support for the vehicle drivetrain, however their expertise is in complete production-ready implementation, not simulation.

Fog node reconnection

When a vehicle becomes out of range from the Fog node it needs to reconnect to the next, closer node to it. This requires the vehicle to always be scanning for the Fog nodes and to connect to the one which has the best signal. This could be done by using the same SSID but

this could not be implemented and tested in the available time. This addition would really bring the project closer to a production system.

Communication between Fog nodes

Another aspect that needs to be worked on in the future to improve the system, is establishing communication between Fog nodes, so they can relay service or any type of information. This requires further research into how the cellular network works, down to the cable, base-stations and general protocols. Distances, network topologies and security have to be taken into account while devising the communication methods. It could be a possibility to route data through the cloud servers where they have reach over the whole network but this might bring latency to the system

Use real cell tower technology to simulate the V2I connection

Another possible future work that could be carried out is simulating cell tower technology in a closed environment. However, this would require purchasing specialised equipment, receivers and transceivers which have to be in line with existing standards. True simulation of the environment and conditions in which connected vehicles operate can then be achieved. Research into the protocols and technology for cellular communication from end-to-end will be necessary and using that knowledge to replicate a use case on a smaller scale. By doing this, performance, throughput and scalability can be tested.

Containerisation of the Fog service

A potential feature and benefit when creating a Fog service if it were containerised. At the moment, the fog service emulated is simply a script running on a given machine. In the future it will be advantageous if the Fog service becomes more fully-featured, with dedicated storage methods, communication protocols and has its own operating environment to become independent from the platform it is being installed on. This is the notion of containerisation and technologies such as Docker [41]. This way, the service can be propagated through the Fog network and installed on only the necessary Fog nodes and can be brought down or put back up, depending on the needs of the system.

7. Conclusions

Fog and Edge Computing is a novel approach at solving the future high-bandwidth, low-latency needs of connected devices, Internet of Things, automation and vehicles alike. The implications of the technology as conceptualised, prototyped and tested in this project are substantial. Connected vehicles in the near future will most likely be the first ones to experience the advantages of this technology. With the rise of collision prediction and avoidance technologies, the use case shown using the connected car simulator and Fog network here, could be a part of the whole industry movement toward “Vision Zero”[6] accident roads. Due to these implications, the importance factor of the project is high.

Despite only being a partial implementation of a full, production-ready system, the project testifies the advantages of creating a vehicle simulator and using it in a Fog environment. Although each vehicle component could not be simulated, the majority of vehicular, mission-critical systems have been shown to function as intended, with the only limitations to expanding the list even further, being time and resources. What is important, the developed system is extensible, can be built on and improved in the future to support different use cases to the initial scope of this project.

The technologies used for the completion of the implementation have been specifically chosen to allow for the best compatibility with what is existing in the internal vehicle systems. This is especially true for the Instrument Cluster display which makes use of the industry-leading framework ‘Qt’, which allows for deployment of the component on any embedded systems and is crucial for the potential adoption of the project with the target audience. The hardware used to create and test the system with is also readily available, making the project even more accessible for potential future development.

In retrospect, the project has achieved what it set out to do, and while it has obvious limitations, it has produced a system which has no analogue in the current computing space and I personally believe that these initial steps taken can feasibly become a mature and valuable tool for the rapid prototyping and development of edge-driven real-time vehicular services. Every component developed with the intention of being stand-alone allows the even further expansion of the separate components into separate products for different purposes in fields other than automotive, which is especially true for the CAN Bus Simulator.

Only time will tell what the future may bring and which technologies will prevail in the connected world of tomorrow.

8. Reflection on Learning

The purpose of this section is to critically evaluate my personal performance in tackling the project, to show which skills and abilities have been developed as well as the learning areas which would need focusing on in the future. Apart from the regular learning of programming languages, I will look into the improvement of personal and professional skills. I will be comparing my progress against the Skills Framework for the Information Age (SFIA) [42] in order to examine the employability and professional skills developed.

8.1. Personal Skills

Time Management

Having finalised the project, I can really appreciate how important time-management is in the development of systems of this scale. Even the careful planning was insufficient from the initial report, as it was expected that some parts of the system would take a disproportionate amount of time to complete. Any disruptions in the learning flow were not taken into account or any unpredictable design changes.

After having done the research and creating the initial system design, planning the workload of the project, proved to be a challenge and the best way I found I could keep track of my time was to segment the separate functionalities of the project to be designed and developed into time-frames, working incrementally to get a better view of the scale of the project. I then assigned each functionality an estimated completion time in order to get a perspective of what could be included and what had to be omitted or put aside until the major components were complete.

Adhering to the work plan I set out to complete, issues into developing some of the components started to arise. The Instrument Cluster took a substantial amount of time to develop and the rest of the functionalities had to be pushed back because of it. Reflecting on this experience, I believe it is a better way to plan the project with time to spare. For example, if initially assigning a week to a given component (which might actually be the real estimate time to develop), an extra quarter or half of that time should be assigned as room for problems and their resolution. By doing this to every requirement, the system is bound to be completed without time management issues.

A further method that would have been useful is carrying out a mid-term project review. By doing this it would have been possible to identify what I have been trailing behind on. I was so involved in solving the development of the Instrument Cluster that a review of the progress for the system as a whole would have given me a better idea that overtime work

would be needed if the rest of the system had to be completed within the overall time-frame. Other components would subsequently lag behind in their completeness, by ‘eating into’ their time on the Instrument Cluster implementation.

This learning experience was valuable to me as it showed me the pitfalls of time management and gave me a better understanding of how to cope with unexpected project time irregularities.

Project Planning and Organisation

Having regular meetings with the project supervisor meant I could discuss any details of the project I was uncertain of, such as the choice of technologies, approach and general guidance for the project. From those very useful conversations I could adjust and plan my project accordingly and much of the time-scheduling from the previous section contributed to the overall satisfactory organisation of the project.

Even though initial project planning was in place, having a coursework for another module mid-term affected the scheduling and required careful reorganisation so as to not overlap the two projects. This was also true during the Easter break, when for unpredicted reasons crucial time had to be taken away from the already tight schedule. Missing small details like these builds up and necessitates working on the project overtime to avoid the possibility of not meeting the requirements. This experience has taught me a valuable lesson to “expect the unexpected” and give the project some head-room.

8.2. Progress against SFIA

During my Placement year in industry, I became familiar with the SFIA network for evaluating personal and professional development against a set of criteria defined by the framework for different skills specific to IT people. I have chosen a few particular skills that I think I have improved in over the course of the project. I will be using version 6 of the SFIA framework to make a self-reflection and analysis of my progression of the most notable skills from the development of this project.

Solution Architecture (ARCH)

The SFIA Solution architecture skill defines that the person can deal with the overall design of a high level architecture that would give the development stages of the project further guidance. From the descriptions of the SFIA Level mappings I believe that through the work on this project I have achieved a Level 5 on the mapping. This is because I believe I cover most of the requirements to possess this level of achievement. Using appropriate tools, logical models, components and interfaces in order to contribute to the development of systems architectures in functional areas description of the mapping fits the work done on this project in Sections 3 and 4. Producing detailed component specification and detailed designs

for implementation using selected products is also covered in this project in section 2 and 3. Since this project is not used in a business, the business change and assurance requirements are not relevant. Providing advice on technical aspects of system development and integration such as future changes, deviations from specifications is also something covered in the Future Development sections. I can not consider myself Level 6 on the mapping because it defines the need to be in a professional environment to get feedback from project stakeholders in order to ensure the design covers the key points of the solution.

Systems Design (DESN)

During the course of the project I believe I have reached a Level 3 in SFIA Systems Design which means that I am able to specify user/system interfaces, interpret logical designs into physical ones, taking into account the target environments, any performance and security requirements and existing systems (Sections 2 and 3). As well as this, I have to be able to produce detailed designs and documents that work using required standards (sections 3 and 4), methods and tools as well as prototyping tools where necessary (section 4). I believe that I satisfy these requirements for Level 3 in systems design as they are mostly covered to the desired level throughout the report.

Network Design (NTDS)

Through the research and development that went into creating the Connected Vehicle Simulator, I have gained valuable insight and knowledge as to how existing vehicle, roadside and in general, network infrastructures operate and have conceptualised the network for the system. This learning experience goes in line with Level 5 of the SFIA Network Design professional IT skill which mentions that producing system designs and specification and overall architectures and design of networks and networking technology. The skill description also mentions specification of user/system interface including validation, access, security, risks and translates logical designs into physical designs which is the case with the work done in this project. I believe that this is a very useful skill in my intentions to have to deal with networks in the professional future environment in a company.

Programming/Software Development (PROG)

Of course, having dealt with a software development project of this scale, improvements in Programming/Software Development skills as defined by SFIA are bound to happen. From my year in industry, having been a systems administrator for a company, I did not have much of an opportunity to improve my Software Development skills. I am happy to say that with the development of this project I believe I have reached Level 3 in this professional skill. This means that I have achieved abilities such as designing, coding, testing, correcting and documenting moderately complex software programs and scripts from agreed specification and subsequent iterations. I believe that this is a valuable skill to improve because of the potential future work in the industry implications.

Table of Abbreviations

Abbreviation	Meaning
SLA	Service level agreements
IoT	Internet of Things
OS	Operating System
ITS	Intelligent Transportation systems
Fog	Stratum of small servers or miniaturized data centers that communicate with end devices and cloud services
Edge	Stratum of end-devices which consume given content or services
ECU	Electronic Control Units
Raspberry Pi	Low power, low cost, system on a chip boards used for development projects
AGL	Automotive Grade Linux
FDD	Feature Driven Development software methodology
3G,4G,5G	3rd, 4th, 5th generation mobile communications technologies
V2V	Vehicle-to-Vehicle
V2I	Vehicle-to-Infrastructure
V2X	Vehicle-to-Everything
VCS	Vehicular Communications Systems
DSRC	Dedicated Short Range Communications
OSI	Open Systems Interconnect model
MAC	Media access control address
ARP	Address resolution protocol

CAN	Controller area network
DLC	Data Length Code
WAVE	Wireless Access in Vehicular Environment
ETSI	European Telecommunications Standards Institute
Kb,Mb,Gb,Tb	kilo, mega, giga, tera bits
KB,MB,GB,TB	kilo, mega, giga, tera bytes
Kbps,Mbps,Gbps,Tbps	kilo, mega, giga, tera bits per second
MIMO	Massive Multiple-input Multiple-output
SSID	Service Set Identifier
ABS	Anti-lock braking system
RPM	Revolutions per minute

Appendices

Appendix A

List of CAN Messages and their functions for the CAN Bus Simulator

CAN ID (HEX)	CAN ID (DEC)	Data in Byte (From 0 to 5)	Vehicle Function
0x244	580	3	Speed value (num)
0x1F5	501	3	Gear value (num)
0xF7	503	5	Park brake (on/off)
0x1F8	504	5	Lights value (on/off)
188	392	5	Turn Signals (0-off, 1-left, 2-right)
0x1FB	507	5	Seat belt (on/off)
0x1FC	508	5	Fuel level (num 0-100)
0x1FD	509	5	Engine Temperature level (num 0-100)
0x1FE	510	5	Battery Level (num 0-100)
0x1FF	511	5	Oil Level (num 0-100)
0x200	512	4 and 5	RPM (0 - 65 535)
0x201	513	5	ABS warning (on/off)
0x202	514	5	ABS warning (on/off)
0x203	515	5	Engine (on/off)

Appendix B:

GPS Location decimal points and their distance accuracy.

Decimal	Accuracy
Six .000001	4 inches
Five .00001	3.6 feet
Four .0001	36 feet
Three .001	360 feet
Two .01	3600 feet 0.7 miles
One .1	36,000 feet 6.9 miles

Appendix C - Edge-driven Real-time Vehicular Services - Initial Report Submission

References

1. Amazon Web Services, Inc.. 2018. *Global Cloud Infrastructure | Regions & Availability Zones | AWS*. [ONLINE] Available at: <https://aws.amazon.com/about-aws/global-infrastructure/>. [Accessed 10 Jan 2018].
2. Patrick Nelson. 2018. *One autonomous car will use 4,000 GB of data per day | Network World*. [ONLINE] Available at: <https://www.networkworld.com/article/3147892/internet/one-autonomous-car-will-use-4000-gb-of-dataday.html>. [Accessed 13 Jan 2018].
3. Joann Muller. 2018. *10 Obstacles For Connected Cars - pg.1*. [ONLINE] Available at: <https://www.forbes.com/pictures/mkk45ihlk/10-obstacles-for-connected-cars/#61ad1aa012b8>. [Accessed 13 Jan 2018].
4. Automotive Grade Linux. 2018. *Home - Automotive Grade Linux*. [ONLINE] Available at: <https://www.automotivelinux.org/>. [Accessed 10 Jan 2018].
5. Automotive Grade Linux. 2018. *Automotive Grade Linux Platform Debuts on the 2018 Toyota Camry - Automotive Grade Linux*. [ONLINE] Available at: <https://www.automotivelinux.org/announcements/2017/05/30/automotive-grade-linux-platform-debuts-on-the-2018-toyota-camry>. [Accessed 20 Feb 2018].
6. Vision Zero - Wikipedia. 2018. *Vision Zero - Wikipedia*. [ONLINE] Available at: https://en.wikipedia.org/wiki/Vision_Zero. [Accessed 21 Feb 2018].
7. Feature Driven Development (FDD) and Agile Modeling. 2018. *Feature Driven Development (FDD) and Agile Modeling*. [ONLINE] Available at: <http://agilemodeling.com/essays/fdd.htm>. [Accessed 21 Feb 2018].
8. CNBC. 2018. *Mobile data usage set to explode nearly six-fold by 2020: Report*. [ONLINE] Available at: <https://www.cnbc.com/2015/06/03/mobile-data-usage-set-to-explode-nearly-six-fold-by-2020-report.html>. [Accessed 10 May 2018].
9. Statista. 2018. *Connected Cars - Statistics & Facts | Statista*. [ONLINE] Available at: <https://www.statista.com/topics/1918/connected-cars/>. [Accessed 10 Feb 2018].
10. CISCO. 2015. *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are | Cisco*. [ONLINE] Available at

https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf [Accessed 10 Jan 2018].

11. Department of Transport. 2018. *Reported road casualties in Great Britain: quarterly provisional estimates year ending September 2017* | Department of Transport. [ONLINE] Available at: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/681593/quarterly-estimates-july-to-september-2017.pdf [Accessed 14 Apr 2018].
12. Edgeline Converged Edge Systems for Secure Edge Computing | HPE United Kingdom. 2018. *Edgeline Converged Edge Systems for Secure Edge Computing* | HPE United Kingdom. [ONLINE] Available at: <https://www.hpe.com/uk/en/servers/edgeline-iot-systems.html>. [Accessed 12 Jan 2018].
13. TechCrunch. 2018. *More cars than phones were connected to cell service in Q1* – TechCrunch. [ONLINE] Available at: <https://techcrunch.com/2016/06/20/more-cars-than-phones-were-connected-to-cell-service-in-q1/>. [Accessed 25 Feb 2018].
14. Fog computing - Wikipedia. 2018. *Fog computing - Wikipedia*. [ONLINE] Available at: https://en.wikipedia.org/wiki/Fog_computing. [Accessed 26 Feb 2018].
15. TerminalWorks Team. 2018. *TerminalWorks Blog | Fog Computing*. [ONLINE] Available at: <https://www.terminalworks.com/blog/post/2017/05/13/fog-computing>. [Accessed 28 Feb 2018].
16. E. Siegel, D. Erb, S. Sarma. 2017. *A Survey of the Connected Vehicle Landscape—Architectures, Enabling Technologies, Applications, and Development Areas* | IEEE [ONLINE] Available at: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8058008>. [Accessed 3 Mar 2018].
17. Dedicated short-range communications - Wikipedia. 2018. *Dedicated short-range communications - Wikipedia*. [ONLINE] Available at: https://en.wikipedia.org/wiki/Dedicated_short-range_communications. [Accessed 5 Mar 2018].
18. eeNews Automotive. 2018. *Why 802.11p beats LTE and 5G for V2x* | eeNews Automotive. [ONLINE] Available at: <http://www.eenewsautomotive.com/design-center/why-80211p-beats-lte-and-5g-v2x/page/0/6>. [Accessed 5 Mar 2018].

19. B. E. Bilgin, V. C. Gungor. 2013. *Performance Comparison of IEEE 802.11p and IEEE 802.11b for Vehicle-to-Vehicle Communications in Highway, Rural, and Urban Areas* | International Journal of Vehicular Technology Volume 2013. [ONLINE] Available at: <https://www.hindawi.com/journals/ijvt/2013/971684/> [Accessed 6 Mar 2018].
20. Steven Winkelman. 2018. *5G is coming — here's what to expect, and when to expect it on your carrier* | *www.DigitalTrends.com* [ONLINE] Available at: <https://www.digitaltrends.com/mobile/what-is-5g/>. [Accessed 10 Mar 2018].
21. Alphr. 2018. *5G: What is 5G and when is it coming to the UK?*. [ONLINE] Available at: <http://www.alphr.com/technology/1005678/what-is-5g-and-when-is-it-coming-to-the-uk>. [Accessed 11 Mar 2018].
22. CAN bus - Wikipedia. 2018. *CAN bus - Wikipedia*. [ONLINE] Available at: https://en.wikipedia.org/wiki/CAN_bus. [Accessed 15 Feb 2018].
23. OSI model - Wikipedia. 2018. *OSI model - Wikipedia*. [ONLINE] Available at: https://en.wikipedia.org/wiki/OSI_model. [Accessed 27 Apr 2018].
24. Oliver Hartkopp. 2017. *The CAN Subsystem of the Linux Kernel | Automotive Grade Linux* [ONLINE] Available at: https://wiki.automotivelinux.org/_media/agl-distro/agl2017-socketcan-print.pdf [Accessed 9 Apr 2018].
25. SocketCAN - Wikipedia. 2018. *SocketCAN - Wikipedia*. [ONLINE] Available at: <https://en.wikipedia.org/wiki/SocketCAN>. [Accessed 18 Feb 2018].
26. The Linux documentation Project. 2018. *Chapter 10 Networks* [ONLINE] Available at: <https://www.tldp.org/LDP/tlk/net/net.html>. [Accessed 27 Apr 2018].
27. GitHub. 2018. *GitHub - linux-can/can-utils: Linux-CAN / SocketCAN user space applications*. [ONLINE] Available at: <https://github.com/linux-can/can-utils>. [Accessed 12 Feb 2018].
28. SocketCAN (python) — python-can 2.1.0 documentation. 2018. *SocketCAN (python) — python-can 2.1.0 documentation*. [ONLINE] Available at: http://python-can.readthedocs.io/en/stable/interfaces/socketcan_native.html. [Accessed 13 Feb 2018].
29. Qt (software) - Wikipedia. 2018. *Qt (software) - Wikipedia*. [ONLINE] Available at: [https://en.wikipedia.org/wiki/Qt_\(software\)](https://en.wikipedia.org/wiki/Qt_(software)). [Accessed 27 Apr 2018].

30. Adafruit Industries. 2018. *Adafruit Industries, Unique & fun DIY electronics and kits*. [ONLINE] Available at: <https://www.adafruit.com/>. [Accessed 27 Apr 2018].
31. Setup - Raspberry Pi Documentation. 2018. *Setup - Raspberry Pi Documentation*. [ONLINE] Available at: <https://www.raspberrypi.org/documentation/setup/>. [Accessed 13 Feb 2018].
32. MoSCoW method - Wikipedia. 2018. *MoSCoW method - Wikipedia*. [ONLINE] Available at: https://en.wikipedia.org/wiki/MoSCoW_method. [Accessed 23 Feb 2018].
33. Wi-Fi - Wikipedia. 2018. *Wi-Fi - Wikipedia*. [ONLINE] Available at: <https://en.wikipedia.org/wiki/Wi-Fi#Range>. [Accessed 10 Feb 2018].
34. Cardiff Racing - CR13. 2018. *Cardiff Racing - CR13*. [ONLINE] Available at: <http://www.cardiffracing.co.uk/CR13.html>. [Accessed 12 Jan 2018].
35. HowStuffWorks. 2018. *When You Step on the Gas - How Fuel Injection Systems Work | HowStuffWorks*. [ONLINE] Available at: <https://auto.howstuffworks.com/fuel-injection2.htm>. [Accessed 26 Apr 2018].
36. The Qt Company. 2018. *Driving the future with Qt | Qt*. [ONLINE] Available at: <https://www.qt.io/qt-in-automotive/>. [Accessed 26 Apr 2018].
37. qtcluster\snippets\src\doc - qt/qtdoc.git - Qt Documentation. 2018. *qtcluster\snippets\src\doc - qt/qtdoc.git - Qt Documentation*. [ONLINE] Available at: <http://code.qt.io/cgit/qt/qtdoc.git/tree/doc/src/snippets/qtcluster?h=dev>. [Accessed 29 Feb 2018].
38. QSslSocket Class | Qt Network 5.10. 2018. *QSslSocket Class | Qt Network 5.10*. [ONLINE] Available at: <http://doc.qt.io/qt-5/qsslsocket.html#encrypted>. [Accessed 24 Mar 2018].
39. QIODevice Class | Qt Core 5.10. 2018. *QIODevice Class | Qt Core 5.10*. [ONLINE] Available at: <http://doc.qt.io/qt-5/qiodevice.html#readyRead>. [Accessed 24 Mar 2018].
40. Decimal Latitude Longitude Accuracy – LightManufacturing Systems. 2018. *Decimal Latitude Longitude Accuracy – LightManufacturing Systems*. [ONLINE] Available at: <https://lm.solar/heliostats/support/decimal-latitude-longitude-accuracy/>. [Accessed 25 Mar 2018].
41. Docker. 2018. *Docker - Build, Ship, and Run Any App, Anywhere*. [ONLINE] Available at: <https://www.docker.com/>. [Accessed 3 Mar 2018].

42. SFIA framework — SFIA. 2018. *SFIA framework — SFIA*. [ONLINE] Available at: <https://www.sfia-online.org/en/sfia-6>. [Accessed 28 Apr 2018].
43. The Qt Company. 2018. *Download Qt: Choose commercial or open source*. [ONLINE] Available at: <https://www.qt.io/download>. [Accessed 27 Apr 2018].
44. Carla Mouradian. 2018. *A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges* | IEEE [ONLINE] Available at: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8100873> [Accessed 14 Mar 2018].
45. Electronic instrument cluster - Wikipedia. 2018. *Electronic instrument cluster - Wikipedia*. [ONLINE] Available at: https://en.wikipedia.org/wiki/Electronic_instrument_cluster. [Accessed 24 Apr 2018].
46. HVAC - Wikipedia. 2018. *HVAC - Wikipedia*. [ONLINE] Available at: <https://en.wikipedia.org/wiki/HVAC>. [Accessed 29 Apr 2018].
47. Jason Torchinsky. 2018. *The Tesla Model S Is Basically A Good Looking IT Department On Wheels*. [ONLINE] Available at: <https://jalopnik.com/the-tesla-model-s-is-basically-a-good-looking-it-depart-1558372928>. [Accessed 2 May 2018].