ECE 4600 - GROUP G05

Smart Traffic Network for Autonomous Vehicles

Huy Hoang Bui

Hyunhoon Jo

Farhan Rahman

Aleksa Svitlica

Wyatt Thomson

supervised by

Dr. Ekram Hossain

March 2018

Abstract

The technology behind autonomous vehicles is steadily advancing due to the advantages that autonomous vehicles bring. Our project focuses on one particular advantage, using network communication between vehicles to coordinate their behaviour and optimize traffic flow. We demonstrate the capabilities of our system with a small scale four-way intersection using small robot vehicles built for the project. The vehicles are connected together by a base station through a Bluetooth network. We developed a simulation of our demonstration in Unity, first to test different methods of coordinating traffic, then to interface with the communication system to represent the real world demonstration and determine what commands to send to the vehicles. We were able to successfully simulate a traffic network and test two intersection routing algorithms. We built five miniature autonomous vehicles that were capable of executing commands sent over Bluetooth. Our communication system was tested and maintains four Bluetooth connections while receiving updates from the vehicles and interfacing with the simulation.

Contribution

The contribution of our project is to show that even on a small budget we can use the networking of autonomous vehicle concept to optimize traffic flow. While certain aspects of the project would have to be changed in order to scale the project up to a real world scenario, such as the Bluetooth connection and the network architecture, the core idea of project, networking vehicles together and coordinating their movements, is the real value of this project.

	Huy Hoang Bui	Hyunhoon Jo	Farhan Rahman	Aleksa Svitlica	Wyatt Thomson
Research communication protocol	✓				
Research power supply		✓	✓		
Research hardware requirements			✓		
Research simulation engine				✓	✓
Research pathfinding	✓				✓
Simulation coding				✓	
Vehicle hardware design		✓	✓		
Vehicle chassis design			✓		
Vehicle assembly			✓		
Pathfinding scripting/testing				✓	✓
Collision Avoidance scripting/testing	✓			✓	✓
Power supply testing		✓			
Communication system and Simulation interfacing	✓			✓	
Vehicle motor control scripting/testing			✓		
Bluetooth communication testing	✓				
Build track		✓			
Full system integration	✓	✓	✓	✓	✓

Table 1: Contributions

Contents

List of Figures

List of Tables

Nomenclature

ACK - Acknowledge
API - Application programming interface
ARM - Type of Computer Processor Architecture
BT - Bluetooth
DC - Direct Current
GUI - Graphical User Interface
GPIO - General-purpose input/output
GPS - Global Positioning System
IC - Integrated Circuit
IR - Infrared Radiation
JSON - JavaScript Object Notation
LED - Light Emitting Diode
MCU - Microcontroller unit
OS - Operating system
PCB - Printed Circuit Board
PIC - Brand of Microcontrollers by Microchip™
PWM - Pulse Width Modulation
SSH - Secure Socket Shell
USB - Universal Serial Bus
WAVE - Wireless Access in Vehicular Environments

Chapter 1

Introduction

The report is divided into three main sections each discussing a component of the project:

- The simulation component, which keeps track of all the vehicles in the demonstration and coordinates them. The simulation is responsible for instructing the vehicles on what movements to make and ensures that collisions are prevented.
- The communication system chapter discusses how the network is implemented and structured and describes the purpose of the base station. The communication system transfers information between the simulation and physical vehicles. The simulation and communication system together are called the base station.
- A chapter devoted to the design of vehicles themselves and the track that they drive on.

These three components together form our project. The vehicles receive commands generated by the base station and perform those commands. The vehicles then send their status back through the base station, allowing the simulation to match the physical vehicles. The simulation coordinates the movements of each vehicle and sends the next command to follow.

The fifth chapter outlines the lessons learned and extensions beyond our scope that can be made.

1.1 Goals and Motivation

The recent advances in autonomous vehicles allow a complete rethinking of the way we manage the flow of traffic through urban centres. This project focuses on the four-way intersection as

an example to show the capabilities of a network of autonomous or semi-autonomous vehicles. With the ability to give vehicles specific commands and coordinate them with each other, we believe that human-oriented traffic systems, such as stop signs and traffic lights, will become unnecessary. The goal of this project is to provide a small scale proof of concept of a network of autonomous vehicles and demonstrate an effective solution on a small budget with limited resources. We chose this project for four main reasons:

- The mix of electrical and computer engineering components.
- The network communication aspect.
- The recent surge of interest in autonomous vehicles.
- The challenge that the project presented.

1.1.1 Goals Achieved

Simulation:

- Simulate a figure eight path and vehicles with the Unity graphics engine.
- Implemented collision avoidance and intersection routing.

Communication System:

- Able to create and maintain four Bluetooth connections.
- Receive status updates from all vehicles.

Autonomous vehicles:

- Wrote motor control software which allows for forward, left, right and stop commands.
- Receives command from the base station and executes these commands sequentially.
- Built five fully functional robots.

Demonstration:

- Built a figure eight track.

Some of the most significant challenges we faced includes the mechanical design of the vehicles as well as the track, the networking of multiple vehicles to a single server, and developing an efficient way to route vehicles through an intersection. All of these challenges and our methods of overcoming will be described in more detail in this report.

1.2 Real World Applications

Our project is flexible in its applicability as many elements can be scaled up. The Bluetooth communication protocol was chosen only for the purposes of our demonstration, as the number of connections it can maintain and its effective range are not suitable for the distances between vehicles in an intersection. A different protocol could be easily used instead that fits the requirements of a real world situation. With the inclusion of a more powerful processor the vehicles could increase the autonomy by implementing GPS, more sensors and computer vision. Having greater autonomy allows for a mesh network implementation in which the vehicles route themselves and communicate together to avoid collisions. A mesh network would eliminate the need for a separate base station and improve the flexibility of our implementation. The goal of our project is not to implement a fully functional traffic controlling system for automobiles, but rather to demonstrate the core concept of using a network to coordinate vehicles in a more efficient way. We based our design decisions around the demonstration, as well as keeping within the limits of our budget, while also leaving room for future extensions of our project.

The focus of this project has largely been on the future of autonomous vehicles, however, full integration of driverless vehicles into our current transportation system is still many years away. Even so, our project could still be applied to increase efficiency and driver safety on the road today. Instead of sending commands for a vehicle to carry out, our system could be used to predict collisions and send warnings to drivers before a collision occurs. Even if a driver is to ignore traffic signs, say by running through a red light, the other drivers could be warned of this ahead of time because each vehicle is aware of every other vehicles velocity and future position.

Another way our system could be used is to make recommendations to the driver whether or not the next turn they are going to make is safe. Essentially, augmenting the drivers own perception with information that may not be readily apparent, such as the speed of approaching vehicles, the presence of vehicles in blind spots, and even pedestrians and obstacles if working in conjunction with external sensors.

Chapter 2

Simulation

The simulation is a road system with vehicles rendered in a 3D graphics engine. The simulation component was conceived as a way of testing pathfinding and intersection routing algorithms prior to building the vehicles. It was intended to accomplish three main tasks and a fourth optional task:

- Navigate a simulated vehicle in a figure eight road system.
- Navigate multiple simulated vehicles in a figure eight road system.
- Navigate the simulated vehicles through the intersection without a traffic light or stop sign.
- Scale up to multiple intersections and more complex road systems (time permitting).

A figure eight was chosen as our base goal because it is the simplest possible enclosed system that has at least one intersection. The goal for the number of vehicles was left intentionally vague, because the number of vehicles we can simulate is greater than the number of vehicles we can build. The only requirement is that our routing algorithms is tested to work with the amount of vehicles we can build.

We accomplished the three main tasks and did not complete the fourth optional improvement.

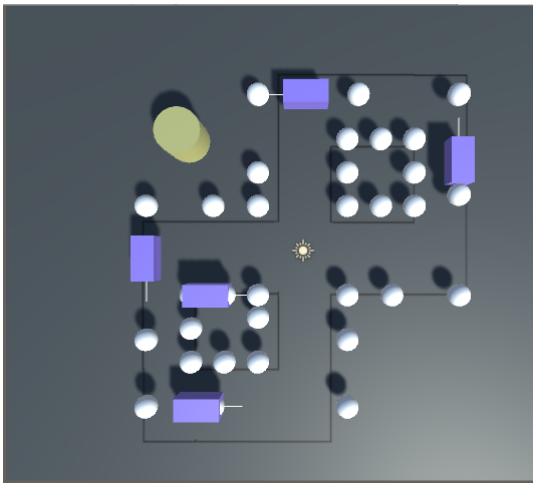

Figure 2.1: Simulation screenshot

2.1 Purpose

The simulation exists for two main reasons: testing the algorithms we use in our physical demonstration and to test even more complicated road systems if time permits. The simulation provides a platform to develop and test vehicle routing algorithms without physical vehicles. It contributes to the modular structure of our project, the vehicles, the communication system and the simulation can all be worked on concurrently to save time. Additionally, the simulation does not limit the number of vehicles which is important for testing an algorithm which will scale up to as many vehicles as possible.

2.2 Requirements and performance metrics

The performance metrics are largely the same as our goals. We set out to get several vehicles up and running to test our routing algorithms. We did not set any performance requirements for our routing algorithm because this project is meant to test whether an automated solution without stoplights is better than a traditional four-way intersection. One of the requirements is to use a graphics engine to display the vehicles rather than using a simplified GUI. A benefit of a graphics engine is that it helps with another requirement, implementing collision avoidance for the simulated vehicles. The last two requirements are the same as our goals: have multiple vehicles running and routing them through a figure eight road system.

2.3 Design

The design covers four topics:

- Graphics Engine
- Defining the track
- Code structure
- Collision detection and avoidance

2.3.1 Graphics Engine

We chose to use Unity as the graphics engine that the simulation is made in. In choosing which graphics engines to use, the two main options that we considered were Unity and Unreal Engine. Each was considered because they are free, well supported and well documented. Both support 2D and 3D but we are interested in 3D graphics for the theoretical possibility that this system could scale to road systems with different height levels, like an underpass.

We did not accurately simulate the physics of our real vehicles because the benefits would not justify the amount of time required. The benefit is increased accuracy in modeling the real system; the issue is the difficulty creating an accurate model and the routing algorithms is dependent on the graphics engine. If the routing algorithm depends on physics it needs to take into account variations between physical vehicles and the system becomes less flexible and portable.

Another substantial difference between the two engines is the language they use, Unity uses C# and Unreal Engine uses C++. Three members of our team have experience with C++ and none have experience with C#. This is a point in favor of Unreal Engine but not very important towards our decision because learning a new language is be relatively easy with the amount of documentation for C#. One of our team members already has experience with Unity which we viewed as a great asset. Additionally, through our research online there seemed to be a general conception that Unity is easier to use at the cost of being a less powerful. Given that we do not need to display complicated meshes our priority is ease of use not graphics power; this is a huge point in favor of Unity.

We preferred Unity but decided that it was worth the time investment to have one member begin working in Unity while another works in Unreal Engine. After two weeks we did a review of the two demonstrations and compared our personal experiences with each engine. Overall the Unity demonstration progressed further and had more promising documentation for our purpose. This test and the fact that one of our team members had experience with Unity finally convinced us to settle on it.

2.3.2 Defining the track

Creating a track is the first requirement of the simulation. Specifically a figure eight track which we have determined to be the simplest self-contained system with one intersection. There are two ways to approach the design of a track: define certain areas which represent the roads or create a series of points which define the roads. Defining an area to represent the road is well documented online since many vehicle projects in Unity use this approach. The advantage is that it closely emulates the real world and since many other Unity projects have used this method there is more help to be found online. Defining the roads as a series of points is easier but potentially less flexible. Another consideration is how our physical vehicles operate, which is turning on a pivot and otherwise going in a straight line; therefore a series of points fits our vehicles better. Since the goal of the simulation is to simulate our demonstration and not the real world a series of points is what we chose.

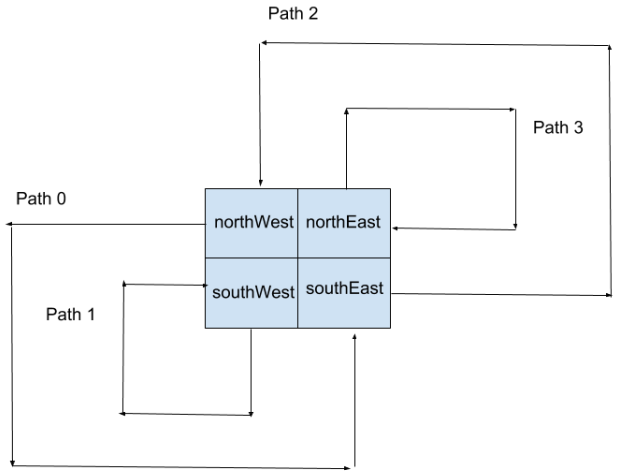

Figure 2.2: The track layout

We broke the figure eight shape down into four paths, these are outlined in the figure 2.2. In a real world road, this could be viewed as two roads with two lanes of traffic. All the paths defined start and end at the intersection because this allows the project to scale. The vehicles are designed to travel to the intersection and request a new path to follow at random. The new path will eventually loop around and return to the intersection where the process repeats. The reason this solution scales is that each intersection only needs to be aware of the paths leading to it. We can add another intersection somewhere else and it could have its own paths associated with it, although at least one would be a copy of the same points. This functions as follows: a vehicle will start at a point and have a series of points it follows to reach the intersection. Once the vehicle reaches the intersection it will receive a random choice of the four paths available and when permitted will travel through the intersection to the start of its path. Eventually, this path leads back to the intersection and the process can then repeat indefinitely.

When trying to perform pathfinding in a larger system the paths can be ignored and the intersections are all that needs to be considered. To route between intersections we will need to select the shared path which is information that is easily be stored. This design could support

pathfinding algorithms such as Dijkstra's algorithm[22]; where the intersections are the nodes and the number of points in the paths between them could represent the costs of those paths. We did not implement any of these algorithms because they are outside the scope of our project.

2.3.3 Code structure

The simulation code is broken down into three classes: vehicle, base station and intersection. This is to match the simulation to our physical setup. There are multiple vehicles, communicating with one base station which keeps track of one intersection. The number of intersections can be scaled up with our implementation but there can only be one base station.

The vehicle has four major tasks it needs to handle. The first and most basic requirement is that it must be able to move towards the points defining its path. This is accomplished using a built-in Unity function called MoveTowards which takes in the current position, the target point and the magnitude of the step required to take towards that point. While this is a built-in Unity function and would not be available if we tried to convert this to something outside of Unity, it is not an issue because it is a simple function to write separately. Subtracting the current point by the target point results in a vector pointing from the current point to the target. The normalization of this vector is passed into MoveTowards.

The second major task of the vehicle is to check for collisions and avoid them. This is a major topic which will be further explained in section 2.3.4.

The third major task is for the vehicle is to check when it is about to enter the intersection and to request its next path from the base station. This matches the physical vehicles because their positions are known when they reach a node. Specifically, there is a node placed at the end and beginning of each path right before the intersection, once a vehicle detects that it is reaching the intersection it attempts to reserve a space in the intersection for it to drive in. If the space is open, the vehicle is free to move to the next node and again request more space in the intersection if necessary. If the space is not open then the vehicle must stop and wait for that space to become available. The management of space in the intersection is handled by the intersection class.

The fourth major responsibility is converting the path made of points into a path defined by

words such as Forward, Left and Right. This is necessary for integrating with the real vehicles and will be explained more in section 2.4.

The base station class is responsible for four main duties:

- Storing the paths which the vehicles can use.
- Distributing these paths when requested by the vehicle.
- Serving as the bridge between the vehicles and the intersection.
- Dealing with any data which is imported or exported.

The paths as previously discussed are stored as lists of points, stored Vector3 objects in Unity. When a vehicle requests a new path the base station randomly generates a number from zero to three which corresponds to one of the four paths. The paths do not include the intersection so base station must add up to two points to navigate the vehicle through the intersection. The base station chooses these points based on which path the vehicle was previously on and which path it has been randomly selected to take next.

Exporting and importing of data is done through the use of JSON files. Our primary reason to use JSON, beyond testing, was to interface the real world communication system with the simulation. The reasoning for this will be discussed in section 2.4.

The intersection class is meant to represent the intersection itself as a set of resources to be shared by each of the vehicles. It is a member of the base station class but its methods are called from each vehicle in their update method. The intersection class keeps track of four string variables, one for each of the four nodes that make up the intersection, these variables can be assigned to a vehicles name to reserve or block its corresponding node for that vehicle. Once a node has been blocked, it cannot be blocked by any other vehicle, and it can only be reset back to an open state by the vehicle that blocked it. The intersection class also has a method that returns false if a given node is blocked, or true if it is open or owned by the vehicle calling the method. When a vehicle approaches the intersection, it checks to see if the next node in the intersection it is going to is open; if the node is open then the vehicle blocks it and continues until it reaches that node and repeats. If the node was blocked by a different vehicle, the first vehicle must stop and wait until that node becomes open, only then is it allowed to block that

node and continue moving along its path. A node is opened when a vehicle reaches the next node after in its path.

The methodology behind this solution is similar to the ones used in resource sharing problems in operating systems, each of the four nodes or quadrants of the intersection can be likened to resources that need to be shared between the vehicles. Certain concepts that apply to resource sharing algorithms would also apply here, such as giving certain vehicles priority based on how long they have waited. This method could potentially be extended beyond intersections, such as coordinating multi-lane freeway traffic.

2.3.4 Collision detection and avoidance

Avoiding collisions is one of the fundamental requirements for the simulation. The intersection collision avoidance is handled differently than the rest of the path because it is being optimized for speed. We tested two methods: rigid bodies with colliders and ray-casting. These methods deal with collisions when outside of the intersections, all of them using specific tools provided by Unity. After testing both methods we decided to use use ray-casting for collision avoidance.

Rigid bodies were tested because they are the preferred method of collision detection in Unity. Rigid bodies are attached to game objects and allow the physics engine to properly calculate the motion of the object due to gravity and colliding with other objects. Rigid bodies have many useful properties like drag, mass and centre of mass which can define its behavior. However, rigid bodies by themselves are not useful because we are trying to avoid collisions and not accurately simulate the effects of a collision. What we wanted to use are colliders, which are a geometric shape such as a sphere, box, capsule or custom mesh; colliders must be attached to rigid bodies to function. The colliders had the useful property of triggering an event when it detected another collider intersecting it. To actually detect collisions before they happen all we would need to do is stretch the collider geometry so that it is actually larger than the vehicles. What this means is that the collision event will be triggered slightly before a collision would occur, giving the vehicle time to stop.

The problem with this method is that the event will trigger for both vehicles and they have no way of knowing whether they are the vehicle that should stop or keep going. To explain this further, if one vehicle is traveling behind the other and is going slightly faster, it will eventually

hit the back of the first vehicle. Slightly before this happens the colliders will trigger events for both vehicles. Ideally, the vehicle in front keeps going while the vehicle behind stops or slows down. However, all the vehicles know is that they were about to collide with something and therefore do not have enough information to correctly respond.

To try and solve this issue we looked into the implementation of two colliders on the same vehicle. One collider would stretch slightly in front of the vehicle while the other would extend to the back and sides. Based on our research it seemed like it might be possible to disable triggering an event for a collider and we attempted to do this with the back and side collider. To explain the intended behaviour we will again use the example from before with one vehicle traveling behind another and moving faster. When the second vehicles colliders intersect with the colliders of the vehicle in front we wanted only the second vehicles collision event to trigger. The goal was to disable event triggering for the back/side collider while still being detectable by the front collider. In our testing, however, we were unable to get this working and had to explore alternative methods.

The alternative we looked at was ray casting, a useful tool available in all graphics engines. The way ray casting works is by defining a start point, a vector direction and a length. The engine will then create a ray, basically a line, with those properties and can check whether it collided with anything. There are other useful features, such as determining what the ray has collided with, but for our purposes it did not matter. Since the vehicles turn on the spot and otherwise only travel straight we decided that simply ray casting in front of the vehicle should be sufficient. Interestingly, this would also be analogous to our original collision detection design for the physical demonstration. Since the physical demonstration originally had ultrasonic sensors to detect objects ahead, it behaved like ray casting.

The implementation of ray casting is very simple; provide a start point at the front of the vehicle, a vector from the current position to the next target point and then provide a length. When the ray cast detects a collision it simply stops the vehicle and continues to check, once the casts come back false the vehicle will continue to move. Again it is important to note that ray casting is not used at intersections because the vehicles must be free to move much closer together. At the intersections a specialized algorithm makes sure the vehicles will not collide; using raycasts would generate a lot of unwanted stopping.

2.4 Transition to controller

The simulation was first intended to function completely separate from the communication system. The original design was to use the simulation to develop the algorithms for managing the traffic and then port those over to whatever interfaced best with the communication system, likely Python. This allowed us to work on all the modules of our project concurrently with the trade-off that there would be some time taken up for converting the code.

As the project developed we further investigated what would be the easiest way of integrating the routing algorithms with the communication system. There were two options that we considered: convert the code to python figure out a method of communicating with the Bluetooth code or keep the algorithms running on Unity and interface that. We decided to choose the second option based on the amount of time invested in Unity and the fact that both would require equal effort to interface with the communication system.

2.4.1 Design Changes

When interfacing Unity and the communication system there are three potential avenues to explore. The first is to set up a form of interprocess communication, using pipes. This method was decided to be non-viable for us because C# does not have any support for pipes and we would be creating a difficult problem for ourselves.

The second optionis to set up Bluetooth in Unity and use that to send all the necessary messages. The advantage of this method is that it involves the least amount of work for the communication system and it would be an active connection. An active connection is a great benefit because it means we can avoid any concurrency issues, loss of data and scheduling problems. The downside of this method is Unity does not have great support for Bluetooth. It uses an open source implementation of C# called Mono and is missing the DataReceived and BytesToRead property. There are also few examples of Bluetooth being used with Unity, which further suggests Unity does not support Bluetooth. Also, setting up a Bluetooth connection would require the communication system and simulation to run on different computers. This would not prevent implementation of this solution but would make testing more difficult outside of group work sessions.

The final method was to use files which would be written to and read from by the simulation and communication system. We would be using JSON files because these are easier to parse than regular text files and are supported by most languages, including python and C#. This solution simplifies interfacing because reading and writing JSON files is easy and also abstracts any communication done. The downside of this solution is that it will not be an active connection and therefore we must deal with the issues that an active connection does not. We decided on the third option, reading and writing to shared files. With our choice of the third option, there were four things we needed to figure out:

- What information needs to be sent?
- How often do we need to check the files?
- How do we avoid data loss/overwriting?
- How do we avoid both programs trying to use the same file?

From the simulation to the communication system we determined we would need to send the vehicle name and the string path. From the communication system to the simulation we pass along the updates the real world vehicles create. This includes its current action, the queue of commands it has and some other information the simulation does not use.

To handle the issue of both programs trying to write to the same file we decided to implement multiple JSON files, two for each vehicle. One JSON file is for the communication system to write to and the simulation to read from. The other JSON file is for the simulation to write to and the communication system to read from. This means the two programs will not overwrite each others data and also will not have file write collisions.

The next issue to solve was determining when the files have been updated and there are two potential solutions for that. The first is to check the file metadata and the second is to add an extra piece of the data exported to the JSON file which increments each time the file is updated. We opted for the second solution because it is substantially easier and achieves the same net result. Whenever the simulation writes to the JSON file it increments an integer stored in the JSON file, the communication system stores an integer which it compares against to determine whether there is a new update.

The final issue is how often the programs need to check the JSON files. The simulation checks whenever the simulated vehicle had reached a node, compares to the updates and adjusts based on that information. The simulation only writes to the JSON file when performing collision detection and giving a new path. Giving a new path is an easy case because that is infrequent. The real limiting factor is collision detection and finding a balance between checking often enough to avoid collisions and not blocking the communication system from performing its other functions. In our testing we found with our vehicle speed and the latency in communication, collision detection will still work.

2.4.2 Intersection Routing Results

The simulation currently supports two different algorithms for routing at the intersection: stop sign and automated. To compare the two algorithms a tests were performed to compare the average time spent waiting by the vehicles. Each test was performed with five vehicles, for one minute and the waiting time was averaged across the vehicles.

	Stop Sign	**Automated**
Trial 1	28.96 s	9.46 s
Trial 2	22.50 s	3.32 s
Trial 3	30.88 s	6.51 s
Trial 4	25.16 s	4.15 s
Trial 5	23.65 s	5.35 s
Average	26.23 s	5.76 s

Table 2.1: Car wait times

From these results we can see the automated intersection outperforms the stop sign. The stop sign does not perform exactly like the real world version because it only allows one vehicle through at a time; in the real-world, drivers will make decisions about when it is free to go through. However, the large difference in time still reflects positively on the efficiency of the automated intersection.

Chapter 3

Communication System

The objective of the communication module is to relay information between the simulation and the vehicles. The base station, which is hosted on a laptop, runs software to send commands and receive the status from each individual vehicle.

The base station coordinates the path of each vehicle as well as the order they cross the intersection. The base station also manages the synchronization between the simulation and real vehicle movement.

This section discusses requirements and constraint for the communication system and how the design decisions were made.

3.1 Requirements and Constraints

This projects specific requirements and constraints for the communication system are as follows:

- The communication module must be able to support more than two vehicles.
- The module receives and sends data to the base station concurrently. We must avoid concurrent elements overwriting each others data.
- The communication must have a stable data transmission rate within 10 metres. The stability can be
- determined by the round trip transmission time between a vehicle and the base station.

The average time requirement we must meet is around one second as this is the expected reaction time needed for a driver to stop the vehicle [1].

- The communication must be secure, it needs a mechanism to block an anonymous connection. Only the vehicles known to the base station are allowed to connect. This minimizes the interference of the communication.
- The budget to invest the communication system is under $200 and this is applied to five different vehicles with one laptop.

3.2 Design Decisions

We chose Bluetooth as our primary communication system and Wi-Fi as a support due to their stability, simplicity, low cost and how well-developed they are. The microcontroller on each vehicle carries the Wi-Fi/Bluetooth module that handles all of the Wi-Fi/Bluetooth signals. We did not set up everything with Wi-Fi because of university regulations. Bluetooth fits our project scale and is easier to manage compared to Wi-Fi.

We used Python3 to write the software that implements the communication module because we found a robust, open source library that supports Bluetooth communication. All the data is transferred in the JSON format a lightweight format for exchanging data, which easy to read and write [4].

We came up with a list of possible protocols to use: Zigbee, IEEE 802.11p(WAVE), Bluetooth and Wi-Fi. Zigbee is a close-range wireless communication [2] that can support our project. By utilizing Zigbee, each vehicle can communicate with each other, in what is called a mesh network, rather than one server with multiple clients. IEEE 802.11p (WAVE wireless access in vehicular environment) is a standard used for future inter-vehicular communication. The spectrum of WAVE is in the 5.0 GHz range. The advantage of using WAVE is the ability to prioritize commands in a priority queue, however “dense and high load scenarios the throughput[sic] is decreases while the delay is increasing significantly”[3]. We did not use Zigbee and IEEE 802.11p due to their complexity and we do not have the resources or knowledge to implement them. Bluetooth frequencies is located in the 2.4 GHz band and Wi-Fi currently supports dual-band with both 2.4 GHz and 5.0 GHz. Both Bluetooth and Wi-Fi are well known and there are many libraries that support both protocols that work well with all kinds of programming

language.

3.3 Implementation

This section discusses how the software and hardware were implemented, and the design decisions we made.

3.3.1 Software Architecture

All the code for the communication module is written in Python3 - an interpreted high-level programming language for general-purpose programming. We used the Visual Studio 2017 Integrated development environment application as the programming tool. We also used an open-source library called PyBluez to handle all of the Bluetooth connections. All the software programs and libraries are installed on Windows OS.

We modularized all the components so they can work without relying on information from the others. The communication system is made of three different components: client, server, and data holder.

The client component is the communication module on each vehicle, which receives each command from the vehicle and puts them into the action queue (section 4.2.1.2). The server component is the communication done by the laptop (the communication server). This server listens for incoming connection requests from vehicles. Once the request is accepted, the server assigns a thread to handle all the communication with that vehicle, and can support up to four different vehicles simultaneously. The last component holds the commands and status information of each vehicle. We put the data holder into a specific folder that the simulation can get the data from.

3.3.2 Hardware

Each vehicle is equipped with Pine64 WIFI 802.11 BGN/Bluetooth 4.0 Module which allows the vehicle to gain access to other devices through Wi-Fi and Bluetooth. This module has throughput of up to 150 Mbps in theory and supports Bluetooth version 4.0.

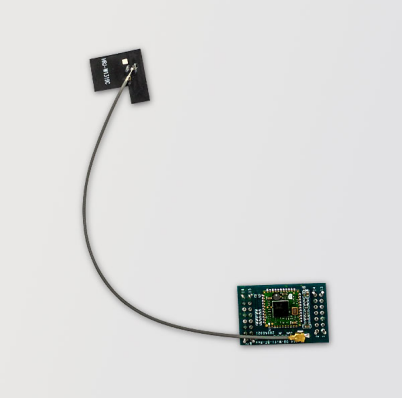

Figure 3.1: Pine64 Wi-Fi/Bluetooth Module.[5]

We performed an experiment comparing an embedded Bluetooth module on a laptop versus a USB Bluetooth dongle in order to compare their performance. The embedded Bluetooth only manages to connect to two vehicles, the USB Bluetooth dongle gives better result with four vehicles supported at once. Thus, we chose to use the Bluetooth dongle. The Bluetooth dongle uses the CSR8510 chip, with dual-mode transmission enabled. The manufacturer of the USB dongle claims the transmission rate can go up to three Mbps in range of 10-20 m which meet our expectation. Our USB Bluetooth dongle supports our laptop, acting as the main server to coordinate all the vehicles. Figure 3.2 shows what the USB dongle and chip looks like, and some of its specifications.

Figure 3.2: USB Bluetooth 4.0 module [6]

3.3.3 Setup

Connecting multiple vehicles and communicating with them simultaneously is achieved by the multithreaded function on the server. Due to the limitations of Bluetooth communication on a single server, up to four vehicles can be supported with a good connection. This number meets our requirement since we initially planned up to four vehicle.

Each vehicle will have its own thread - a section of code that runs concurrently to the rest of the vehicles code - dedicated to communication. The server component has pre-defined MAC addresses for each vehicle connect to each client directly without handshaking. This reduces the time to connect for each vehicle and prohibits anonymous connections from accessing the network. In a scaled up implementation, handshaking would be necessary.

A wireless router was set up to remotely connect to each vehicle to see its performance. The technique we used to remotely monitor each vehicle is called SSH Secure Socket Shell- which is a network protocol that provides the user a secure way to gain access to other devices that have an operating system. Figure 3.3 shows an example of SSH being used.

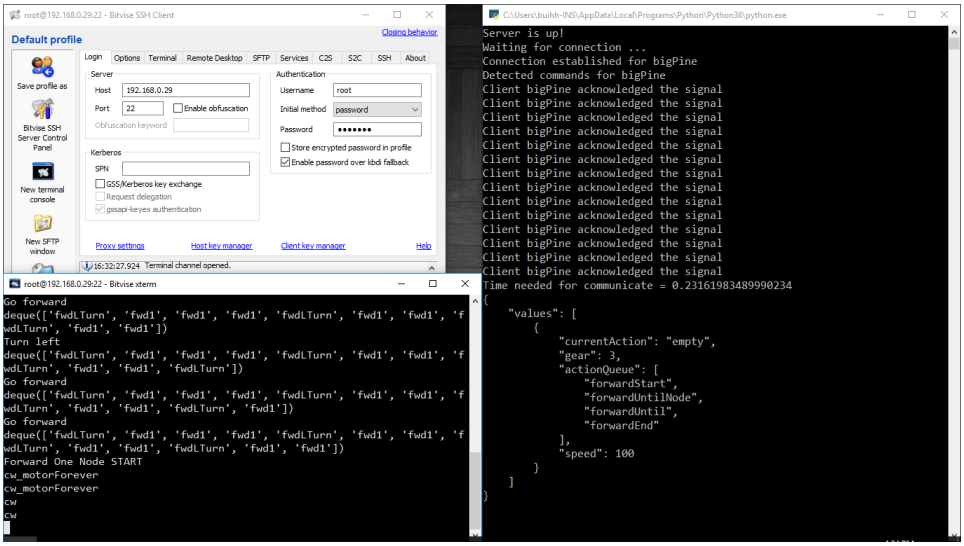

Figure 3.3: SSH Interface

3.3.4 Operation

The simulation generates a JSON file containing all the necessary commands. These commands are: FORWARD, DOWN, LEFT, RIGHT, STOP, UPDATE. Each JSON file ends with at least one UPDATE command in order to get the status from the vehicles, also to ensure that all the commands are successfully transmitted. The server component on the same laptop takes the JSON file as input. Each JSON file is named with the vehicles name tag so the server is able to send to corresponding vehicles.

The client component on each vehicle receives the commands, translates them into tasks and puts them into the action queue(section 4.2.1.2). If the vehicle detects the UPDATE command then instead of queuing an action, the vehicle prepares a JSON file which contains the status of the vehicle to send back to the server. The status includes gear, speed, current action and the action queue of the vehicle. The status information is used by the simulation on the laptop to synchronize with the physical vehicles. If the vehicle detects commands other than UPDATE, the acknowledge signal (ACK) is sent back to the server component on the laptop.

Finally, if the server component receives an ACK signal then either another new command is sent to the vehicle or the system waits for another input; otherwise when the communication receives the status of the vehicle, the system produces a new status file in JSON format. This JSON file is read by the simulation to update itself.

There needs to be a way for the simulation to tell if the JSON file has an updated status of the vehicle. We resolved this issue by introducing a new variable called ID in each JSON file. Both the simulation and communication system check this variable when the UPDATE signal is received. If the same ID is read, then no changes are needed, otherwise, the simulation updates the vehicles status.

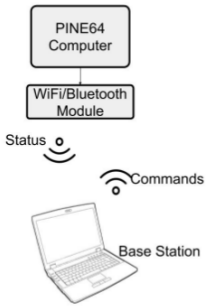

Figure 3.4: Communication system model

3.4 Achievements

The vehicles are controlled through commands from the server and the server receives updated statuses from up to four different vehicles simultaneously through Bluetooth. This exceeds our initial goal of supporting up to two vehicles.

Throughout all the testing there is no package loss, which ensures that the communication is reliable and secure. This is important for traffic of autonomous vehicles, as our project heavily relies on both performance of the motor controls on each vehicle and the quality of the communication system. With a high quality communication, the delay time for each signal transmitted and received is decreased, thus giving a better response time. Figure 3.5 illustrates a benchmark for the communication system.

```
Connection established for Pineapple
Start ping the client
Successfully ping Pineapple in 0.06916618347167969
Successfully ping Pen in 1.223296880722046
Finish ping program. Number of successful transfer:5 out of 5
Successfully ping Pineapple in 0.5160112380981445
Successfully ping Hoodie in 1.232456922531128
Successfully ping Pineapple in 0.10031867027282715
Successfully ping Hoodie in 1.2553420066833496
Successfully ping Pineapple in 0.3162400722503662
Finish ping program. Number of successful transfer:5 out of 5
Successfully ping Pineapple in 1.7346811294555664
Finish ping program. Number of successful transfer:5 out of 5
Connection established for Apple
Start ping the client
Successfully ping Apple in 0.4854903221130371
Successfully ping Apple in 0.0
Successfully ping Apple in 0.08459305763244629
Successfully ping Apple in 0.13741087913513184
Successfully ping Apple in 0.0312502384185791
Finish ping program. Number of successful transfer:5 out of 5
Connection established for Pen
Start ping the client
Successfully ping Pen in 0.3380465507507324
Successfully ping Pen in 0.4475088119506836
Successfully ping Pen in 0.08455848693847656
Connection established for Hoodie
Start ping the client
Successfully ping Hoodie in 0.20064616203308105
Successfully ping Pen in 1.1692962646484375
Connection established for Pineapple
Start ping the client
Successfully ping Pen in 0.864016056060791
Successfully ping Hoodie in 2.434894561767578
Successfully ping Pineapple in 0.785761833190918
Finish ping program. Number of successful transfer:5 out of 5
Successfully ping Pineapple in 0.0605187416076660016
Successfully ping Hoodie in 1.4384822845458984
Connection established for Apple
Start ping the client
Successfully ping Pineapple in 0.2523534297943115
Successfully ping Apple in 0.5160098075866699
Successfully ping Hoodie in 0.4326941967010498
Successfully ping Apple in 0.28446197509765625
Successfully ping Pineapple in 1.154466152191162
Successfully ping Hoodie in 1.2392570972442627
Successfully ping Apple in 0.6385014057159424
Finish ping program. Number of successful transfer:5 out of 5
Successfully ping Apple in 0.48891377449035645
Successfully ping Pineapple in 2.123610734939575
Finish ping program. Number of successful transfer:5 out of 5
Successfully ping Apple in 1.249720811843872
Finish ping program. Number of successful transfer:5 out of 5
```

Figure 3.5: Communication quality testing

Chapter 4

Vehicle

An autonomous vehicle is simply a vehicle capable of navigating without any human input. A vehicle needs to be able to stay in its lane, obey the rules of the road, conduct turns consistently and slow or stop whenever it detects an obstacle. Our objective in this project was to build an array of vehicles that could mirror the functionality of an autonomous vehicle in order to test and demonstrate our coordination technology.

We designed vehicles that were able to:

- Perform basic directional movement.
- Sense their environment and react accordingly.
- Connect to our base station to receive commands and send its status.

The vehicles are equipped with two DC motors allowing our vehicles to have two wheel drive functionality. IR sensors are on each vehicle to detect markers placed on the track so that the vehicles have a reference for where they were. We use the data generated from these IR sensors to build an error correction system to adjust the vehicles if they veer off their lane. A portable powering system ensures vehicles could stay online and perform a lengthy demonstration. A 3D model chassis houses all the components onto them and to provides a consistent structure across all these vehicles. As the Capstone program provided us with a set budget, the vehicles are designed to be as effective as possible while maintaining the budget constraints. We constructed five vehicles in total for the system while staying within the scope of the budget.

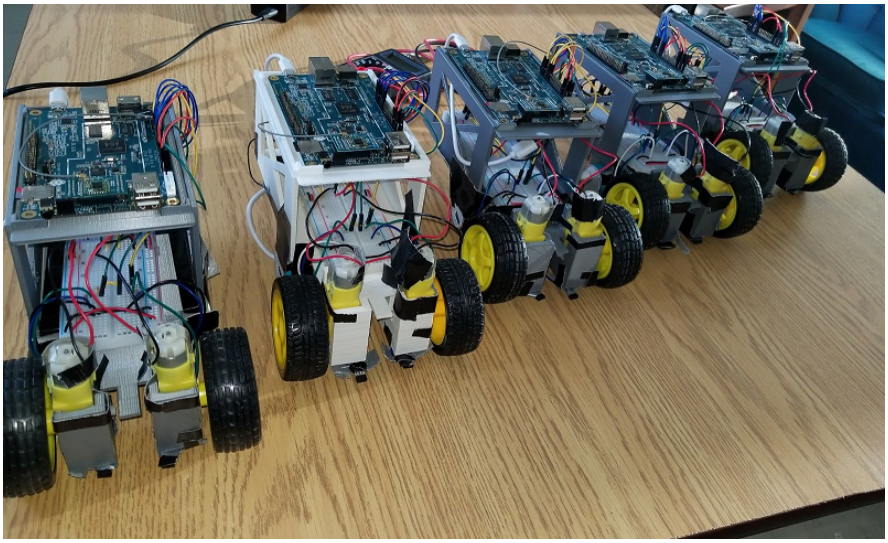

Figure 4.1: The miniature autonomous vehicles

To process and actuate all the commands the vehicle needs to perform, we used the PINE64 single board computer. This reduces the difficulty of designing and testing software and made communicating with the base station straightforward. The vehicles receive commands (such as forward, right turn, etc.) from the base station and execute them in sequence. The base station also receives the status of the vehicles in order to efficiently coordinate them. To achieve that effect, the software architecture receives high-level abstract commands and processes them down to the low-level instructions needed to control the peripheral components of the vehicle such as the motors and sensors. The vehicles software constantly scans and reacts to its environment so that the vehicles can take appropriate actions in order to stay on track, know how far it has to travel and when its appropriate to perform certain actions.

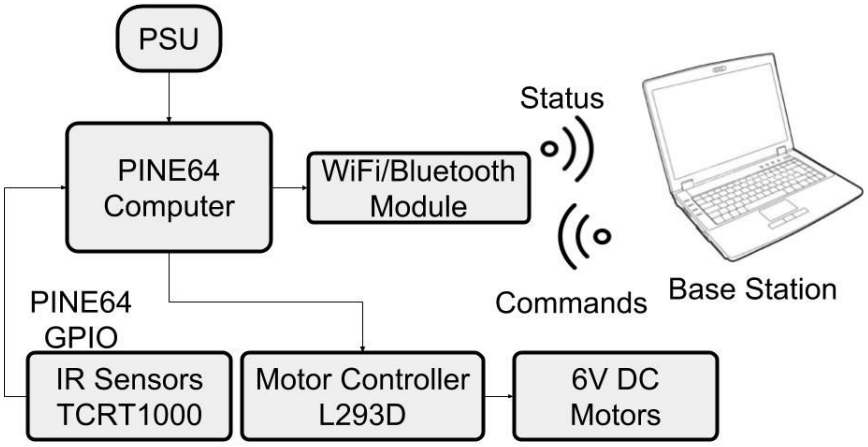

Figure 4.2: High-level block diagram of the vehicle components

4.1 Hardware

The following sections describe all the hardware components of the vehicle, their specifications and how we interfaced with them.

4.1.1 Pine64 (Processor Boards)

We use the PINE64, 64-bit single board computer, for our vehicles . This computer has an embedded GPIO pin we interface with and a Bluetooth module [5]. Bluetooth technology wis well suited for our design as Bluetooth is already well established and there are many resources to build an interface with it. Before settling on the PINE64 for our design, we considered a few different options for a main processor/controller. The foundation of a functional autonomous vehicle is its processing unit. We required an efficient method of providing inputs to the motor drivers and receive inputs from the sensors. Another crucial feature is wireless connectivity, whatever processing unit we chose for our design needed a wireless solution available.

Our first consideration was using a PIC microcontroller and a ZigBee module from the

same manufacturer. The PIC is a family of microcontrollers by Microchip Technology, the same manufacturer also had a peripheral ZigBee module for these line of processors. ZigBee is an IEEE Communication protocol for use in small ranges and Wireless Personal Access Networks [2].Our idea was to take this microcontroller IC and wireless module combination and design a PCB around it. We would have designed a PCB that all of our components (such as the motor control IC and sensors in addition to the microcontroller) were fitted on. This would have been efficient as all the components would be on one custom board and would be specific to our application. We decided not to go with this as the process would have taken more time than we had for this project. Also, it would have been very difficult to program and update software onto the microcontroller. One other concern with this was that the ZigBee module itself was too difficult to interface with. There were not enough resources for this technology when compared to other technologies and it would have been a large task to build an entire ZigBee server.

We had also considered using Arduino(s) in conjunction with a Bluetooth shield to drive our vehicles. Arduino is a line of microcontroller boards [7] that can have extended functionality (such as Bluetooth or GPS) by installing shields onto them. We would have designed all our peripheral devices to interface with the Arduino board and have the Bluetooth shield communicate with the base station. We decided not to go with this option as we found it to be too costly for our design. Our design required us to build multiple vehicles within the constraint of our budget. An Arduino Uno and Bluetooth shield alone costs around $70 CAD. This was too much for just the processing alone as we had to factor in the costs for all the components as well.

Another candidate was the Raspberry Pi, a single board computer that had GPIO pins we could interface with. This option suited our requirements and constraints, but we then found a much more cost efficient alternative, the PINE64. The PINE64 single board computer is a competitor to the Raspberry Pi, which dominates the low-cost single board computer market. As mentioned before, the PINE64 met all the right interfacing requirements and was much more cost efficient than its competitors (The board itself only cost $15 USD). Using this single board computer also allowed us to develop on a much more familiar platform (Linux) and language (Python).

Technology	Cost	Pros	Cons
PIC MCU + ZigBee module [8]	$25 CAD	-Low-cost -Dedicated controller -Meant for small ranges	-Requires PCB design -Proprietary software -Few resources
Arduino BT shield [7]	$70 CAD	-Dedicated controller -Great documentation	-High cost
Pine64 [11]	$29.87 CAD	-Low-cost -Runs Linux OS -GPIO interface -Many powering options	-No dedicated controller -Few permissions

Table 4.1: Processor options

As this project involved producing multiple vehicles, we developed and set up one prototype board to get a sense of the process of setting up these boards for the vehicles. The first step to setting up the board is installing an OS. For this project we use Armbian, an offshoot of the Linux distribution Debian, we chose this OS as it was the most compatible one for this board. This distribution has all the basic functions of Debian, except it is made to work on ARM processors, which the PINE64 uses. We then had to install all the necessary software and libraries needed for this project. The programming language used for this project is Python, as there are a lot of libraries supported for the different interfaces. We installed the GPIO library for this specific board. This library allows us to control the GPIO pins on the board to provide outputs and inputs for various peripheral devices. We also installed asyncio, a python library for scheduling events and operation, which is crucial to the command system. In order to design proper control and communication software we tested all the libraries and components separately before putting them together into one comprehensive system.

A major benefit to developing on a single board computer using a Unix-like OS is how supported it is for different technologies and protocols. When developing software on the boards, we connect to the computers remotely via SSH. This means we can wirelessly connect to the

boards to upload and edit software easily. Another benefit is the fact that all the storage of the device is on an SD card. When we finalized all the designs for the software, we cloned an image of the prototype onto other SD cards to port all the necessary software and libraries onto all the vehicles in our system. The PINE64 board is incredibly versatile in its functionality. It is compatible with the motors and sensors we implemented in our design, has many different powering options that allows us to choose a stable power supply for our project and provides a stable foundation for our multi-vehicle traffic network.

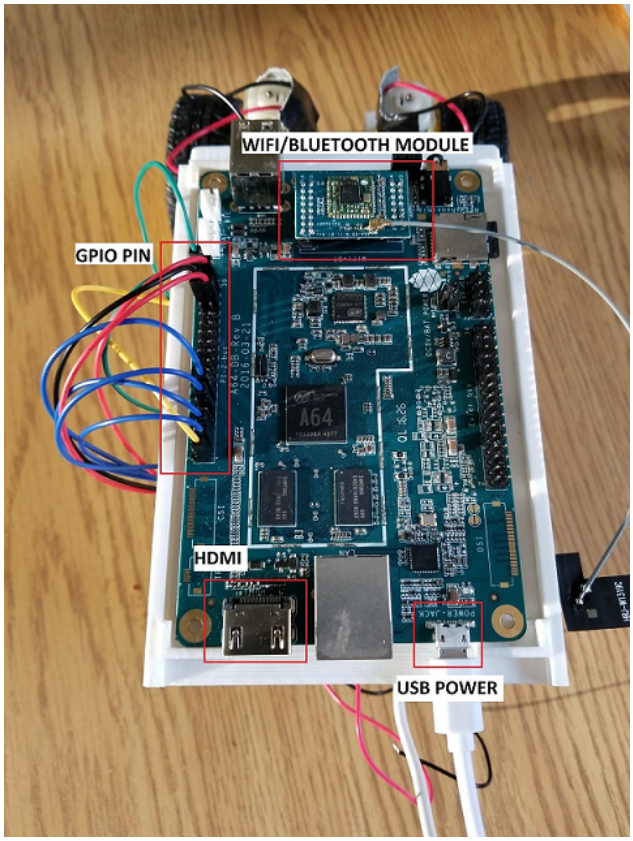

Figure 4.3: Top down diagram of the PINE64 and the interfaces used

4.1.2 Motor & Motor Control IC

The vehicles are driven are by a simple two-wheel drive system. We can control two wheels and can dictate if a wheel is moving clockwise or counter-clockwise. We utilize an H-Bridge circuit with simple DC motors in our design. The DC motors we use are simple 6V motors. If pow-

ered from their positive terminal, they rotate their gear clockwise, if powered from the opposite terminal they rotate counter-clockwise. An H-Bridge circuit is a simple circuit that switches which direction the power is going into the motors. We use a dual H-Bridge IC, the L293D, in our design. We connected the motors to the L293D, and connected the L293D to the PINE64 to programmatically control the logic of the motors. The wheels are attached to the gears of the motor to drive our vehicle. To control the speed of the motors, the L283D has an enable pin for each motor. When the pin is on, it allows power to flow into the DC motor. We use Pulse Width Modulation, or PWM to dictate a duty cycle for the motors to be powered on. Using the GPIO pins of the PINE64, we provide a PWM signal to the L293D so that we can control the speed of each motor.

I1	I2	I3	I4	Left Motor	Right Motor	Vehicle Direction
0	0	0	I4	Stop	Stop	Stopped
0	1	0	1	Clockwise	Clockwiser	Forward
1	0	1	0	Counter-Clockwise	Counter-Clockwise	Backward
1	0	0	1	Counter-Clockwise	Clockwise	Turn Left
0	1	1	0	Clockwise	Counter-Clockwise	Turn Right

Table 4.2: Inputs to the L293D H-Bridge circuit and its effect on the motors

Figure 4.4: One of the DC motors used.

4.1.3 Ultrasonic Sensors

One functionality our system needed was that ability to stop or the vehicle if it was too close to another vehicle. We wanted a sensor that was capable of inputting the range of whatever was in front of the vehicle to the PINE64. Our initial solution was to install an Ultrasonic sensor on our vehicles. This sensor sends an ultrasonic pulse and calculates the time it takes for the pulse to come back to estimate the range. This sensor ended up being removed from our final vehicle design, we still wanted to give insight into what the problem was with this sensor and the compromise we made to still maintain the desired functionality.

The HC-SR04 Ultrasonic Sensor is a premade board with an Ultrasonic Transceiver and Receiver installed [9]. It has one input and one output, the trigger and the echo pin. If the trigger pin is activated, the transceiver sends out an ultrasonic pulse. The echo pin gives a signal when the receiver detects a pulse. How we set this up with the PINE64 is by activating the trigger

when we wanted to sense the range. We would then wait for the echo to receive something, and use the time difference to calculate an integer value of the range.

After testing this device with various setups, we concluded that we had to remove this feature from the final revision of the vehicle. As this was a timing based range sensor, this was a problem from the programming standpoint, We had to generate a thread for this sensor alone to constantly poll for the range. In addition to that, the timing functionality was not very compatible with this computer board, as this was a computer and not a dedicated microcontroller like the Arduino, the timing would occasionally be inaccurate as the computer would sometimes be doing a background process like garbage collection, taking away some of the timing. The most troublesome problem was that the sensor was not calibrated for small distances. We needed to sense for a small range of 10-20 cm, but this sensor is more meant for ranges larger than 50 cm.

In the end, we found we could achieve the functionality of knowing if the vehicle was too close to another vehicle by using the base station. The base station is capable of localizing all the vehicles in our system, so we were able to accurately estimate if a vehicle was getting too close to another vehicle, and send a command to that vehicle to tell it to stop.

4.1.4 IR Sensors

Every vehicle in our system requires:

- The ability to know where it is on the road.
- Where and when to stop or turn.
- The ability to generate and act on feedback if it is veering too far off the road.

Our solution to generate these inputs into the vehicle is to install an array of TCRT1000 IR sensors onto them. These sensors detect for nodes and margins we placed on the track. The vehicles relay to the base station when the they pass by these nodes so that the base station can localize all the vehicles in our network. We laid lines on the road for the vehicle to detect and utilize in an error correction system. Both of these features will be further explained in a later section.

The TCRT1000 sensors function by emitting an infrared light and detecting a reflection of this light. The sensors have two parts, an Infrared LED, and a phototransistor that detects for Infrared light [10]. When the phototransistor detects an IR light it enables the switch, sending that true signal to the PINE64. The IR light the emitter sends is reflected off a surface close enough to the sensor to the phototransistor. We used a black background for our track and used white for all the nodes and margins. The black surface does not reflect the IR light while the white surface does. This allows for precise control of the vehicles.

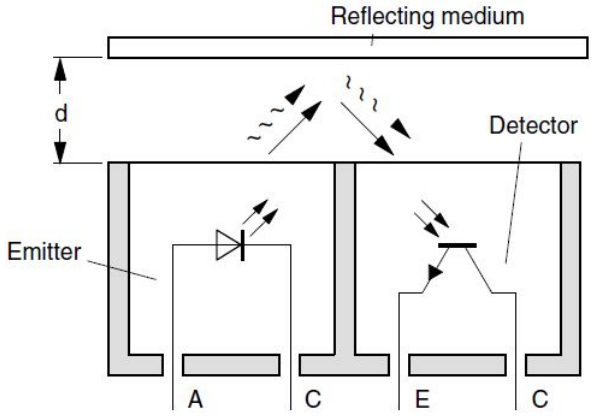

Figure 4.5: Internal diagram of TCRT1000. It contains an IR emitter and a phototransistor

We implement two of these sensors. The phototransistor is connected to the GPIO input of the PINE64 while the infrared LED was connected to the 3V supply of the PINE. The transistor has a 39 kΩ resistor connected to its collector so the switch is activated for a distance of 1 cm. We connect a 100 Ω resistor to the emitter so that the IR light is at an appropriate brightness.

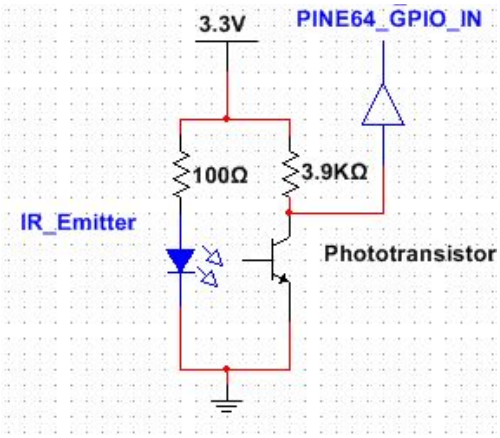

Figure 4.6: Configuration of a single sensor and how it interfaces with the PINE64

4.1.5 Power Bank

Our design requires that our entire system run for at most 30 minutes. There are three interfaces for powering the PINE64: a DC IN pin that takes in 5V 2A of DC power, a battery slot designated for a 3.7V Lithium Ion battery, and a standard USB plug meant for connecting the wall charger into.

The power supply we chose is a 5V 2A USB power bank that powers the device and its radio module and lasts a full two and a half hours without any processes running. The power bank runs our motor control software for longer than 30 minutes. However, we found that the computer crashed in certain circumstances with this setup. When we powered the motor control IC, Wi-Fi, and Bluetooth modules on all at the same time, the computer will crash. A separate 6V DC power is dedicated to just the motor controller IC and motors, as the crashes happened because too much power was being pulled from the board.

Our first thought was to build our own power supply for the board. The PINE64 has another set of pins in addition to its GPIO pins the manufacturer defined as the Euler connector. One set of these pins were DC IN. The datasheets of the board stated this pin takes in 5V 2A of DC power to enable the board, so we started designing methods to deliver this power. One of feasible option we had was designing a DC to DC converter to regulate the desired power. A Buck-boost converter is a DC converter that adjusts a DC input to the desired value. We planned on using four 1.5V AA batteries to input to a buck-boost IC and deliver 5V 2A. We found that a

system like this may not have been any more cost-efficient than using an off the shelf solution, so we pursued those options instead.

The PINE64 had a power slot specifically for 3.7V Lithium Ion Batteries. This option was appealing as Li-Ion batteries were cost efficient and rechargeable, and had a desirable form factor for our vehicles. Once we tested the board with a Li-Ion battery, we found out that the board actually entered a low-power mode. Some of the functionality was not the same as when using other methods such as not being able to output 5V, so that would affect some designs we had in mind. When we tried to interface with the Bluetooth radio, the board would occasionally crash. This made the Li-Ion battery option too unreliable for the system we wanted to build.

4.1.6 Chassis

A solid foundation to put all the aforementioned components onto is required to satisfy this requirement we designed and 3D printed chassis. We considered building a chassis by designing stencils and cutting out pieces of wood to serve as the framework of all the components, but this was a difficult task as all the vehicles could have some variation between each. When we proposed our budget, we were introduced to the 3D printing facility our department has. The technical staff told us how to get a 3D model for our chassis set up and printed. Being able to 3D print a chassis out is time efficient we do not have to manufacture every chassis, and all the vehicles are consistent in terms of its structuring.

To build the model we use 3DS Max, a 3D modeling program that one of our group members had prior experience with and also, the student version is free. This program was able to export into the .STL format, the format needed for the 3D printer we used.

In designing the model for the chassis designed, the model should:

- Mount the computer into it.
- House all the circuit boards, power supplies, batteries, etc.
- Hold the motors in place.

The computer mount is a plate on the top of the 3D model with four smalls nubs that the PINE64 computer fits into. For the circuitry there is a lower compartment underneath the com-

puter mount that all the components are placed into. The compartment is big enough to contain all the peripheral components. There is room underneath the compartment to fit the power supply onto it. For mounting of the motors and wheels, two motor holders are made to fit our motors onto. We used a reference model for the motors we are using to get the correct dimensions for it. These are placed in the front of the vehicle so that IR sensors can be placed underneath them for the sensors to detect the track.

For the back wheel of the chassis, we 3D printed out what is called a Ball Caster. This is essentially just a holder for a ball. Using this as a back wheel provides omnidirectional turning. It also makes building the robots a lot easier. We use pre-made models for Ball Casters found online. Ping pong balls are inserted into the ball casters, and it functions as the back wheel of the vehicle.

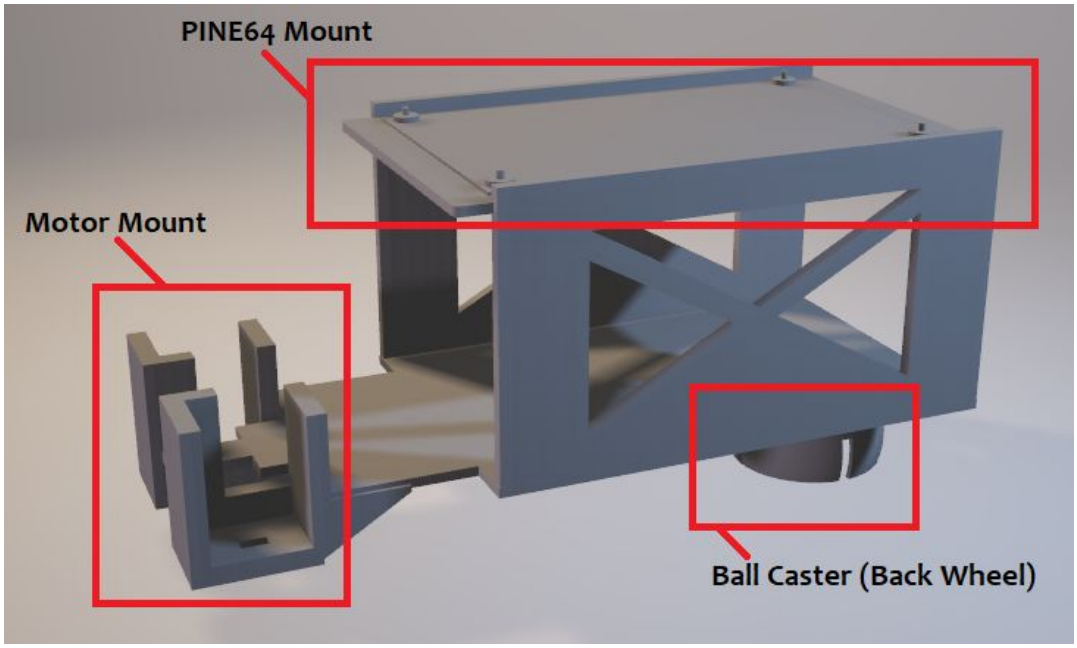

Figure 4.7: 3D model used to print the chassis

4.1.7 Budget per Vehicle

This project involved refining a single design for a vehicle, and then constructing as many vehicles as we could, the budget was constructed on a per vehicle basis. We built five vehicles within the budget.

Table 4.3 contains some of the fundamental components discussed previously. Notably the main computer board, motors, and motor controller. We successfully implemented the Infrared sensors and we also finalized the power supply, which is a USB power bank. We also had to factor in the cost of 3D printing. The tech shop initially allowed us to print the first three chassis complimentary, but as we needed to print more, we had to cover the costs of the printing spool from our budget. The following is the budget for the finalized design of the vehicles.

Name	**Quantity**	**Cost per unit**	**Total cost**
Pine A64 512MB board	1	18.17	18.17
PINE64 WIFI 802.11BGN/BLUETOOTH 4.0 MODULE	1	12.11	12.11
TCRT1000 Infrared Sensor	2	1.68	3.36
L293D H-Bridge Circuit IC	1	0.41	0.41
Plastic Tire Wheel with DC 3-6V Gear Motor	2	1.98	3.96
Poweradd Slim2	1	14.99	14.99
3D Printed Chassis	1	17.00	17.00
SD Card	1	12.99	12.99
Total per robot			82.99

Table 4.3: Budget

4.2 Command Coordination Software

A Linux distribution called Armbian is installed on the PINE64. The code is written in the Python language. The Python programming language is versatile, readable and familiar to our group and the major appeal for this project is the fact that there were many supported libraries to use in our implementation. An open-source library is available specifically for the board we used to interface with the GPIO pins of the PINE64, so we can generate the inputs and outputs needed to send to the appropriate peripheral devices (Motors, Sensors, etc). Another library used is AsyncIO. This library provided infrastructure for writing single-threaded concurrent

code using coroutines [21]. We use its event scheduling system to coordinate and queue outputs to the motors at specific times and schedule when the vehicle should start scanning for nodes. A Bluetooth library called PyBluez is also used to interact with the PINE64s Bluetooth module via Python. Having all these resources available allows us to build a very robust motor controller for our design and implementation.

4.2.1 Software Architecture

The vehicles in our network are treated as a black box from the perspective of the base station. The base station only gives inputs to each vehicle and observe the change in the entire system. The vehicles need to perform three basic functions: Listen for simple commands, execute these commands in sequence, and keep the base station updated on the status of the vehicle.

From the perspective of the base station, it observes inputs and outputs from the vehicles through network, but what happens inside the vehicle is a lot more involved. The vehicles take these high-level commands and distill them into sets of low-level instructions that are sent to the various peripheral devices. To manage all of this and to be efficient we designed the software to be as modular as possible. Figure 4.8 and the following sub-sections detail the general flow from the base station to the vehicle hardware on a module to module basis.

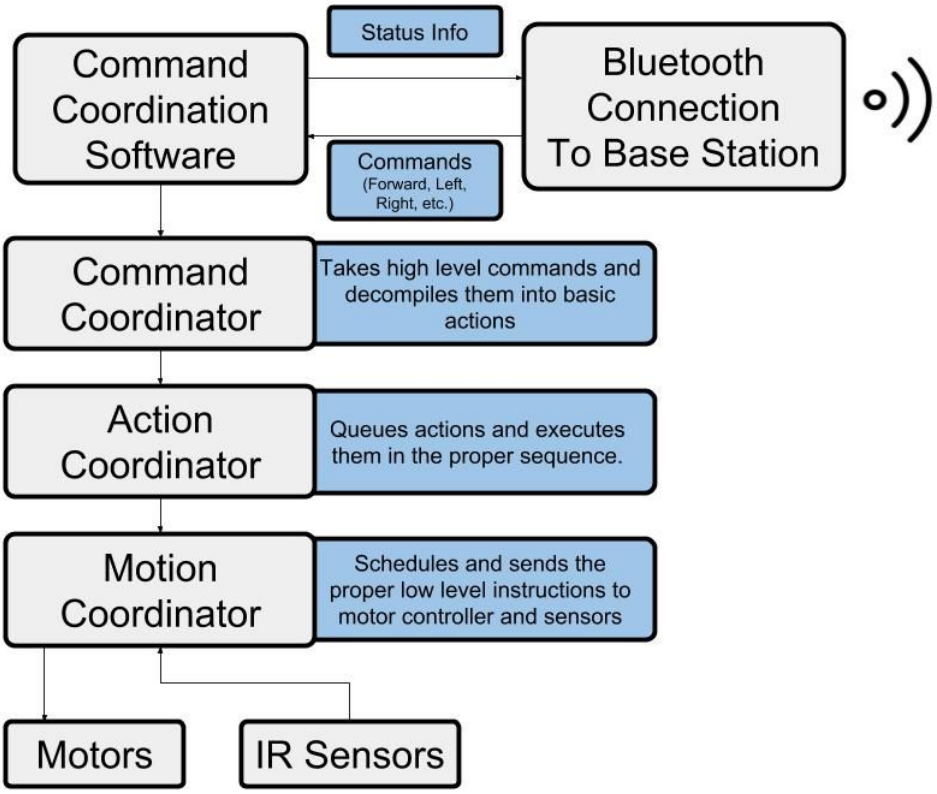

Figure 4.8: Organizational chart of the command coordination system

4.2.1.1 Command Coordinator

The command coordinator is the highest level of the software. When the vehicle receives commands from the base station it is the command coordinators responsibility to interpret that command and compile it into its lower level instructions. The inputs into the command coordinator are basic directions that the base station requires the vehicle to perform. Analogous to this are directions from Google Maps. If directions are loaded from a program like Google Maps, a driver is given a list of roads to take and streets to turn on. What actuates a real-life vehicle is the human driver, interpreting directions like this to change gears, adjust the steering wheel and accelerate. Similarly, the vehicles in our system receive basic directions like Forward or Right Turn from the base station and place them in a command queue. The command queue is the main interface between the vehicle and the base station. The vehicle receives a string messages from the base station to queue up commands. The vehicle then iterates through the queue to decompile those high-level instructions into their baser parts. These lower level instructions are

sent to the action coordinator, whose job is to decide the appropriate timings of these actions and execute them in sequence.

An example of how a simple Move forward and turn right command is issued from the base station to the vehicle is as follows. The base station sends the commands Forward and then Right to the specific vehicle. The vehicle receives the commands and places them in the command queue. The vehicle reads the Forward Command. It decompiles that command into its basic actions: Move forward until you exit the current node, move forward, and scan for the next node. The command coordinator sends those actions to the action coordinator. It then does the same thing with the Right command, telling the action coordinator that the vehicle will go forward until the vehicle hits a turning junction and then to turn until it detects the turn is complete through the IR sensors.

4.2.1.2 Action Coordinator

The Action Coordinator is a simple mediary between the Command Coordinator, which receives and interprets commands, and the Motion Coordinator, which schedules inputs to the motors and sensors. While this class may seem straightforward, it is very crucial for the functionality of our vehicles. The vehicles require basic queueing functionality for each action it takes, so this coordinator is responsible for maintaining that order.

An action is an instruction or step a vehicle takes. In a real-life vehicle, an example of one action is staying straight in a lane until you reach a stop sign or have to turn. This is similar to our vehicle, except we have to account for the nodes we place on the road. So we programmed actions for action queue such as drive straight until a node is reached, drive until a node is exited, drive straight for a certain duration, or turn right on a node. The main feature of this class is to designed to send the actions to the Motion Coordinator in the correct sequence. We needed the functionality of being able to queue up actions to the vehicles so the Action Coordinator is responsible for maintaining this action queue. The coordinator maintains what action is currently in progress and waits for that action to finish before executing the next action in the queue.

4.2.1.3 Motion/Motor Coordinator

Motion Coordinator is the lowest level of the software. The Action Coordinator determines when and what action needs to be executed and sends it to the Motion Coordinator for it to determine what directions the motors should go and to start polling the sensors for information. This class is the main interface between the PINE64 and the peripheral devices. The class initializes instances of the left and right motors and sensors so it can send the appropriate calls to the devices. We utilize the AsyncIO python library to generate an event loop. We use this event loop to schedule when and what order calls to the motors and sensors are done, and to enable or disable a flag telling the action coordinator that the vehicle is in a specific action.

An example of what happens when the Motion Coordinator gets an order from the Action Coordinator is as follows: The coordinator gets the action of moving forward until it reaches a node. The class first tells the Action Coordinator that it has started this action. The coordinator then schedules on the event loop for a call to the left and right motor to go clockwise. This will cause the vehicle to move forward. While the vehicle moves forward is happening the coordinator tells the IR sensors to start scanning for a node and it enables the error correction software. The node detection and error correction systems are described in a later part of the report. Once the vehicle reaches the node, the sensors will send that information to the coordinator and it will stop the motors and tell the Action Coordinator it has fulfilled that action. If there is something else on the action queue the Motion Coordinator will immediately move on to that action.

4.2.1.4 Motor Class

To ensure that the software is readable, interactable and adheres to the basic principles of object orientation, the motors are defined in a class that the rest of the software could easily interface with. This motor class is designed to send the appropriate outputs to the L293D H-Bridge circuit. The class is essentially designed to output the same truth table (table 4.2) from earlier to each motor. There is a clockwise and counter-clockwise function that tells the specific motor what direction to go. There is also a change speed function, which changes the duty cycle of the motors to regulate how much power, and therefore, how much speed, the motor has.

We initialize two instances of this class in the motion coordinator, one for each motor. When initializing this class, we give it appropriate parameters and the associated GPIO pins that are connected to the L293D IC, the two input pins and the enable pin of the H-Bridge IC.

4.2.1.5 IR Class

There is only one pin for the IR input for each sensor when the sensor detects the defined margins, it will output true to the PINE64. We use this property in the classes main check function, which checks what is the status of the sensors and returns either true or false. Before we return true or false, we perform some of the error correction operations. These operations will be detailed in the later section 4.2.2.2. Each of the motors are programmatically attached to one of the IR sensors, so the IR sensors can easily interface with that motor and change some of its properties.

The IR class is instanced in the motion coordinator twice for each sensor. The initialization parameters are as follows: The GPIO pin associated with receiving the input of the TCRT1000, if the sensor detects the white coloured nodes, it returns TRUE. The motor that is associated with the specific IR instance so that this class can interface with that instance of the motor. And the inherent speed difference of the motor, which is used so that we can calibrate the motors of the vehicle to correct for error.

4.2.2 Autonomous functionality

This section covers three elements of our vehicle's autonomous functionality:

- Localization
- Error Correction System
- Status Updater

4.2.2.1 Localization

In order to efficiently coordinate the vehicles, the base station needed an accurate method of localization. Every vehicle in our system relays to the base station its location on the track. This feature is analogous to a GPS system in a real autonomous vehicle. If our system were implemented on fully autonomous vehicles, it would keep track of where the vehicle is and give it directives on that basis. We developed a system that emulates this behaviour.

The defining inputs of our localization system were the Infrared Sensors. We use these sensors to detect for nodes (white lines that both IR sensors detect) we placed on the track. The space between two nodes is considered one unit of movement. Given the Forward command, the vehicle goes forward until the sensors detect a node. The vehicle only needs to send to the base station how many commands the vehicle has fulfilled, and what command it is currently doing to accurately discern where the vehicles are in space.

For example, if the vehicle is at point A, and is going to go to point B. If the vehicle sends to the base station that is is currently completing a forward command, the base station knows that the vehicle is between A and B. When the vehicle reaches B it tells the base station it completed the forward command, and so the base station knows the vehicle is exactly at point B.

At first, we considered using a camera attached to the base station to record a video of the track and vehicles, and then process the vehicles positions to send to the base station. We instead opted for a solution that used the vehicles themselves.

The base station detects nodes by using the fact that there are two IR sensors installed on the vehicle. First off, the IR sensors themselves do not detect black colored surfaces, so we took advantage of this by making the nodes white on a black background. We use the IR sensors as inputs into our error correction system (See the following section 4.3.2), using one sensor at a time. When both sensors are triggered, the vehicle has reached a node and the current command is ended.

4.2.2.2 Error correction system

The error correction system senses when a vehicle is veering off in one direction and adjusts the power in the motors to correct.

In our system, we used a single straight line to denote the lanes and the IR sensors were responsible for detecting these road lines. The IR sensors are tuned to only detect the white colors we define on our track, so we simply placed the road line in between each sensor. The distance between each sensor is roughly 2 cm, so we made the width of each line about 1.7 cm so that the IR sensors wound not accidentally detect the line with both of its sensors, as that triggers the aforementioned node detection system. These lines link one node to another. The

error correction system activates when one IR sensor detects a white line, the respective motor next to that sensor will slow down slightly, adjusting the direction of the vehicle back towards the line. The IR sensors define what motor they are associated to. Whenever a sensor detects a white line, a signal slows the motor down by a few percentage points. The vehicles are able to follow a straight line using this system.

4.2.2.3 Status updater

The main interface from the base station to the vehicle is the command system, the main interface from the vehicle to the base station is the status updater. The base station requires frequent updates from all the vehicles in our system. It cannot afford to receive pertinent information in bits and pieces or in different Bluetooth packages, so the vehicle is required to send these updates in a single, neat, serialized data form. The base station is expected to receive this information from all the vehicles so it can process where all the vehicles are, estimate where they will end up and when, and coordinate the vehicles as efficient as possible.

What the vehicle provides the following information to the base station:

- The speed of the vehicle.
- The command the vehicle is currently carrying out.
- The queue of commands the vehicle has left to do.

This was all the information the base station needs in order to generate an accurate system. We compile all of this information into a single status function that accesses the pertinent data and returns a serialized JSON string, this string is then accessible to the base station when needed. This method is more efficient than constantly sending data to the base station.

4.3 Track

To fully test the vehicle network, we required a track designed to stress test one intersection. The vehicles, communicating with the base station, are able to discern when to turn at the intersection and not collide with each other. The track requires a simple two-lane road to simulate a real traffic environment. The track needs to properly work with the vehicles in our network, so the track has to be designed with the appropriate nodes and error detection margins previously

described in this section. To achieve all of this we settled on a simple, two-lane, figure eight track.

Figure 4.9: The track.

In Figure 4.9 the track has two major features; the nodes and the margins. The nodes are 7 cm^2 squares, that the vehicles detect as junctions they reach, the placement of the nodes allow us to discretise the length of the system. Each node is viewed as one unit of movements in our system. When we tell the vehicle to move forward, it moves forward until it reaches the node, at which point it can inform the base station it has fulfilled its movement. The margins are lines the vehicle scans for, it uses this to correct its angle when it moves forward. This is discussed in the error correction subsection 4.2.2.2. The track has a black background uses white colours for the nodes and margins. This contrast makes it easier for the vehicles IR sensor to discern what it is detecting.

The track is roughly 2 m^2. It has two lanes, and outer and an inner lane. The center is the main intersection, which has four nodes. When a vehicle reaches the intersection, the base station will it when it is allowed to go in order to avoid colliding with other vehicles. The lanes are designed to fit our vehicles, which are 14 cm wide. The nodes are 7 cm^2 and placed at the corners of the road, the intersections and we placed nodes in between the corners to further discretize the units of movement.

Chapter 5

Future Work

5.1 Improvements/Lessons Learned

In this section we outline improvements we would like to make to the three modules of our project, as well as lessons learned from issues experienced throughout development.

5.1.1 Simulation

There are five areas of the simulation that we would like to improve on:

- Multiple intersections
- Pathfinding with Dijkstra's algorithm
- Different intersection routing algorithms
- Stress test
- Machine learning

Multiple intersections has always been a stretch goal for the simulation. There is a couple changes which need to be made to support more intersections. Currently, the base station stores the paths available to the intersection and distributes them as needed. If the intersections held the paths the base station would only need to know of each intersection. Each intersection would also be aware of which paths it contains that lead to other paths; this allows path finding algorithms to be implemented.

Dijkstra's algorithm computes the shortest path in a system made of nodes and costs to travel between those nodes [22]. Given intersections as the nodes and the number of points in the paths between intersections as the cost, Dijkstra's algorithm could be applied to find the shortest possible path. The base station would need to request all the path information from each intersection it is aware of and then use this information when applying Dijkstra's algorithm. To make the system more responsive to traffic, the base station could update path costs with traffic congestion to ensure the shortest path is not only dependent on distance.

In the simulation there are two different intersection routing algorithms implemented: stop sign and the automated intersection. To prove the wider benefits of the automated intersection we would like to implement a traffic light intersection to compare against. The expectation is the automated solution will combine the benefits of a stop sign and a traffic light. We expect that a stop sign offers lower average wait times in low traffic scenarios and that a traffic light offers consistent wait times regardless of the number of vehicles. We expect that the automated intersection will have reduced waiting because it doesn't require stopping if the intersection is free.

With multiple intersections to benchmark we would also like to setup a different test that allows a higher rate of vehicle arrival. Currently the road system is closed, meaning that all vehicles in the system stay in the system and no new vehicles are introduced. A proper stress test would be an open system that pushes as many vehicles as possible into the intersection. That would allow us to properly observe the efficiency of the intersection under a worst case scenario.

A stretch goal of the project was to use machine learning to perform intersection routing. The idea is that over time, with enough repetitions of the simulation we could obtain data defining how the intersection is used. How many vehicles it typically sees, from what direction are vehicles most likely to come and where do they usually need to go. With this data and a cost function that attempts to minimize waiting time and maximize throughput, we believe there is a chance to improve the automated algorithm we currently have.

5.1.2 Garbage Collection Issues

There was one unforeseen issue regarding the JSON implementation, the time it takes to write to files. After introducing the code to export data to JSON files we found a significant stutter being

introduced whenever a vehicle approached the intersection. Tracing through function calls we determined this happened because at the intersection the vehicle requests a new path which is then subsequently exported to the JSON file. After more investigation using the Unity profiler we have determined it is not actually I/O write speeds causing the stutter, it is the Unity garbage collection.

C# uses an automatic garbage collection system, much like Java. This makes memory management easier but also means the user has little to no control over when garbage collection occurs. We are able to call garbage collection at the desired time but are unable to prevent garbage collection from running during specific functions.

Initial analysis of the relevant code led to the conclusion that there might be too many lists dynamically created in the export process which could be causing the garbage collector to run. To solve that all the dynamically created lists were moved to a couple predefined and constant size lists, which would therefore not need to be cleared. After implementing this change we noticed no change in performance. What we can infer from this is that creating many lists was not causing garbage collection issues and that something else must be the problem.

With the preallocating memory test failing we were left with the fact that exporting to JSON and garbage collection will be tied together no matter what we did. In an attempt to mitigate the performance issues we implemented multi-threading. The export is not critical for the rest of the simulation to operate so it can be implemented in a separate thread while everything else continues to run. Each simulated vehicle creates a thread when it is exporting so multiple exports are capable of happening concurrently if that situation arises. We were able to get the threaded solution working but it did not improve performance either. The data exported correctly and we were able to confirm that it was running in a separate thread. It seems that while exporting causes a lot of garbage collection, that garbage collector will not run in the exporting thread. The garbage collection occurs after the threading is done and continues to cause the stutter.

While we were unable to find a satisfactory solution to the export stutter it does not prevent the simulation from functioning as it should. The stutter occurs for less than one second, not long enough to affect anything else. For presentations of just the simulation and for benchmarking algorithms the exporting can be turned of to ensure proper timing and many the simulation

more visually appealing.

5.1.3 Communication System

Regarding future improvement, we can scale up the communication system to be capable of controlling real traffic systems. This requires more powerful hardware with more reliable signal transmitting, a graphical user interface for easier interaction and development, and a more accurate error correction and detection algorithm. This next section will explain the methods to improve the performance and scale up the project.

On the hardware and protocol side, IEEE 802.11p (WAVE) would be the choice to implement on real system as we mentioned in Section 3.2: Design Decision, IEEE 802.11p is going to be a standard for controlling smart vehicles. Furthermore, we need a better understanding of real-time embedded systems and embedded systems to meet hard deadlines, which is a requirement for a real-life traffic system.

Next, we would improve the interaction between humans and system by providing a graphical user interface. We were not able to do this due to the time constraints of the project, but if we were able to create one, it would enhance the end user experience. All the data we got so far is printed in the command line, which requires a trained user to interact with. We would have reformatted the data into meaningful events or commands. For example: if one of the vehicles connects with the base station, we can have one LED figure on the graphical user interface turn green to indicate that the vehicle is now connected and waiting for commands.

Finally, the communication system must not be error-prone, or at least have an auto recovery from failure mechanism. Delivering the wrong command or having unexpected behavior will led to fatal consequence in real-life. In order to avoid the catastrophe, this system must be able to automatically correct for error in a timely manner.

We experienced some delays that take up to two seconds, which means the deadline for vehicles to execute each command is not strictly followed. A hard real-time system is not possible for our communication system because we would have to switch to a different OS that would not have as much support for Bluetooth. This problem is mitigated by having the

vehicles update their status to the base station regularly, so the base station can safely control the vehicles.

5.1.4 Vehicle

Constructing five vehicles from scratch seemed to be a manageable task. As can be seen in the Vehicle chapter, we were able to formulate a relatively robust design for the vehicles, both in the hardware and software. Being able to build interfaces with DC motors and sensors, and processing the data to and from all of these components is well within the scope of Electrical and Computer Engineering. However, in constructing these vehicles, we did encounter problems that, as ECE students, we were not suited to compensate for.

There was a relatively large mechanical component to the construction of the vehicles, and as much as we tried to simplify the physical mechanics of the design, we still encountered a multitude of mechanical problems that were outside the scope of this project and our skill sets. These problems included things like accounting for an uneven center of mass and the wheels not having enough traction. We attempted to debug and compensate for these problems to the best of our abilities, but we did not possess the mechanical insights to formulate the best solution. As a result, when we tested a full demonstration of the system, we encountered many problems.

When we initially built the vehicle prototype, we kept all the components on in the lower compartment of the chassis. After testing, we found the center of mass of the vehicle was slightly imbalanced. This lead to more veering than our error correction system could account for, so it would occasionally veer off the track, which was detrimental to our system. We attempted to change the weight configuration of the vehicle by distributing the components differently. We changed where the power supply and batteries were placed, this allowed for the vehicle to drive straight enough for the error correction system to work.

The mechanical problem we were not able to fully account for was the traction of the wheels on some vehicles. The wheels attached to the DC motors seemed to have no trouble moving the vehicle forward and backward, as both motors were active. But when we tested turning, we found that not all of the DC motors we used were of the same quality, some of the motors did not deliver the same amount of torque to the vehicle. Initially, when we programmed turns in the vehicle, we had it set up so that one wheel would be turning, and the vehicle would stop

once it turned a certain amount of degrees. This worked for the initial prototype, but we found, due to building all our vehicles with cost efficient, low-end motors, that it produced variable results when turning. When turning, some of the motors of the vehicle would simply hang or turn without moving the whole vehicle. We concluded that the DC motors had deteriorated in quality after testing all the motors for a period of time. We attempted to compensate of this problem by changing up the code for the turning and trying different configurations of weight distribution, as we had done when the center of mass was off on the vehicle but had no success on fixing the problematic vehicles. We wanted to find a solution to this problem, but concluded that this was a mechanical problem and dedicated more time on optimizing the control, communication and coordination systems, which were the focus of the project.

5.2 Future Design Alternatives

Over the course of our project, we discussed a number of ways we could extend our work in a real world scenario, and many of the design decisions made were due to the constraints of the budget and resources. This section will discuss four options should this project ever be scaled up.

The largest improvement that could be made is the ability to have network communication between each vehicle. The decision to set a separate computer to process and coordinate the vehicles positions was made to keep the computational load on each vehicle light. With a more powerful processor in each vehicle the need for a separate base station would be eliminated. Each vehicle would be able to communicate with each other directly. We think the best way to manage this new network design would be to choose one vehicle to act as the base station, accepting information about all the other vehicles position and next position, and sending the appropriate commands to the other vehicles to coordinate their movements. Imagining a real four-way intersection on any given street. As vehicles enter they connect to each other and a base station vehicle is chosen, likely based on best average connection to all other vehicles. New vehicles that enter later are also connected to the network and are coordinated until they leave its proximity. This method would eliminate the need for a dedicated controller stationed at each intersection and is more efficient as a network at an intersection only exists for as long as there are at least two vehicles entering it.

One of the earliest challenges we faced was in this project was deciding on if the vehicles or the base station should have an internal map that they follow or if they should dynamically sense and route their paths through the intersection. The latter is very difficult to implement. Modern vehicles are all equipped with GPS and accurate map data, meaning each vehicle can route its own path and know the relative positions of other vehicles around them. Using GPS we can perform error correction between the simulation and the real position of the vehicles. The map data that comes with GPS applications is also extremely useful, as we would not have to manually map out the details of each intersection, but instead it would be very easy to dynamically assign node positions based on some given map data and apply our coordination management directly.

The choice of Bluetooth as our communication protocol was also made to fit the particulars of our demonstration. Bluetooth is not fitting for the distances between full sized vehicles on the road, neither is the number of clients it can maintain at once. A better choice for communication protocol would be necessary to expand our project. A protocol that supports a range of around 15 metres, multiple connections that can drop in and out at any time, self repairing networks, broadcasting, and other features could be used to appropriately scale our project up. Only some communication details would need to be changed to support a new protocol, largely our project could remain unchanged.

Sensors are also very useful as a tool for error correction and collision avoidance. Any amount of direct physical data we can apply to our simulation of the intersection can be used to adjust for any drift that occurs between where the simulation thinks a vehicle is and where it really is. We already made use of IR sensors for this purpose by following within a guideline, this idea could be further extended to include cameras with computer vision, range detectors, or ultrasonic sensors. Camera object tracking technology is steadily getting more reliable and accurate, we could equip each vehicle with its own cameras that are trained to detect other vehicles or obstructions and use that data to update our simulated map on the fly. Range sensors would give us an exact measurement of how far vehicles are away and allow us to make accurate judgements of their trajectories. The sensors could also be used as a failsafe to our coordination system by providing and absolute limit to how close vehicles are allowed to come before requiring to stop.

Chapter 6

Conclusion

As vehicle technology becomes increasingly automated, we believe networking and automated traffic flow solutions will need to be developed. To that effect we succeeded in designing a small scale implementation that shows the viability of a smart traffic network. Our core project requirements are:

- Build at least two miniature autonomous vehicles.
- Create a network which allows these vehicles to communicate with a base station.
- Create a simulation to test intersection routing algorithms.
- Integrate the three modules to create a demonstration.

We successfully built five autonomous vehicles, capable of receiving and executing commands via Bluetooth. Our base station manages up to four Bluetooth connections and distributes commands from the simulation. In the simulation we successfully implemented two different routing algorithms. Integration between the simulation and communication system is achieved through the use of JSON files and Bluetooth integrates the communication system and vehicles. Together, the three modules are able to coordinate the vehicles and update the simulation to maintain an accurate representation of the vehicles positions. We believe we showcase the viability of a smart traffic network with this demonstration.

We were able to successfully simulate the full system but due to mechanical issues were not able to show the full demonstration with all our vehicles. Our inexpensive motors were unable

to achieve the necessary torque to provide consistent turning.

In the future we would fix the mechanical issues present in the vehicles, switch to a communication protocol which scales to a real life traffic situation and improve the intersection routing algorithms.

Appendices

Appendix A

Code

All the code for the vehicles, communication system and simulation can be found at: https://github.com/Farhannibal/ECE4600

This includes:

- Motor control software
- Bluetooth server
- Bluetooth client
- Unity simulation

Appendix B

Budget

Ordered from	Order	Cost
PINE64.com	PINE64 + Wi-Fi modules	$147.68
Digikey	Ultrasonic Sensors*	$36.73
Banggood.com	DC Motors	$36.73
Digikey	TCRT1000 IR Sensors	$21.35
Digikey	Jumper Wires	$35.78
Amazon	Power Bank + SD Cards	$125.00
Filaments.ca	3D Printing Filaments	$84.79
Total	$478.12	
		*Did not use in final design

Table B.1: Total budget

Bibliography

[1] H. Summala, ”Brake Reaction Times and Driver Behavior Analysis,” Transportation Human Factors, vol. 2, no. 3, pp. 217-226, 2000.

[2] ”Zigbee.org,” 2018. [Online]. Available: http://www.zigbee.org/what-is-zigbee/. [Accessed 21 February 2018].

[3] S. Eichler, ”citeseerx.ist.edu,” 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.5573&rep=rep1&type=pdf. [Accessed 05 March 2018].

[4] ”json.org,” 2018. [Online]. Available: https://www.json.org/. [Accessed 21 February 2018].

[5] ”Pine64.org,” PINE64, 2018. [Online]. Available: https://www.pine64.org/?product=wifi-802-11bgn-bluetooth-4-0-module. [Accessed 25 February 2018].

[6] ”Amazon.ca,” Amazon, 2018. [Online]. Available: https://www.amazon.ca/USB-Bluetooth-4-0-Dongle-Adapter/dp/B075GKYYHP?pd_rd_wg=mfq6Z&pd_rd_r=b2e95308-b539-46a3-83d8-1f64ee75fb5f&pd_rd_w=UjWev&ref_=pd_gw_simh&pf_rd_r=4GP9BS2B7VNMEVSBC7F
19a3-5a54-add4-62c82ff200de. [Accessed 25 February 2018].

[7] ”arduino.cc,” Arduino, 2018. [Online]. Available: https://www.arduino.cc/en/Guide/Introduction. [Accessed 8 March 2018].

[8] ”http://www.futureelectronics.com,” Future Electronics, 2018. [Online]. Available: http://www.futureelectronics.com/en/technologies/semiconductors/wireless-rf/rf-modules-solutions/802154-zigbee/Pages/7606452-MRF24J40MAT-I-RM.aspx?IM=0. [Accessed 8 March 2018].

[9] ”sparkfun.com,” Spark Fun, 2018. [Online]. Available: https://www.sparkfun.com/products/13959. [Accessed 08 March 2018].

[10] ”vishay.com,” Vishay, 11 June 12. [Online]. Available: https://www.vishay.com/docs/83752/tcrt1000.pdf. [Accessed 8 March 2018].

[11] ”Pine64,” PINE64, 2018. [Online]. Available: https://www.pine64.org/?product=pine-a64-board. [Accessed 08 March 2018].

[12] ”Pine64,” Pine64Wifi, 2018. [Online]. Available: https://www.pine64.org/?product=wifi-802-11bgn-bluetooth-4-0-module . [Accessed 8 March 2018].

[13] ”digikey,” Digikey, 2018. [Online]. Available: https://www.digikey.ca/product-detail/en/sparkfun-electronics/SEN-13959/1568-1421-ND/6193598. [Accessed 8 March 2018].

[14] ”ebay,” Ebay, 2018. [Online]. Available: http://www.ebay.ca/itm/10Pcs-L293D-L293-Push-Pull-Four-Channel-Motor-Driver-IC-/140832908826 . [Accessed 8 March 2018].

[15] ”ebay,” Ebay, 2018. [Online]. Available: http://www.ebay.ca/itm/Smart-Car-Robot-Plastic-Tire-Wheel-with-DC-3-6V-Gear-Motor-for-Robot-65-27MM-/191901698035?hash=item2cae3b87f3:g:U6QAAOSwmtJXZMEb. [Accessed 8 March 2018].

[16] ”Pine64,” Pine64, 2018. [Online]. Available: https://www.pine64.org/?product=lithium-polymer-battery-us-only. [Accessed 8 March 2018].

[17] ”NewEgg,” NewEgg, 2018. [Online]. Available: https://www.newegg.ca/Product/Product.aspx?Item=N82E
_-20-211-742-_-Product. [Accessed 8 March 2018].

[18] ”Digikey,” Digikey, 2018. [Online]. Available: https://www.digikey.ca/product-detail/en/vishay-semiconductor-opto-division/TCRT1000/751-1031-ND/1681165. [Accessed 8 March 2018].

[19] ”Amazon,” Amazon, 2018. [Online]. Available: https://www.amazon.ca/Poweradd-Ultra-compact-Portable-External-Identify/dp/B00KG45W08/ref=pd_sim_23_9?_encoding=UTF8&refRID=YTDE
[Accessed 8 March 2018].

[20] ”Amazon,” Amazon, 2018. [Online]. Available: https://www.amazon.ca/Sandisk-Ultra-Micro-UHS-I-Adapter/dp/B073K14CVB/ref=sr_1_6?s=pc&ie=UTF8&qid=1515614714&sr=1-6. [Accessed 08 March 2018].

[21] ”Github,” Github, 2018. [Online]. Available: https://github.com/python/asyncio/wiki. [Accessed 8 March 2018].

[22] ”Dijkstra’s algorithm,” Wikipedia, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm. [Accessed 8 March 2018].