NTNU
Norwegian University of
Science and Technology

Development of a Virtual Traffic Light Testbed

Anders Brastad

Master of Science in Communication Technology
Submission date: February 2017
Supervisor: Wantanee Viriyasitavat, IIK

Norwegian University of Science and Technology
Department of Information Security and Communication

Title: Development of a Virtual Traffic Light Testbed

Student: Anders Brastad

Problem description:

Virtual Traffic Light (VTL) is a self-organising traffic control concept proposed to manage traffic at intersections for use in Intelligent Transport Systems (ITS). This VTL concept is enabled by the design of local rules which allow vehicles approaching an intersection to resolve the ensuing conflict in a seamless and self-organising manner. Through the use of vehicle-to-vehicle (V2V) communication, the VTL protocol can dynamically optimise traffic flow at intersections without the need for any roadside infrastructure.

While the VTL has been extensively studied (in terms of technical and business perspectives), the real testbeds for VTL has not been implemented. This project thus aims to fill this gap by using miniature robot cars to implement the VTL algorithm and demonstrate the feasibility and effectiveness of the VTL concept in a test environment.

Responsible professor: Wantanee Viriyasitavat, ITEM

Abstract

Today we are facing many challenges within our systems of transport and traffic. The world of transportation is huge, and the problems are often difficult to resolve. Intelligent transport systems (ITS) and services are the future solutions for today's transportation problems. For the years to come the evolution of ITS will become more important than ever before. Virtual Traffic Light (VTL) is a self-organising traffic control concept proposed to manage traffic at intersections for use in ITS. This VTL concept is enabled by the design of local rules which allow vehicles approaching an intersection to resolve the ensuing conflict in a seamless and self-organising manner. Through the use of vehicle-to-vehicle communication, the VTL protocol can dynamically optimise traffic flow at intersections without the need for any roadside infrastructure.

While the VTL has been extensively studied (in terms of technical and business perspectives), the real testbeds for VTL has, to the author's knowledge not been adequately explored. This project thus aims to fill this gap by using Diddyborg robot cars powered by Raspberry Pi 3 computers. We present a design and implementation or a functional VTL testbed which is used to determine the effectiveness and feasibility of the VTL concept. Simulation done with the VTL testbed show promising results.

Sammendrag

I dag står vi overfor mange utfordringer innen transport- og trafikksystemer. Transportverden er stor, og problemene er vanskelige å angripe. Intelligente transportsystemer (ITS) er fremtidens løsninger på dagens transportproblemer. I årene som kommer vil utviklingen av ITS bli viktigere enn noen gang før. Virtuelle trafikklys (VTL) er et selvorganiserende trafikkonsept presentert for å administrere trafikken i veikryss for bruk innen ITS. VTL-konseptet er realisert gjennom lokale regler som tillater biler som nærmer seg et veikryss å løse den påfølgende konflikten på en sømløs og selvorganiserende måte. Gjennom bruk av kjøretøy-til-kjøretøy kommunikasjon, kan VTL-protokollen dynamisk optimere trafikkflyten i veikryss uten behov for noen slags form for infrastruktur.

Mens VTL-konseptet har blitt grundig studert (i form av tekniske og forretningsmessige perspektiver), har fysiske prototyper av konseptet, så langt forfatteren vet, ikke blitt tilstrekkelig utforsket tidligere. Dette prosjektet tar derfor sikte på å fylle dette tomrommet ved hjelp Diddy-Borg robot-biler drevet av Raspberry Pi 3 datamaskiner. Vi presenterer et design og en implementasjon av en funksjonell VTL prototype som brukes for å se på effektiviteten og gjennomførbarheten av VTL-konseptet. Prototypen er testet gjennom simulering med lovende resultater.

Preface

This thesis is the final work of my master's degree at the Department of Telematics (ITEM) at the Norwegian University of Science and Technology (NTNU).

I would like to thank my Professor and supervisor Wantanee Viriyasitavat for providing support and guidance throughout the work of this thesis. A special thanks to Studentersamfundet i Trondhjem for making my years in Trondheim the best years of my life.

Trondheim, 1st of February 2017

Anders Brastad

Contents

List of Figures

List of Tables

List of Acronyms

AU Application Unit.

DSRC Dedicated short range communication.

GPS Global Positioning System.

ITS Intelligent transport systems.

LT Location Table.

NAF Norwegian Automobile Federation.

NOK Norwegian kroner.

NTNU Norwegian University of Science and Technology.

OS operating system.

UDP User Diagram Protocol.

V2V vehicle-to-vehicle.

VANET Vehicular ad-hoc network.

VTL Virtual Traffic light.

WLAN Wireless Local Area Network.

Chapter 1
Introduction

Today we are facing many challenges within transport and traffic systems. The world of transportation is large, and the problems are difficult to grasp.

Intelligent transport systems (ITS) are the future solutions to today's transportation problems. Systems and services that will contribute to a more efficient and safe traffic flow. For the years to come the evolution of ITS is more important than ever before. Streamlining goods and passenger transport while developing a transportation system with a broader focus on security and environment, may save the society from large expenses.

1.1 Motivation

The traditional answer to a problem within the transport sector, whether it's a road that has capacity problems or a road section prone to accidents, has been to build new and better road systems [13]. Through ITS we will be able to exploit new technologies in communication between vehicles and infrastructure to solve these problems in new ways.

ITS is a field of study with rapid development and can act as a supplement, and alternative to traditional infrastructure projects. ITS may contribute to a reduction in the number of fatalities and serious injuries, a sharp reduction in travel times and a significant reduction in environmental impact [13].

Problems with traffic congestion prove to be increasing. Figures from the Norwegian Automobile Federation (NAF) shows that the current situation is untenable. With traffic congestion in the major Norwegian cities costing as much as 2.5 million Norwegian kroner (NOK) per minute. In Oslo, this problem costs up to 1.2 million NOK per minute [2].

1.2 Scope and Objectives

The *Self-Organized Traffic Control* [5] paper is working towards mitigation of traffic congestion by increasing traffic flow at road intersections. Intersections is a crucial part of the traffic system and have a lot of room for improvement when it comes to traffic congestion. In [5] we are presented a VTL concept which uses vehicle-to-vehicle (V2V) communication to establish in-vehicle traffic lights without the need for any road infrastructure. The goal of this study is to design and develop a functional testbed setup of the VTL concept using several miniature robot cars. We use this testbed to demonstrate the feasibility and effectiveness of the VTL protocol. We will take a closer look on the VTL concept in Chapter 3.

This study has several objectives:

- Select suitable robots for the testbed setup.
- Configure the robot cars as network nodes, making them able to send and receive data packets across an ad hoc network.
- Implement a VTL testbed from scratch using miniature robot cars.
- Determine the effectiveness and feasibility of the VTL concept through simulation.

1.3 Methodology

During the work of this thesis, we have been through different phases. The first stage consisted of a literature study of the the VTL concept. This step was done to get an in-depth understanding of how the VTL protocol works. Secondly, we performed a technology study. The goal of this phase was to search for hardware and robots eligible for use in our project. Thirdly we designed and determined the system architecture of the following implementation phase. The next phase went by to implement the proposed system architecture. In the last phase of this project, we experimented with the system implementation, running a simulation to see how the system works in practice.

1.4 Outline

Chapter 2 presents an overview of the technology used in the study. Here we include hardware, communication technologies and software tools.

Chapter 3 gives an introduction to the VTL protocol. This is includes concepts and rules which is essential to understand the following chapters.

Chapter 4 introduce the overall system architecture, the network topology and how the devices are set up and configured.

Chapter 5 presents a simulation scenario, how we conducted the simulation and the results of the simulation.

Chapter 6 discusses the results from Chapter 5 comparing them with each other. We discuss the effectiveness and feasibility the VTL protocol and comments the implementation process. It also concludes this study with a section of further work.

1.5 Related Work

There has been conducted an extensive amount of articles related to the VTL protocol. *Self-Organized Traffic Control* [5] by Ferreira, Michel, et al. and *VANET-Enabled In-Vehicle Traffic Signs* [3] by Fernandes, Ricardo Jorge both presents the VTL concept of VTL and conducts a data simulation of the protocol in a Manhattan-like scenario using traffic simulators. [5] states that the effectiveness of traffic flow is increased by over 60% using the VTL protocol in urban areas.

Feasibility of virtual traffic lights in non-line-of-sight environments [12] by Neudecker, Till, et al. and *On the impact of virtual traffic lights on carbon emissions mitigation* [4] by Ferreira, Michel, and Pedro M. d'Orey. both present a feasibility study of the VTL protocol, though in two different manners. [12] takes a look at the feasibility of VTL in areas where buildings and other structure may interfere with the V2V communication. The article concludes that non-line-of-sight environments have an impact on the delay communication, but not significantly and a VTL seem to be feasible under such challenging conditions. [4] looks at the environmental aspect of a VTL implementation. It evaluates the impact in terms of Carbon Dioxide (CO2) emissions of VTLs [4]. Compared with an approximation of the physical traffic light system they present results that show a significant reduction in CO2 emissions when using VTLs, reaching nearly 20% under high-density traffic [4]

A prototype of Virtual Traffic Lights on Android-based smartphones [11] by Nakamurakare, Manuel, Wantanee Viriyasitavat, and Ozan K. Tonguz present a prototype design on VTLs using Android-based smartphones. They conclude that the Android-based VTL implementation clearly shows feasibility using hardware available in the current smartphones.

Chapter 2
Background

This chapter presents background information about the technology used in the development phase of this study. This includes communication technologies, hardware and software tools.

2.1 DiddyBorg

The robot cars used in this project is the DiddyBorg robot [15]. This is a battery powered miniature robot car and use six 6V DC gear motors [1] to control its six wheels, three on each side. It has a Raspberry Pi [7] as its central computer which makes is a powerful and highly suitable robot for our project. Figure 2.1 and 2.5 both show one of our assembled DiddyBorg robots seen from different angles.

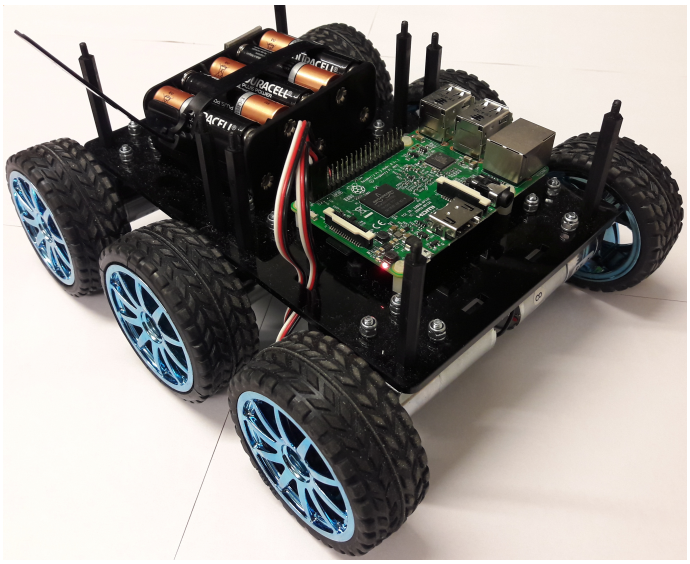


Figure 2.1: The DiddyBorg robot

Several other robot cars would be suited for this project. But when selecting the right robot, the criteria and assumptions of the VTL protocol defined in Chapter 3 had to be fulfilled. The Diddyborg is an excellent choice because the robot is relatively easy to assemble and runs on batteries. Its support for Raspberry Pi is what makes it such a good option. The Raspberry Pi provides support for the wireless technology we need. Also, it has a framework allowing easy control of its motors.

The DiddyBorg robot consist of several parts which we are going to get into in the following subsections:

2.1.1 Raspberry Pi 3

The Raspberry Pi 3 is a credit-card sized computer provided at a low cost [8]. The Raspberry Pi 3 is the third generation Raspberry Pi released by the Raspberry Pi Foundation. It replaced the Raspberry Pi 2 Model B in February 2016. It has a 1.2GHz 64-bit quad-core ARMv8 CPU, 1GB RAM, full HDMI port, 4 USB ports, 40 GPIO pins and ethernet port. Different from the previous model it also has 802.11n Wireless Local Area Network (WLAN) and support for Bluetooth 4.1 and Bluetooth low energy [7]. Figure 2.2 shows a picture of a Raspberry Pi 3.

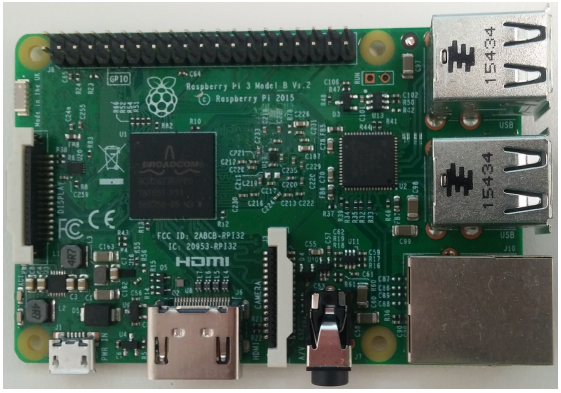
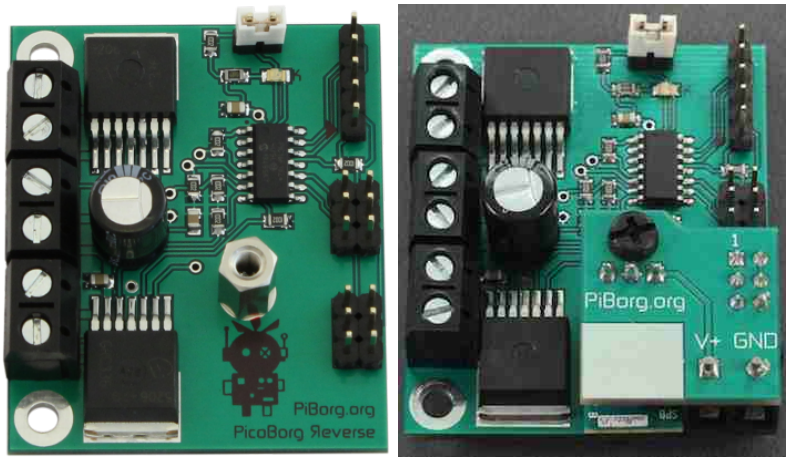


Figure 2.2: The Raspberry Pi 3

2.1.2 PicoBorg Reverse

PicoBorg Reverse is a powerful dual motor control board for Raspberry Pi. Connected to a Raspberry Pi it can to control the motors on the DiddyBorg, also doing speed control both forward and backwards [17]. The PicoBorg Reverse is shown in Figure 2.3a

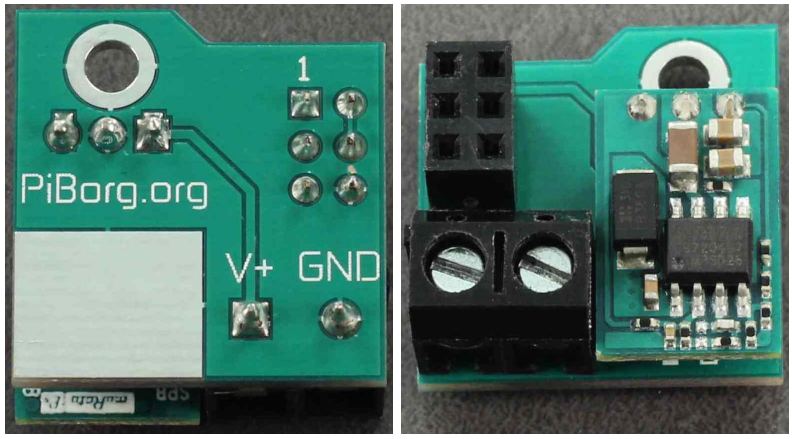


(a) **(b)**

Figure 2.3: The PicoBorg Reverse [17]

2.1.3 BattBorg

The BattBorg is a power converter which allows us to power the DiddyBorg motors and the Raspberry Pi off batteries without needing a USB supply [14]. The BattBorg is shown in Figure 2.4 and is mounted on top of the PicoBorg Reverse as shown in Figure 2.3b.



(a) Battborg top **(b)** Battborg bottom

Figure 2.4: The Battborg [14]

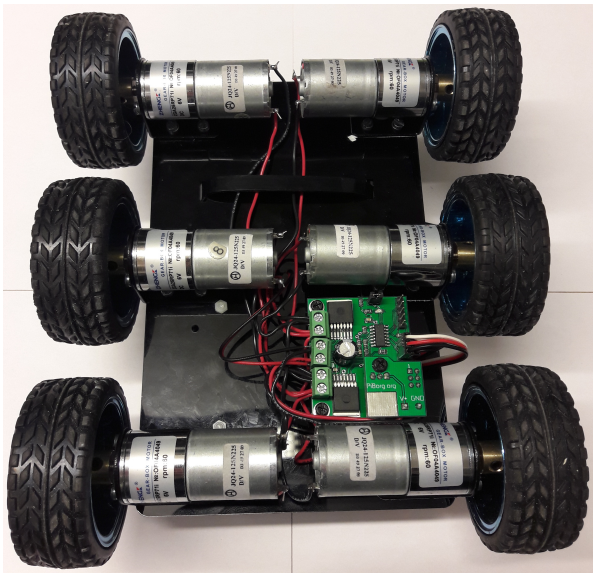


Figure 2.5: The DiddyBorg robot seen from underneath

2.2 Communication

2.2.1 Dedicated Short Range Communication

Dedicated short range communication (DSRC) is a two-way short-to-medium-range wireless communications capability that permits very high data transmission critical in communications-based applications. It is allocated 75 MHz of spectrum in the 5.9 GHz frequency band for use by ITS vehicle safety and mobility applications [19]. DSRC was developed with a primary goal of enabling technologies that support safety applications and communication between vehicle-based devices and infrastructure to reduce collisions [19].

2.2.2 Vehical Ad-hoc Network

Vehicular ad-hoc network (VANET) is a decentralised wireless network meant for automobiles and V2V communication. An ad-hoc network is not in need of any infrastructure such as centralised routers and access points to function. The vehicles act as network nodes in a self-organising dynamic manner. VANET is an important application in the development of ITS enabling vehicles to communicate directly with each other.

2.3 Software tools

2.3.1 Raspbian Jessie

We are using Raspbian Jessie as operating system (OS) for the Raspberry Pi computers in this study. The main reason we use this OS is because installation guide of DiddyBorg [16] recommends it. Raspbian Jessie is a standard operating system provided by the Raspberry Pi Foundation found on their website [18]. Raspbian Jessie also has a graphic user interface which makes it easy to use along with the command line tool.

2.3.2 Wireshark

Wireshark is a network protocol analyser which lets you analyse network information down to a microscopic level [9]. Wireshark supports several communication protocols and able to provide live capture on multiple interfaces. During a capture session from a network interface Wireshark gives you data packet's time of arrival, origin, destination, payload among other useful information. Figure 2.6 shows a screenshot of the Wireshark graphic user interface during a capture. Wireshark lets you highlight, select and extract the information you need. This tool was very helpful during the implementation phase of this study.

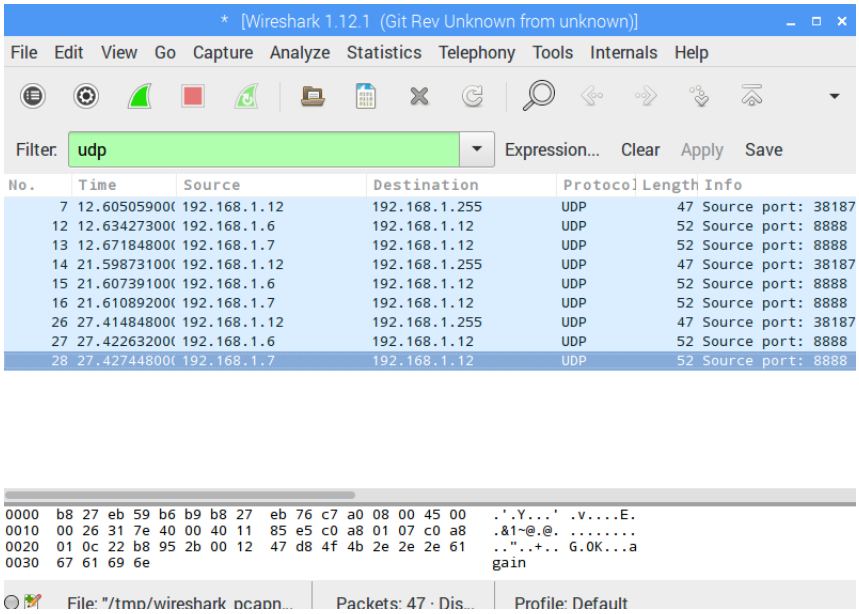


Figure 2.6: Wireshark graphic user interface during a capture

Chapter 3
Virtual Traffic Light

VTL is a self-organising traffic control concept presented in [5, 3]. The authors propose a migration from roadside-based traffic lights to in-vehicle signs supported by V2V communication through DSRC. Elected vehicles act as temporary road junction infrastructures and broadcast traffic light messages that are presented to the drivers through in-vehicle displays [5].

3.1 Assumptions

The implementation of the in-vehicle VTL system is based on the following assumptions [5]:

- All vehicles are equipped with DSRC devices.
- All vehicles share the same digital road map.
- All vehicles have a global positioning system Global Positioning System (GPS) device that guarantees global time and position synchronisation with lane-level accuracy.
- The security, reliability, and latency of the wireless communication protocol are assumed to be adequate for the requirements of the VTL protocol.

3.2 Principle of Operation

The VTL system relies on having information on every vehicle in its vicinity. All vehicles maintain a Location Table (LT) storing this information, constantly updating the LT upon receiving messages from neighbouring vehicles. The location data is distributed through VANET, periodically beaconing, letting every vehicle to broadcast its position, speed and heading to all surrounding vehicles.

All vehicles have an Application Unit (AU) installed. The AU is responsible for maintaining an internal database with information about where VTLs can be created. When approaching an intersection, the AU should check if there is a VTL running, or if there is a need for creating one. Using the LT the AU discovers if there are ensuing crossing conflicts between approaching vehicles.

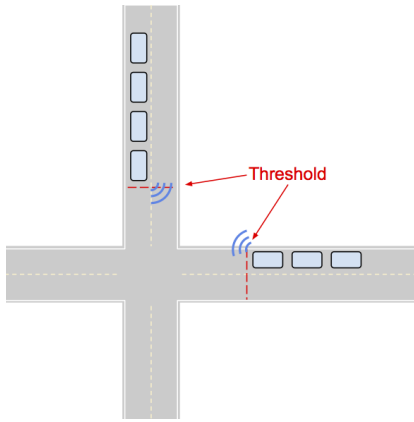


Figure 3.1: Leader election at intersection

Figure 3.1 shows a scenario where vehicles are approaching an intersection, creating crossing conflict. The cars are passing a threshold distance from the intersection. At this point, they should start looking for a VTL, which in this scenario does not exist. The approaching vehicles have to cooperate and elect a leader who will be responsible for establishing a VTL and broadcast traffic light messages to the surrounding vehicles. The leader is elected based on a pre-defined rule and information from the LT. If a vehicle is approaching the intersection with no crossing conflicts ahead, there is no need to create a VTL, and the vehicle may continue ahead.

The elected leader will act as a temporary VTL and should be presented a red light and position itself as close to the intersection as possible. This vehicle is responsible for broadcasting VTL messages to the network. This situation is shown in Figure 3.2

While the elected leader broadcast the VTL messages to surrounding vehicles, the other vehicles act as passive nodes in the protocol, listening to the traffic light messages and just presenting them to the driver through the in-vehicle displays [5].

Once the current light cycle is finished, or there are no crossing vehicles at the green light, the leader changes the virtual traffic light message to apply the next phase. If the leader is presented a green light, there must be elected a new leader election.

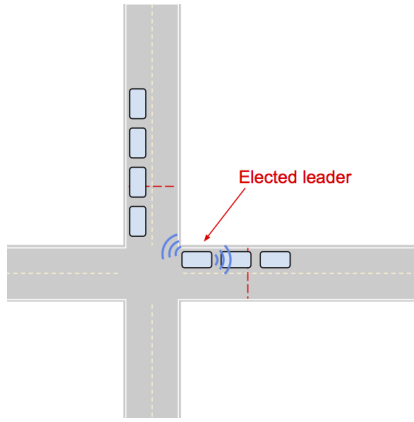


Figure 3.2: Leader broadcasting VTL messages at intersection

The situation where there are vehicles stopped at a red light at the intersection the current leader just selects one of these vehicles to become the new leader, that continues the broadcast of the VTL messages. If the VTL no longer is needed, the cycle can be interrupted, and vehicles proceed without stopping [5]. This situation is illustrated in Figure 3.3

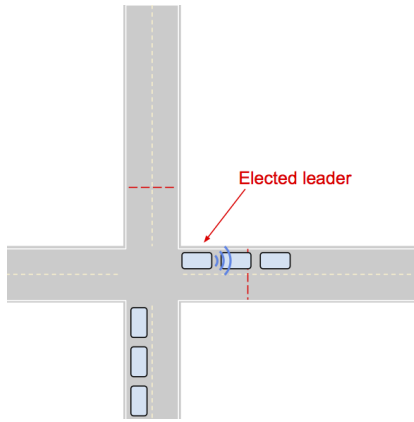


Figure 3.3: Disband of VTL at intersection

Chapter 4
Testbed Implementation

This chapter will introduce a detailed description of our testbed setup. As mentioned in Chapter 3, [5] defines some assumptions tied to the implementation of the system. These are assumptions made on the development of a real size VTL system. Naturally, therefore a miniature testbed setup has its limitations. We present a complete system architecture and how different parts of the system interact with each other. Also, we explain how we have translated the real size concept into a miniature testbed. The source code for this project can be found as appendices.

4.1 Network Topology

The VTL concept proposes a VANET communication model over DSRC. Our response to this is to configure an ad-hoc network over WLAN. Since we are using Raspberry Pi 3 for the DiddyBorg robots, we take advantage of the onboard wireless interface as our DSRC device.

The wireless network interface *wlan0* has to be configured in ad-hoc mode. This is done by editing the network interface file located at */etc/network/interfaces*. We added the following lines in our network setup:

```
auto wlan0
iface wlan0 inet static
  address 192.168.1.12      #This address has to be unique
  netmask 255.255.255.0
  wireless-channel 1
  wireless-essid adhoc_vanet
  wireless-mode ad-hoc
```

By configuring all devices in our setup the same information, assigning each device with its unique static IP address, we can communicate peer-to-peer or V2V in an

ad-hoc manner. Meaning devices seamlessly connect whenever they are in range of each other without the need for any infrastructure.

Figure 4.1 shows the ad-hoc network topology in this project. All Raspberry Pi computers are set up in ad-hoc mode with its unique IP address. Making them able to communicate with every device in its vicinity.

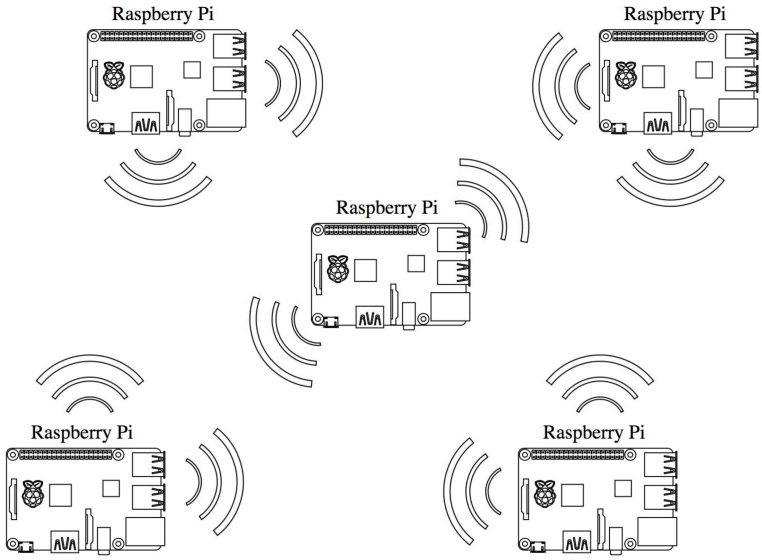


Figure 4.1: Ad-hoc network with Raspberry Pi computers

4.2 System Architecture

The system we present is separated into several modules for handling different parts of the system. All software modules are developed with Python [6] as the programming language. A high-level system architecture is illustrated in Figure 4.2

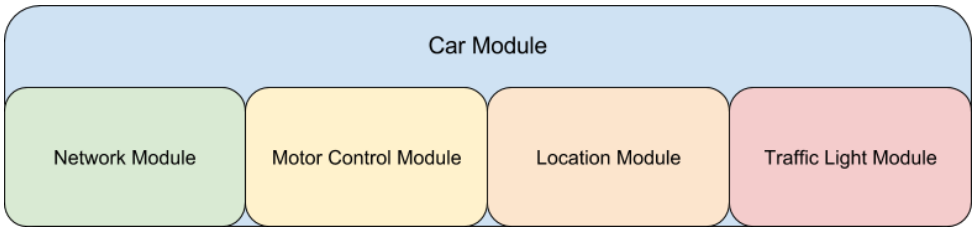


Figure 4.2: High-level system architecture

4.2.1 Network Module

The network module of the system can be translated to act as the DSRC device of the car. It handles all the communication with the network, both sending and receiving messages. Therefore the module is separated into two parts, one for sending and one for receiving messages. The code is found in Appendix A.

The VTL concept requires us to broadcast messages to the network through beaconing. This feature is needed both when sending updates on the vehicles location and when transmitting traffic light messages. To handle this situation we are using the Python socket module. This allows us to bind a socket to an IP address and a port for sending and receiving packets across the network. In our setup, we are using the User Diagram Protocol (UDP) for sending messages. We are then able to send packets rapidly over the network without needing a confirmation message from the destination. To make the socket able to broadcast messages to all devices within its vicinity, the socket has to be tied up to the network broadcast address. In our network setup, the broadcast address is *192.168.1.255*. Upon receiving messages from the network, we establish a server socket listening to messages sent to its IP address at a particular port.

Figure 4.3 shows a sequence diagram of a scenario where a message is sent to the broadcast address and passes through the ad-hoc network. The Raspberry Pi A is the origin of the message sent to broadcast. The message is then sent to all devices in range of the source. From the Raspberry Pi B and C's point of view, the message is sent directly from the Raspberry Pi A. In this scenario, we have asked the receiving entities to respond to the message received. The entities receiving the message sends a response directly back to the originating entity. A Wireshark capture of this message sequence is traced and shown in Figure 2.6 in Chapter 2.

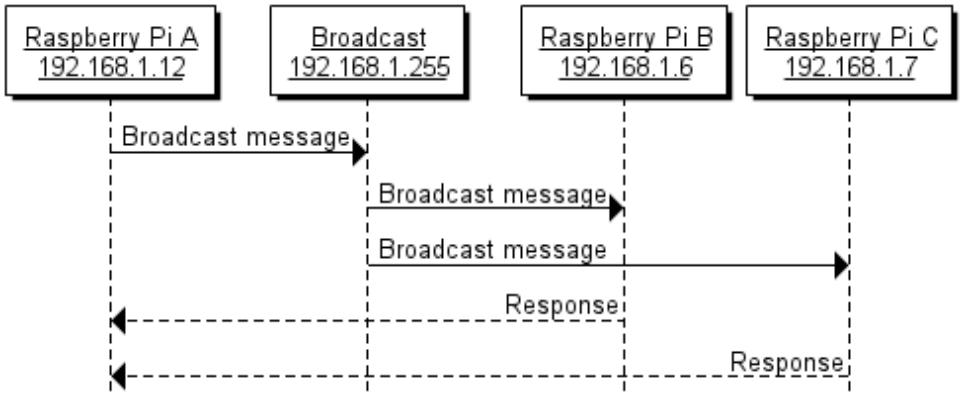


Figure 4.3: Sequence diagram of a broadcast message scenario

4.2.2 Motor Control Module

The motor control module is the part of the system that controls the movement of the robot. It initializes the motors connected to the robot connecting to the PicoBorg Reverse. Handling movement forwards and backwards together with rotation to both sides is the main tasks of the module. The motors are controlled by providing equal voltage to all motors at the same time in a set time interval. Movement in a straight forward direction is performed by ensuring that the motors on each side of the robot are rotating the same way. Rotation of the robot is done by making the motor on each side to rotate in different directions. This makes the robot able to turn 360 degrees on the spot. This module also handles calibration of the motors. By calculating and setting the time the robot takes to do specific movements, we can give instructions to the robot. As an example, if we want to move 10 cm forward and rotate 90 degrees to the left, it will be possible to do this with proper calibration and the right commands. The motor control module is provided in Appendix B

4.2.3 Location Module

The location module handles the location features of the system. The VTL system is based on the assumption that every vehicle is equipped with a GPS with lane-level accuracy. Also, it requires all vehicles to share the same digital road map. In our testbed, the DiddBorg robots did not have an on-board GPS device. Acquiring a GPS device precise enough achieve indoor lane-level accuracy proved to be difficult.

In the testbed setup, we faced this challenge with what we have called a relative GPS. During the initialization phase of the system, the location module is loaded with an internal map and an initial position. The internal map contains information of the road structure and the available VTL intersections presented as a coordinate system. At the start of the system, we provide an initial position of the vehicle. The position is updated whenever a movement is made. This way we can keep track our position relative to our initial position. For debug reasons, we created a function to write the map to the console. A screenshot of the internal map in a test scenario is shown in Figure 4.4. Here the numbers symbolise different things. The 1s is supposed to represent the road structure. The 3s show where the intersections are placed. The number eight show where our car is located. The 0s are meaning out of bounds. In scenarios where there are other cars in the system, they are represented by the number 4.

Another task assigned to the location module is keeping track of other cars in the system maintaining the LT. The LT keeps track of what intersection all other cars are heading against, in what direction and their distance to the intersection. Here the distances are calculated using Manhattan distance, the sum of the horizontal

and vertical coordinates. The location module engages the leader election if there is a need for creating a VTL.

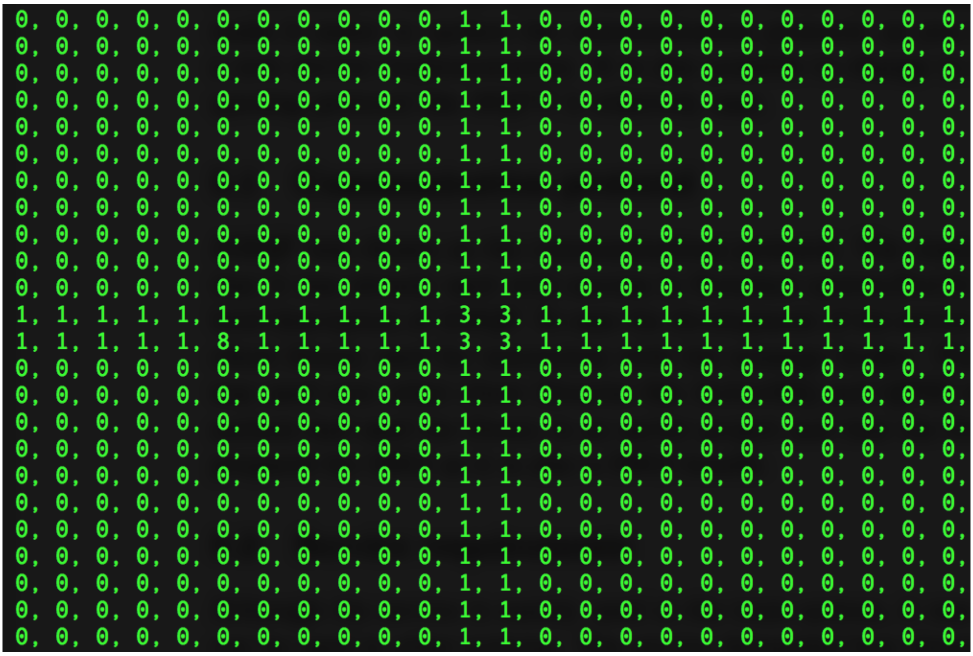


Figure 4.4: Screenshot of the internal map in a test scenario

4.2.4 Traffic Light Module

The traffic light module is activated if this vehicle is elected as the leader of the VTL at an intersection. The module handles the initialization of the VTL and the broadcasting of VTL messages to the network.

Whenever a VTL is created, the vehicle creating it gives them self a red light and start broadcasting messages. When the light cycle is over, or there are no more crossing vehicles the VTL is disbanded. If there still are vehicles waiting at the intersection when this leader is leaving, the responsibility of the VTL is handed over to another vehicle.

4.2.5 Car Module

The car module is the main module of our system. As shown in Figure 4.2 it connects all the other modules together managing the flow of data through the system. This module functions similar to how an actual car works. When the program is started,

the motors and all the different modules of the car are instantiated. To handle all the different processes we enable the use of threading in our system.

Upon the initialization of the car module we create two instances of the network module (one for sending and one for receiving messages) and one instance of the location, motor control and traffic light modules. Figure 4.5 illustrates how important data flow through the system.

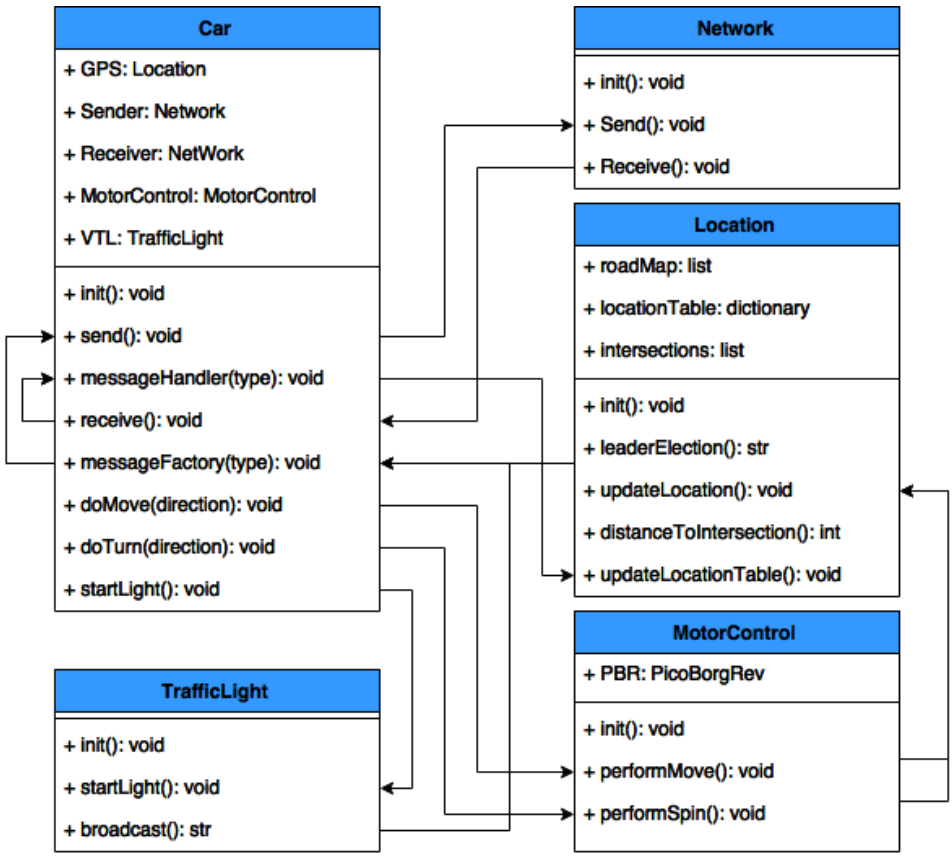


Figure 4.5: Data flow of important processes between modules

4.2.6 Data Flow

Using Figure 4.5 as a base, we are going to take a look at some of the functions and what mechanisms they initiate. At first, we start listening to messages on the network. This is done by starting a **receive()** thread in the car module allowing us to continuously handle incoming messages. Whenever a message is received, it is sent to the **messageHandler()**. Based on the message topic, an action is made,

and the information is taken care of. This may be messages about other vehicle's location or VTL messages.

To start a movement the **doMove(direction)** is called sending the task to the **performMove()** function in the motor control unit. Making the vehicle move a certain distance in a certain direction. When a movement is done the vehicle updates its location and generates a call on the **updateLocation()** function in the location module. This again forwards the process to the **messageFactory(type)** function which initiates a broadcast of the vehicle's location to the network.

When approaching an intersection, the location module checks its LT for potential crossing conflicts. If there is an ensuing conflict, the vehicle closest to the intersection does the leader election. This is a calculation based on several parameters such as distance to intersection and the number of cars approaching from each side. When the leader election is made, the leader is notified. Then the VTL leader starts a new thread with the **startLight()** function broadcasting VTL messages to the network. When the leader is leaving a new leader election is made if necessary.

4.3 Challenges and Lessons Learned

During the implementation of the system, we encountered some challenges. First off, it seemed almost impossible to configure and make the ad-hoc network function properly. A lot of time were spent on this part of the study. After trying several configurations, this problem turned out to be a hardware problem with the WLAN dongles connected our Raspberry Pi 2 computers. This problem was not encountered after upgrading to Raspberry Pi 3.

Secondly, the system is made from nothing with few projects similar to this. At times it was frustrating having to improvise solutions as the internal GPS system. In hindsight, this turned out to be both challenging and interesting.

The author had little or nothing experience with robot programming, especially not building and assembling the robots. Even though the final system may look like a simple solution, an extensive amount of time has been used to find solutions to occurring problems. Resulting in the implementation phase taking longer time than expected.

Chapter 5
Simulation

This chapter presents a simulation scenario, how the simulation was conducted, and the results of the simulation phase.

5.1 Simulation Setup

In our simulation setup, we have assembled three DiddyBorg robots. All three robots are set up with the system presented in Chapter 4. We run simulations on the system implementation to measure the effectiveness and feasibility of the VTL protocol. An internal map corresponding to the Figure 5.1 is loaded onto the DiddyBorg cars. The map has two intersections with the possibility to create a VTL if needed. The three coloured arrows illustrate three different routes on the map. Each of our three robots is assigned its route to follow autonomously from its initial position.

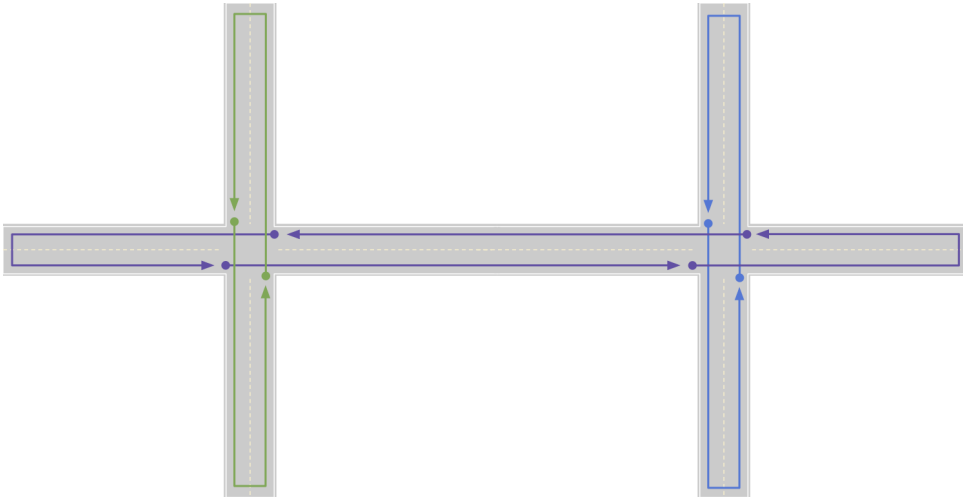


Figure 5.1: Test scenario

5.2 Scenarios

The simulation was conducted as three different scenarios. We measure the time it takes for a car to drive from one side of the map over to the other side of the map. The cars will be passing intersections on its way. The first scenario does not have any traffic lights. The second scenario is a case where we have the possibility to establish VTLs. In the third and last scenario, we have standard centralised traffic lights. We measure the time it takes to cross the map, calculating the average of 50 crossovers for each car.

5.3 Results

After running the simulation, we calculated the average time in each case. The results of the simulation are shown in Table 5.1.

Table 5.1: Simulation results

Car	**Average time without traffic lights**	**Average time with VTL**	**Average time with standard traffic light**
Green car	28.7 s	35.5 s	42.3 s
Blue car	28.9 s	34.6 s	43.5 s
Purple car	44.6 s	52.4 s	61.3 s

From the numbers in Table 5.1 we derive that the effectiveness of the VTL protocol in our testbed is approximately 18% more than the case of standard traffic lights.

Chapter 6
Discussion, Conclusion and Further Work

There is a great belief that the solution to problems of transportation is through autonomous vehicles. The reason for this is that driverless cars will enable many new possibilities, while they will make people's lives easier. Theoretically, these cars can drive closer together since they do not rely on a human reaction time. This applies also in critical danger situations, where elimination of human error theoretically could save more lives.

ITS makes the system of transportation more efficient. The system is intended for cars to communicate and adapt to each other, not only orient themselves by the information they register and process themselves using cameras, sensors and lidar. When vehicles communicate with each other, the system can be useful.

Although the VTL system requires all other vehicles to have the same on-board equipment, the technology within the automobile industry developing fast. VTL systems do not require expensive equipment, in fact it is rather cheap.

Another aspect of the VTL is that the improvement in traffic flow do not only apply at one individual intersection. The system has great scalability and able to create VTL intersections almost anywhere. This allows us to maximise the throughput of the complete road network, rather than the reduced number of road junctions that are currently managed by traffic lights [5].

6.1 Conclusion

In this study, we present an implementation of a Virtual Traffic light VTL testbed we have designed and made from scratch using DiddyBorg robot cars. We have configured the DiddyBorg robots to communicate across an ad-hoc network. A simulation has been conducted on the implemented system demonstrating the feasibility and effectiveness of the VTL concept. The simulation phase determine that our implementation of aVTL is approximately 18% more efficient compared to standard

traffic lights. These are promising results and show that the VTL concept is feasible for implementation.

6.2 Further Work

This study was limited to three robot cars. It would have been interesting to see the system implemented in a larger scale having more vehicles acting together. One could also try to measure the feasibility in other scenarios with more intersections. Another interesting outlook would be to reproduce this study using actual GPS devices on the robots.

References

[1] Electromotor, Z. Zgb25rq - pmdc spur gear motor. http://wzh001.gotoip55.com/upload/file/ZGB25RQ%20&%20ZGA25RP(1).pdf. Visited: 2017-01-10.

[2] Federation, N. A. Byfolk bruker hvert fjerde minutt i kø. https://www.naf.no/om-naf/naf-mener/sa-lenge-star-pendlere-i-byene-i-ko/. Visited: 2017-01-15.

[3] Fernandes, R. J. (2009). *Vanet-enabled in-vehicle traffic signs*. Ph. D. thesis, Master's thesis, University of Porto.

[4] Ferreira, M. and P. M. d'Orey (2012). On the impact of virtual traffic lights on carbon emissions mitigation. *IEEE Transactions on Intelligent Transportation Systems 13*(1), 284–295.

[5] Ferreira, M., R. Fernandes, H. Conceição, W. Viriyasitavat, and O. K. Tonguz (2010). Self-organized traffic control. In *Proceedings of the seventh ACM international workshop on VehiculAr InterNETworking*, pp. 85–90. ACM.

[6] Foundation, P. S. Python: about python. https://www.python.org/about/. Visited: 2017-01-28.

[7] Foundation, R. P. Raspberry pi 3 model b. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. Visited: 2017-01-15.

[8] Foundation, R. P. What is a raspberry pi? https://www.raspberrypi.org/help/what-is-a-raspberry-pi/. Visited: 2017-01-15.

[9] Foundation, W. Wireshark: about wireshark. https://www.wireshark.org. Visited: 2017-01-18.

[10] Moon, S. Programming udp sockets in python. http://www.binarytides.com/programming-udp-sockets-in-python/. Visited: 2016-12-27.

[11] Nakamurakare, M., W. Viriyasitavat, and O. K. Tonguz (2013). A prototype of virtual traffic lights on android-based smartphones. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2013 10th Annual IEEE Communications Society Conference on*, pp. 236–238. IEEE.

[12] Neudecker, T., N. An, O. K. Tonguz, T. Gaugel, and J. Mittag (2012). Feasibility of virtual traffic lights in non-line-of-sight environments. In *Proceedings of the ninth ACM international workshop on Vehicular inter-networking, systems, and applications*, pp. 103–106. ACM.

[13] Ødegaard, L. O. Intelligente transportsystemer. http://www.ramboll.no/services/infrastruktur_og_transport/trafikkregulering-trafikkplan-sikkerhet/its. [Visited: 2017-01-15.

[14] PiBorg.org. Battborg - power your raspberry pi from aa batteries. https://www.piborg.org/battborg. Visited: 2017-01-10.

[15] PiBorg.org. Diddyborg - the most powerful raspberry pi robot available. https://www.piborg.org/diddyborg. Visited: 2017-01-10.

[16] PiBorg.org. Diddyborg - the most powerful raspberry pi robot available. https://www.piborg.org/diddyborg/install. Visited: 2016-12-27.

[17] PiBorg.org. Picoborg reverse - advanced motor control for your raspberry pi. https://www.piborg.org/picoborgrev. Visited: 2017-01-10.

[18] Raspbian. Installing operating system images. https://www.raspberrypi.org/downloads/raspbian/. Visited: 2017-01-15.

[19] Sill, S. Dsrc: The future of safer driving. http://www.its.dot.gov/factsheets/dsrc_factsheet.htm. Visited: 2017-01-20.

Appendix A
Pyhton Network Module

Code inspired by forum post about UDP sockets in Python found here [10]

```
import socket
import sys

class Receive(object):

    HOST = ''
    PORT = ''
    s = ''

    _ACTIVE = False
    def get_ACTIVE(self):
        return self._ACTIVE
    def set_ACTIVE(self,state):
        self._ACTIVE = state
    ACTIVE = property(get_ACTIVE, set_ACTIVE)

    def __init__(self):
        self.HOST = ''
        self.ACTIVE = True

    def setPort(self,port):
        self.PORT = port

    def bind(self,port):
        self.setPort(port)
        try :
            self.s = socket.socket(socket.AF_INET,
```

```
            socket.SOCK_DGRAM)
            print 'Socket created'
        except socket.error, msg :
            print 'Failed to create socket. Error Code : '
            + str(msg[0]) + ' Message ' + msg[1]
            sys.exit()

        # Bind socket to local host and port
        try:
            self.s.bind((self.HOST, self.PORT))
        except socket.error , msg:
            print 'Bind failed. Error Code : ' + str(msg[0])
            + ' Message ' + msg[1]
            sys.exit()

        print 'Socket bind complete'

    def listen(self):
        d = self.s.recvfrom(1024)
        data = d[0]
        if data.strip()=='quit':
            self.ACTIVE = False
        return d

    def setPort(self,port):
        self.PORT = port

    def setActive(self,state):
        self.ACTIVE = state

    def close(self):
        self.s.close()

class Send(object):

    HOST = ''
    PORT = ''
    s = ''

    def __init__(self,port):
        self.HOST = '192.168.1.255'
```

```
        self.PORT = port
        # create datagram udp socket
        try:
            self.s = socket.socket(socket.AF_INET,
            socket.SOCK_DGRAM)
            self.s.setsockopt(socket.SOL_SOCKET,
            socket.SO_BROADCAST, 1)
        except socket.error:
            print 'Failed to create socket'
            sys.exit()

    def sendPacket(self,message):
        try :
            self.s.sendto(message, (self.HOST, self.PORT))

        except socket.error, message:
            print 'Error Code : ' + str(message[0])
            + ' Message ' + message[1]
            sys.exit()

    def close(self):
        self.s.close()
```

Appendix B

Pyhton Motor Control Module

Code inspired by a DiddyBorg movement example code from piborg.org [16]

```
import PicoBorgRev
import time
import math
import sys

class MotorControl(object):

    def __init__(self):
    # Setup the PicoBorg Reverse
        self.PBR = PicoBorgRev.PicoBorgRev()
        self.PBR.Init()
        if not self.PBR.foundChip:
            self.boards = PicoBorgRev.ScanForPicoBorgReverse()
            if len(self.boards) == 0:
                print 'No PicoBorg Reverse found,
                check you are attached :)'
            else:
                print 'No PicoBorg Reverse at address \%02X,
                but we did find boards:' \% (self.PBR.i2cAddress)
                for board in self.boards:
                    print '    \%02X (\%d)' \% (board, board)
                print 'If you need to change the I2C address
                change the setup line so it is correct, e.g.'
                print 'PBR.i2cAddress = 0x\%02X' \% (self.boards[0])
            sys.exit()
        self.PBR.SetCommsFailsafe(False)
        self.PBR.ResetEpo()
```

```
        # Movement settings
        self.timeForward1m = 7.2
        self.timeSpin360   = 6.2

        # Power settings
        self.voltageIn = 12.0       # Total battery voltage
        self.voltageOut = 6.0       # Maximum motor voltage

        # Setup the power limits
        if self.voltageOut > self.voltageIn:
            self.maxPower = 1.0
        else:
            self.maxPower = self.voltageOut / float(self.voltageIn)

    # Function to perform a general movement
    def PerformMove(self,driveLeft, driveRight, numSeconds):

        percent = 0.8
        speed = self.maxPower*percent

        self.PBR.SetMotor1(driveRight * speed)
        self.PBR.SetMotor2(-driveLeft * speed)

        # Wait for the time
        time.sleep(numSeconds)
        # Turn the motors off
        self.PBR.MotorsOff()

    # Function to spin an angle in degrees
    def PerformSpin(self,angle):
        if angle < 0.0:
            # Left turn
            driveLeft  = -1.0
            driveRight = +1.0
            angle *= -1
        else:
            # Right turn
            driveLeft  = +1.0
            driveRight = -1.0
        # Calculate the required time delay
```

```
    numSeconds = (angle / 360.0) * self.timeSpin360
    # Perform the motion
    self.PerformMove(driveLeft, driveRight, numSeconds)

# Function to drive a distance in meters
def PerformDrive(self,meters):
    if meters < 0.0:
        # Reverse drive
        driveLeft  = -1.0
        driveRight = -1.0
        meters *= -1
    else:
        # Forward drive
        driveLeft  = +1.0
        driveRight = +1.0
    # Calculate the required time delay
    numSeconds = meters * self.timeForward1m
    # Perform the motion
    self.PerformMove(driveLeft, driveRight, numSeconds)
```

Appendix C

Pyhton Location Module

```
class Location(object):
#   Sets scenario, internal map and initial position of car

    locationTable = {}

    map = [[0 for j in range(61)] for i in range(40)]
    intersections = []

    _myPos = (None,None)
    def get_myPos(self):
        return self._myPos
    def set_myPos(self,val):
        self._myPos = val
    myPos = property(get_myPos, set_myPos)

    def __init__(self, x,y, scenario):
        self.initMap(scenario)
        self.setInitPos(x,y)
        self.initIntersections(scenario)

    def initMap(self,scenario):
        if scenario == 1:
            self.map[19] = [1 for j in range(61)]
            self.map[20] = [1 for j in range(61)]

            for i in range(40):
                self.map[i][19] = 1
                self.map[i][20] = 1
                self.map[i][40] = 1
```

```
                self.map[i][41] = 1
        return self.map

    def initIntersections(self, scenario):
        if scenario == 1:
            self.intersections.append((19,19,1,1))
            self.intersections.append((19,20,1,2))
            self.intersections.append((20,19,1,3))
            self.intersections.append((20,20,1,4))

            self.intersections.append((19,40,2,1))
            self.intersections.append((19,41,2,2))
            self.intersections.append((20,40,2,3))
            self.intersections.append((20,41,2,4))
        return

    def setInitPos(self,x,y):
        self.myPos = (x,y)
        return

    def drawMap(self):
        tempMap = self.initMap(1)
        self.addIntersections(tempMap)
        self.addlocationTable(tempMap)
        self.addMyPos(tempMap)
        tempMap[0][0] += 1
        for row in tempMap:
            print (row)
        #print self.myPos
        print ('\n')

    def addMyPos(self,map1):
        map1[self.myPos[1]][self.myPos[0]] = 8
        return map1

    def myPosX(self):
        return self.myPos[0]

    def myPosY(self):
        return self.myPos[1]
```

```
def addIntersections(self,map1):
    for inter in self.intersections:
        interX=inter[0]
        interY=inter[1]
        map1[interX][interY]= 3
    return map1

def addlocationTable(self,map1):
    for car,pos in self.locationTable.iteritems():
        carX=pos[0][0]
        carY=pos[0][1]
        map1[carX][carY]= 4
    return map1

def updatelocationTable(self,idC,pos,dist,interID,direction,
approaching):
    self.locationTable[idC]=[pos,dist,interID,direction,
    approaching]

def deleteCar(self,idC):
    del self.locationTable[idC]

def updatePos(self,direction):
    if direction=='u':
        self.myPos = (self.myPos[0], self.myPos[1]-1)
        return
    elif direction=='d':
        self.myPos = (self.myPos[0], self.myPos[1]+1)
        return
    elif direction=='l':
        self.myPos = (self.myPos[0]-1, self.myPos[1])
        return
    elif direction=='r':
        self.myPos = (self.myPos[0]+1, self.myPos[1])
        return
    else:
        return

def distanceToIntersection(self):
    minDist = 120
    for inter in self.intersections:
```

```
            interX = inter[1]
            interY = inter[0]
            dist = abs(self.myPosX()-interX)
            +abs(self.myPosY()-interY)
            if (dist< minDist):
                minDist=dist
                interID = inter[2]
                interDir = inter[3]
        return [minDist,interID,interDir,0]
```

Appendix D
Pyhton Car Module

```
import time as t
import threading
import Location as Loc
import Networking as Net
import sys
import MotorControl as MC
'''
Class Car
'''
class Car(object):

    ME = ''
    GPS = ''
    RECEIVE = ''
    SEND = ''
    MOTORCONTROL = ''
    VTL = ''


    def __init__(self,car):
        if car == 1:
            self.ME = '192.168.1.6'
            self.GPS = Loc.Location(0,20,1) #car1
        elif car == 2:
            self.ME = '192.168.1.7'
            self.GPS = Loc.Location(19,0,1) #car2
        else:
            self.ME = '192.168.1.12'
            self.GPS = Loc.Location(40,0,1) #car3
```

```
        self.RECEIVE = Net.Receive()
        self.SEND = Net.Send(8888)
        self.VTL = Light()
        #self.MOTORCONTROL = MC.MotorControl()

    def main(self,car):
        t1 = threading.Thread(target = self.receive)
        t2 = threading.Thread(target = self.startLight,
        args = (0,1,1,1))
        t1.start()

        if car == 1:
            self.loop1() #car1
        else:
            self.loop2() #car2 and car3

        #self.RECEIVE.ACTIVE == False
        #self.MOTORCONTROL.PBR.MotorsOff()
        #sys.exit()


    def send(self,message):
        self.SEND.sendPacket(message)

    def receive(self):
        self.RECEIVE.bind(8888)
        while self.RECEIVE.ACTIVE:
            d = self.RECEIVE.listen()
            self.messageHandler(d)

    def messageFactory(self,mType):
        if mType == 1:
            m = str(mType)+':'+str(self.GPS.myPos[1])+':'
            +str(self.GPS.myPos[0])
            return m
        elif mType == 2:
            m = str(mType)+':'+str(self.GPS.myPos[1])+':'
            +str(self.GPS.myPos[0])
            return m
        elif mType == 3:
```

```
            m = str(mType)+':'+str(self.intersectionID)+':'
            +str(self.state1)+':'+str(self.state2)+':'
            +str(self.state3)+':'+str(self.state4)
            print m
            return m
        elif mType == 4:
            distance = self.GPS.distanceToIntersection()
            m = str(mType)+':'+str(self.GPS.myPos[1])+':'
            +str(self.GPS.myPos[0])+':'+str(distance[0])+':'
            +str(distance[1])+':'+str(distance[2])+':'+str(distance[3])
            print m
            return m
        elif mType == 5:
            m = str(mType)+':'+str(self.GPS.myPos[1])+':'
            +str(self.GPS.myPos[0])
            t2.start()
            return m
        else:
            return

    def messageHandler(self,message):
        data = message[0].split(':')
        carID = message[1][0]
        #if carID == self.ME:
        #    return
        mType = data[0]

        if mType == '3': # VTL update
            self.isGreen(data[1:])
            return
        elif mType == '4': # Location update
            if carID == self.ME:
                return
            carPos = (int(data[1]),int(data[2]))
            carDist = int(data[3])
            interID = int(data[4])
            carDir = int(data[5])
            approach = int(data[6])
            self.GPS.updatelocationTable(carID,carPos,carDist,
            interID,carDir,approach)
            return
```

```
        elif mType == '5': # Message of starting a VTL
            startLight(myDirection = self.GPS.distanceToIntersection()[2])
            return
        else:
            return

    def strToBool(self,bStr):
        if bStr == 'True':
             return True
        else:
            return False

    def doMove(self,direction):
        distance = self.GPS.distanceToIntersection()
        self.GPS.drawMap()
        if distance[0] == 1 and not self.stateMe:
            while not self.stateMe:
                print 'stopping'
                t.sleep(0.5)
        elif distance[0] <= 10:#and not leaving:
            print 'approaching'
        else:
            print 'far away'


        self.GPS.updatePos(direction)
        self.send(self.messageFactory(4))
        self.MOTORCONTROL.PerformDrive(-0.1)

    def doTurn(self,direction):
        #self.GPS.drawMap()
        self.GPS.updatePos(direction)
        self.send(self.messageFactory(4))
        print 'turning'
        self.MOTORCONTROL.PerformSpin(-100)
        self.MOTORCONTROL.PerformDrive(-0.05)
        self.MOTORCONTROL.PerformSpin(-100)

    def loop1(self):
        while True:
            for i in range(60):
```

```
            self.doMove('r')
        self.doTurn('u')
        for i in range(60):
            self.doMove('l')
        self.doTurn('d')

def loop2(self):
    while True:
        for i in range(39):
            self.doMove('d')
        self.doTurn('r')
        for i in range(39):
            self.doMove('u')
        self.doMove('l')

STATE1 = ['010', 2]
STATE2 = ['100', 10]
STATE3 = ['110', 2]
STATE4 = ['001', 10]
state1 = False
state2 = False
state3 = False
state4 = False
intersectionID = None

def leaderElection():

    clusterMe
    clusterYou

    return

def isLeaving(self):
    return False

def isGreen(self,lights):
    myDirection = self.GPS.distanceToIntersection()

    if myDirection[1]:
        self.state1 = self.strToBool(lights[0]])
        self.state3 = self.strToBool(lights[1])
```

```
            self.state2 = self.strToBool(lights[2])
            self.state4 = self.strToBool(lights[3])
            self.stateMe = self.strToBool(lights[myDirection[2]])

#TRAFFIC LIGHT PART OF SYSTEM
    def startLight(self, interID, state,clusterMe,clusterYou):
        self.state1 = False
        self.state3 = False
        self.state2 = False
        self.state4 = False
        self.intersectionID = interID
        myDirection = self.GPS.distanceToIntersection()[2]
        if (myDirection == 1 or myDirection == 3):
            odd = True
        else:
            odd = False

        self.STATE2[1] = clusterMe*3
        self.STATE4[1] = clusterYou*3
        state = int(state)
        #while True:
        state = state % 4
        if state == 0:
            self.state1 = False
            self.state3 = False
            self.state2 = False
            self.state4 = False
            self.send(self.messageFactory(3))
            t.sleep(self.STATE1[1])
            t.sleep(2)
            state+=1
        if state == 1:
            if odd:
                self.state1 = False
                self.state3 = False
                self.state2 = True
                self.state4 = True
            else:
                self.state1 = True
                self.state3 = True
                self.state2 = False
```

```
                self.state4 = False
            self.send(self.messageFactory(3))
            crossingCars = True
            while crossingCars:
                t.sleep(self.STATE2[1])
                crossingCars = False
            state+=1
        if state == 2:
            self.state1 = False
            self.state3 = False
            self.state2 = False
            self.state4 = False
            self.send(self.messageFactory(3))
            t.sleep(self.STATE3[1])
            state+=1
        if state == 3:
            if not odd:
                self.state1 = False
                self.state3 = False
                self.state2 = True
                self.state4 = True
            else:
                self.state1 = True
                self.state3 = True
                self.state2 = False
                self.state4 = False
            self.send(self.messageFactory(3))
            crossingCars = True
            while crossingCars:
                t.sleep(self.STATE4[1])
                crossingCars = False
            t.sleep(2)
            state+=1

if __name__ == '__main__':
    myCar = Car(1)
    myCar.main(1)
```