Vehicle to Vehicle Communication for an Autonomous Vehicle

FINAL REPORT

Introduction

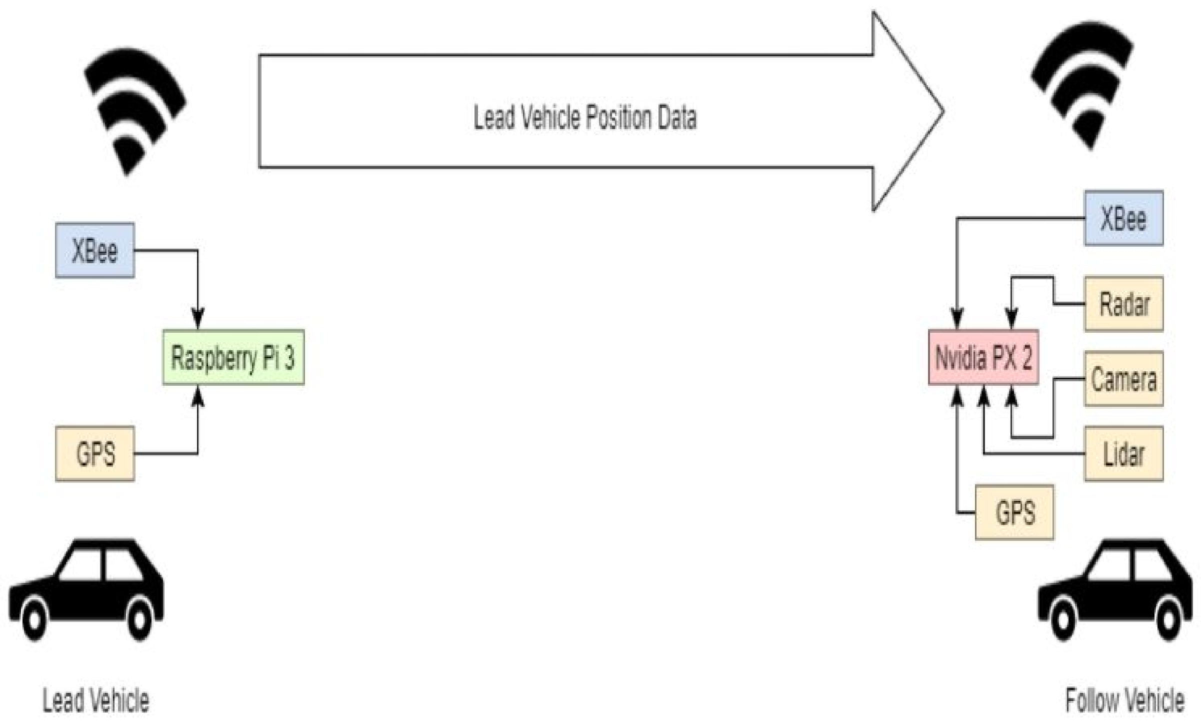
As part of our senior design we had to create the means for the communication between two vehicles. The information that had to be communicated was the latitude, longitude coordinates, and the time. For this we have come up with several solutions throughout our design and testing process. We looked into DSRC, Cellular Communication, Raspberry Pi's over a LAN network, and Xbee transceivers. All these are viable options and several can complement the others pretty well. We will have more details in following sections.

Acknowledgement

First off, we would like to formally thank Dr. Hegde for guidance on our project and the insightful advice on how to approach and overcome different obstacles of our progress. Also we would like to thank Vishal as our project manager who made sure we stuck to our milestones and meeting with us every week to make sure we were working towards our goals. Both of the faculty members have been a significant presence to our progress in this group project.

Problem and Project Statement

General Problem:
Our project revolves around transportation, but a transportation system that ranges more than simply accommodating people from place to place. When we think about transportation of any kind, the first thought that comes to our minds is being inside a certain vehicle, whether it be a person or an object being transported. We would normally think that a person would be manning that certain vehicle: however, the approach here is to transport both a person and a vehicle without having a person operating each one. Having one person operate multiple vehicles does offer a new aspect to transportation. It's similar to having a locomotive pulling multiple cargo carts except this one isn't on tracks, it's on the open road. This is very important because for years we have relied on trains for mass transportation of goods, but they are limited to railroad stations. With the rapid growth of local roads and highway systems, we also rely on big trucks but their cargo is much more limited compared to trains and this is where we offer a solution to mass local shipping. Having a lead truck driver lead multiple autonomous following trucks, we are not limiting mass transportation on the tracks anymore, it can be on any road or highway which saves both time and money for shipping companies. Our group more specifically is in charge of relaying the position of the leading vehicle to any vehicle that may be following.



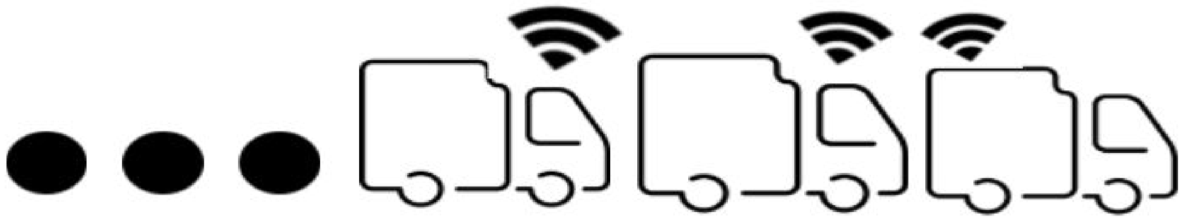
General Solution:
Replacing big trucks drivers is only one of the main uses for this project. The goal for this project is to have an autonomous car capable of following a manned lead car. The autonomous car functions mainly off of information gathered by multiple sensors such as location of the lead car and the obstacles that may lie in between the two cars. This kind of project requires several teams such as a mechanical team, robotics team, controls team, and an electrical team.
As part of the electrical team our role is more defined. We have two main tasks to help create this autonomous vehicle solution. The first task and most important is sending the location of the lead vehicle to the following vehicles. We had three different methods that we had planned on implementing. First we had two transceivers attached to the gps on the lead vehicle as well as one attached to the ROS on the following vehicle. We originally planned on using DSRC but have decided to go with two XBee transceivers. We also planned on using 4G LTE cellular to transmit the location for when the vehicles are at farther apart locations. We also use a raspberry pi which is similar to a small computer. This allows us to retrieve the location information directly from the GPS. We have currently used an XBee and a the raspberry pi to transmit the information. We interpreted the data coming out of the GPS and creating something useful out of it.
We also made hardware and software adjustments to fit our need. Throughout the project we have been soldering on wires to the right connectors. Finding parts for the other groups or anything that they need that we may have. One example is that we attached the proper connectors and power source to the radar that we are using. At the end of this project we have created a reliable way of sending messages from one vehicle to the other, which is sufficient enough for other teams to implement the autonomous car.

Operational environment

For our simulation and testing, we did the prototype testing process over by INTRANS. Since our data transmission is strictly limited between the two vehicles, our first evaluation was preferably on a open field where we could test the trajectory planning of the following vehicle with the lead car going one direction. We are most interested in the Xbee sending the information from the GPS at a distance of at least 300 feet. We did this testing first by making sure that information can be received at 50ft, 100ft, and so on with the vehicles stopped. Then we tested how well the devices did when one vehicle was moving and when both vehicles were moving. Once we are able to reach 300 ft we considered the communication complete and ready for improvement. Our final implementation surpassed this requirement. We implemented vehicles to work in any type of environment which indicates that our vehicle to cover all our components to all types of weather. The mechanical team did all the insulation from the weather. For the GPS and the XBee transceivers we had the antennas mounted on the top of the vehicle. For the GPS the antenna it has a magnetic bottom that allows the antenna to attach to a metal surface. Since not all cars have a metal surface it also comes with an adhesive surface that can make it stick permanently to the top of a car. For the sake of testing we do not want it to permanently stick to our cars so we used metal cars and went out on not so windy days. The XBee extended antennas also do best when on top of a car so for testing we held them outside the window. For final implementation they were mounted near the front of the following vehicle and near the back of the leading vehicle.

Intended uses and users

We look at two different uses for our project: the larger project an autonomous car following the coordinates of a lead vehicle or destination, and our more specific project which is creating the communication between the lead vehicle and the following vehicle. For the larger project we see the end user being a company with many cargo trucks that wish to cut down on number of drivers. We can also see the army or other armed forces that would like to transport a whole base or move supplies, having an autonomous car that only requires one driver may be beneficial. The uses are mostly just moving materials and goods with less expense. Someone with a lot of money may as well wish to move their car collection all together at once such as in a parade. For the more specific project that we are dealing with the user can vary much more. We can see anyone dealing with communication wanting to buy our final product for their own communication. There are many road object avoidance applications that may benefit from our product. Road safety applications as well as any type of medium range communication may wish to use our XBee communication method. As we have seen from our testing when held high the antennas do have a very large range. This implementation can be very useful when out at sea where this product can possibly work for up to nine miles. It could also be attached to other devices that may need the location of a close by target.



Assumptions and limitations

For our end product, we are assuming that the main power source for all of our electronic hardware sensors will be a 12V battery. The reason that the power source will have to be 12 volts is due to the fact that the radar has the maximum voltage intake of 24 volts but that we have created a 12 to 24 volt converter for it. The rest of the sensors that operates along with the radar is below 24 volts and in order to avoid over-currents, we used multiple DC to DC converters to lower the voltage intake from the primary battery or step ups depending on which sensor we are powering. All the components vary from about 5 volts to 24 volts. We also assume that the lead vehicle will have a USB port as most cars do. This is to turn on the GPS and Raspberry Pi automatically when the car turns on.
Our second assumption is that the following car must be able to receive the data from the lead car from an average distance of 300 feet at least. This distance is not set to stone, but our primary goal is to have a data transmit range that is long enough so that incase the following car is stopped by an obstacle/traffic light that is between the following and the lead car, the lead car can transmit its X,Y coordinate location so that the following car can get a rough approximation of where the lead car is heading.
Our last, but not least assumption will be that the communication between the ROS and sensor receivers will be operated using two computers that one has ROS installed on it and the other has python on a raspberry pi. We planned on installing a small monitor on the lead vehicle in order to communicate more effectively with the GPS and Raspberry Pi. We ended up only using it for setting up the GPS initially and programming the Raspberry Pi as our assumption that we would have to manually set all the GPS parameters every time were actually able to be turned into automatic events instead.

One of the biggest limitations was getting the right range of transmission for a reasonable price. One of the first options was DSRC transmitters as they are meant for road related operations. The problem with it was that it had more legal documentation as well as higher cost to implement. We wanted something else that could do the job for a cheaper price especially since most DSRC devices work at about 200 feet instead of the minimum 300 feet required by our project. It also had a 5.9GHz which made it bad to go around buildings and cars so we needed a lower frequency device. We picked the XBee models and liked them and eventually found the Pro S3B which as you will see really became the appropriate transmitter to use.
Another limitation was that we needed to transmit information at 10 Hz. This is a limitation because most devices have to change their baud rate or the amount of information being sent every cycle in order to still get the right amount of Hz.

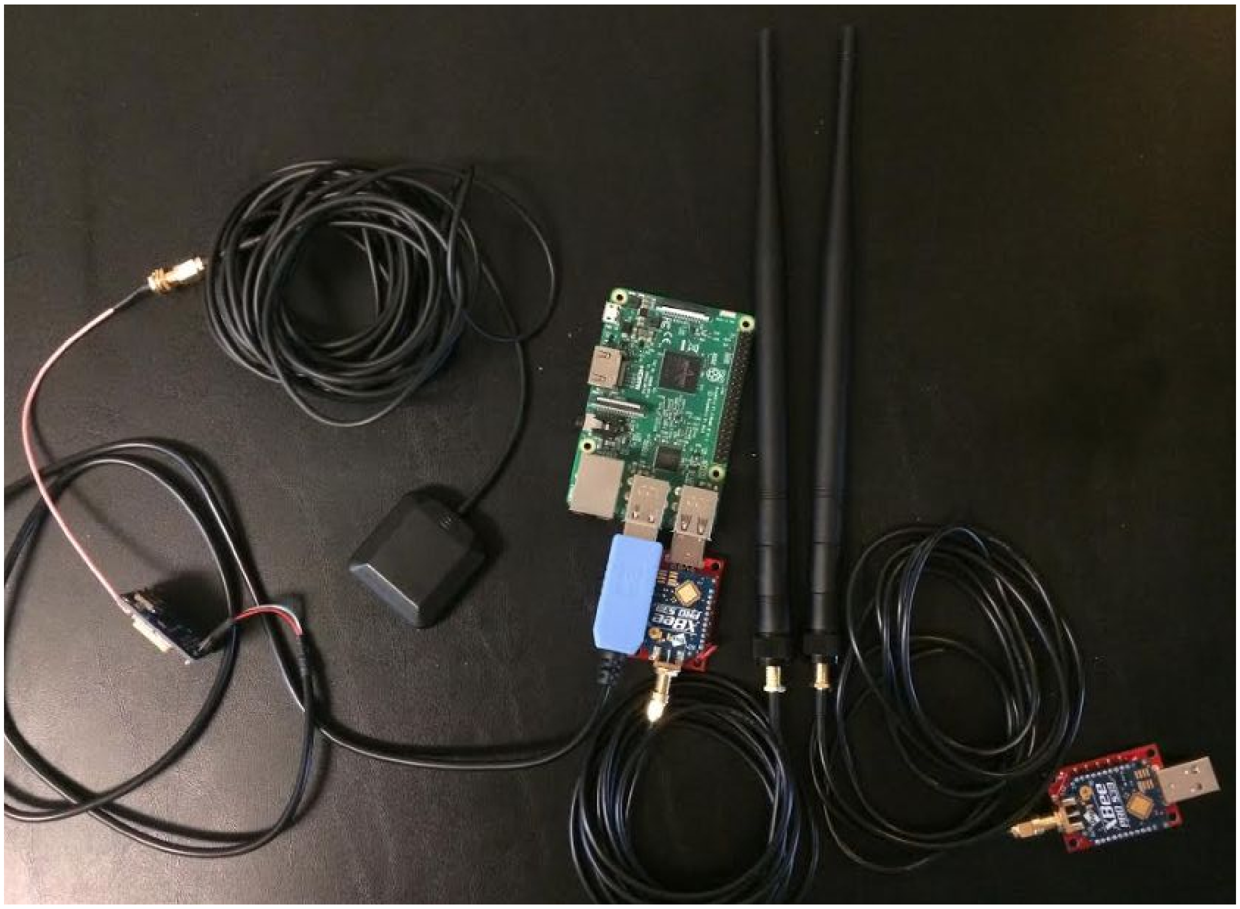
Also before we moved unto the Raspberry Pi we kept playing around with the arduino. The following is a sample of the type of code we would write to get information from the GPS using the arduino:

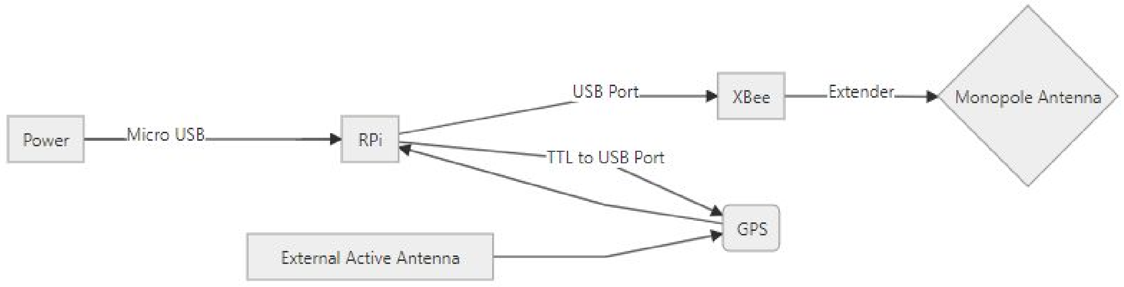
```
char rx_byte;

void setup() {

Serial.begin(9600); } void loop() {

if (Serial.available() > 0) {

    rx_byte = Serial.read();

    Serial.print("You Typed: ");

    Serial.println(rx_byte);

}

}
```

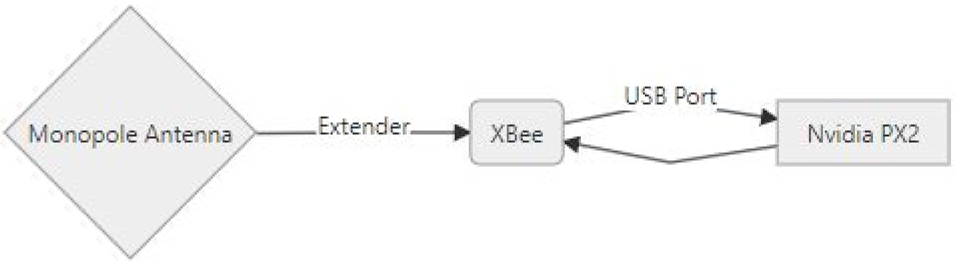
The code did become very complex as we tried more and more things with the arduino to talk to the GPS but unfortunately our connection was limited to protocol RS-232 and we were not using the proper connections to get it.

End Product

Our end product has two main parts as described in the block diagrams below. It has an external active antenna (black square patch) that is able to both with an adhesive pad that it has and a magnet connect to the top of the lead vehicle. It connects to the GPS which in turn sends coordinate and time information to the Raspberry Pi. Then an XBee is connected and sends the information to a monopole 900 MHz antenna which sends it to another XBee that connects to the Nvidia PX2(not shown in the picture). More information on how everything is connected and how it works can be found in Appendix I.




Power
Micro USB
RPi
USB Port
XBee
Extender
Monopole Antenna
TTL to USB Port
GPS
External Active Antenna


Monopole Antenna
Extender
XBee
USB Port
Nvidia PX2

Specifications and analysis

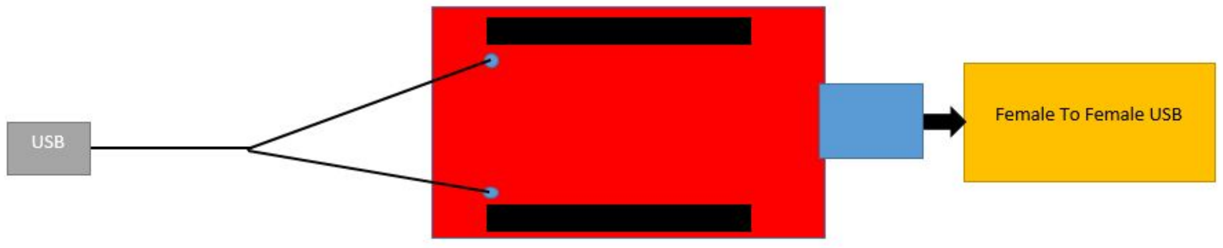
Proposed Design

Our team proposed several possible solutions to solve the problem of communicating GPS data from the lead vehicle to the following vehicle. Each solution has both pros and cons which needed to be looked at. We came up with a list of these to determine which solution out of the three is the best.

The first includes the use of two microcontrollers. One will be attached directly to the GPS on the lead vehicle, while the other would be directly connected to the NVidia PX2 on the following vehicle. The purpose of the microcontroller on the lead vehicle would be to take the serial data, compress it, and send it to the DSRC transmitter. The signals would be transmitted to a DSRC receiver which would be hooked up to the second microcontroller on the following vehicle. This microcontroller would convert the signals into data which could be read from ROS running on the Nvidia PX2.

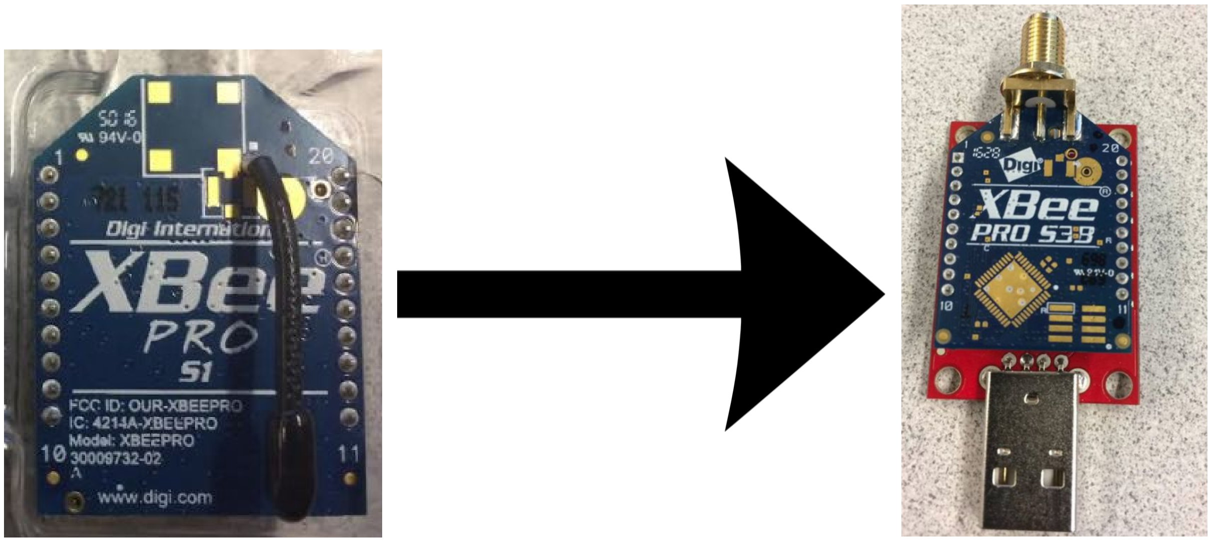
The second solution is to use a Raspberry Pi on the lead vehicle which could connect directly to the GPS. The Raspberry Pi is capable of running ROS on it which would allow it to be a network node. This would make it very easy to communicate the data back to the NVidia PX2 running ROS on the following vehicle. The only downside is the range at which these two devices would communicate through a local network wouldn't be as big. Another option is to attach transceivers such as XBees onto the raspberry pi and use python coding to upload information from the GPS and transmit it to the following vehicle.

The third solution would be to use already built transmitters and receivers called Xbees. These would connect directly to the GPS on the lead vehicle and to the NVidia PX2 on the following vehicle. The connection using this method would most likely be the simplest since the data coming directly from the GPS would be sent directly to the NVidia PX2. For this reason, it would be a lot easier to debug if things are going wrong. This could save us a bunch of time in the long run. Below the red represents the XBee and its dongle. We were using a double female to female USB to try to attach it directly to the GPS. We used another USB connected to a power source to try and power both devices through the XBee. It did not work.



Our analysis has concluded the first solution to be the hardest to debug. If something goes wrong, it would be hard to track which section in the communication wasn't correct. It would also have the best range out of the three. The second solution would suffer due to the range factor.

If the lead vehicle gets too far ahead, the connection would be lost and the following vehicle wouldn't know where to go. The last option was found to have several problems with actually receiving information from the GPS. It gave no option to power the GPS. We then had to do the second part of solution two where we combined the XBee and the raspberry pi. The code that we added to the raspberry pi was efficient luckily to the fact that the GPS can send information at roughly 10 Hz at its best. Since these decisions were made we have changed from the Xbee Pro S1 to the Xbee Pro S3B. The newer model gave us better range and it did better around cars and buildings.



We also switched from the using the XSENS 100 MTi GPS to using the Adafruit GPS. The XSENS GPS was only meant to be used for testing as it can do more than what is necessary from the lead vehicle to send. The XSENS GPS is being used on the following vehicle to help it track itself. The Adafruit GPS is cheaper by a factor of one hundred and manages to still send the required information at 10 Hz to the following vehicle.

Design Analysis

The final design works very well and manages to send the GPS coordinates at a long enough distance. There had to be some changes along the way to accommodate small unforeseen problems. We needed to make sure the GPS always sent at 10 Hz, that the right baudrate was set when the GPS was turned on. We also needed to meet the distance requirement which caused us to question the original design and look into better alternatives, luckily XBees had a newer model that could be used to meet the required distance.

Implementation details

To implement our design we needed several components:
AdaFruit GPS - (sub components include: USB to Serial Cable, GPS Patch Antenna, Adapter Cable)

Raspberry Pi - (sub components include: micro-USB to USB cable)
XBee Pro S3B - (sub components include: USB Dongle, 900 MHz Monopole Antenna, Adapter Cable)

The final working implementation had the AdaFruit GPS receiving information from a lead vehicle. We had to connect an external active antenna that can be attached on the top of a vehicle. It comes with a five meter long wire and only takes up 10mA. It did require a converter SMA to uF in order to plug in to the small socket that the GPS had. The GPS itself has a small built in antenna but in order to more successfully get a fix we needed the external antenna to be able to place it in a better location and not have the arduino susceptible to the weather conditions. We set the Adafruit to transmit at 10Hz the latitude, longitude, and time. Since we had to also send it at 10Hz it made sense for us to also send the time including the milliseconds. At first we had some trouble getting this part to work but we managed to get it working. It also needed a small circle battery (CR 1220) on the back to be able to more easily remember fixed satellites that we had used before.

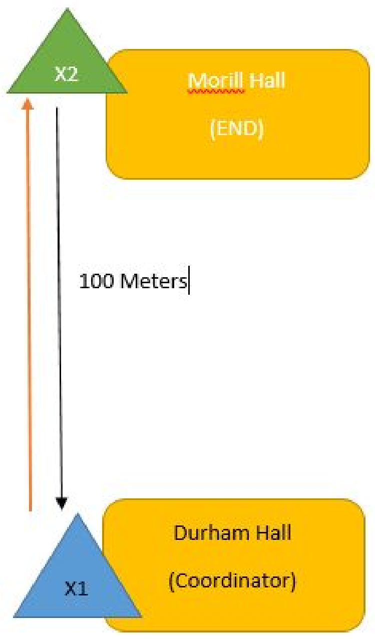
The Adafruit GPS is then connected through four of its pins to a USB to TTL serial cable. This cable is used to communicate information from the GPS to the Raspberry Pi. We added code to the Raspberry Pi to parse the information taken in from the Adafruit GPS and also code to auto detect the baudrate of the GPS. Once the the code was parsed to the correct format required by the other teams we sent the information through the XBee Pro S3B attached to the Raspberry Pi. The Pi is also powered through a micro USB to USB cable that can be installed on the lead vehicle. The moment the vehicle is turned on we have the Pi turn on the GPS and once it finds a fix it will start sending out information through the XBee.

The XBee Pro S3B is connected to one of the USB ports of the Raspberry Pi. This model was specifically chosen because of how well it does with long distance communication and with communication through walls. It works at 900 MHz using a 900 MHz Duck Antenna RP-SMA. We also got it a two meeter long extender to be able to put the antennas on top of the car for maximum efficiency.

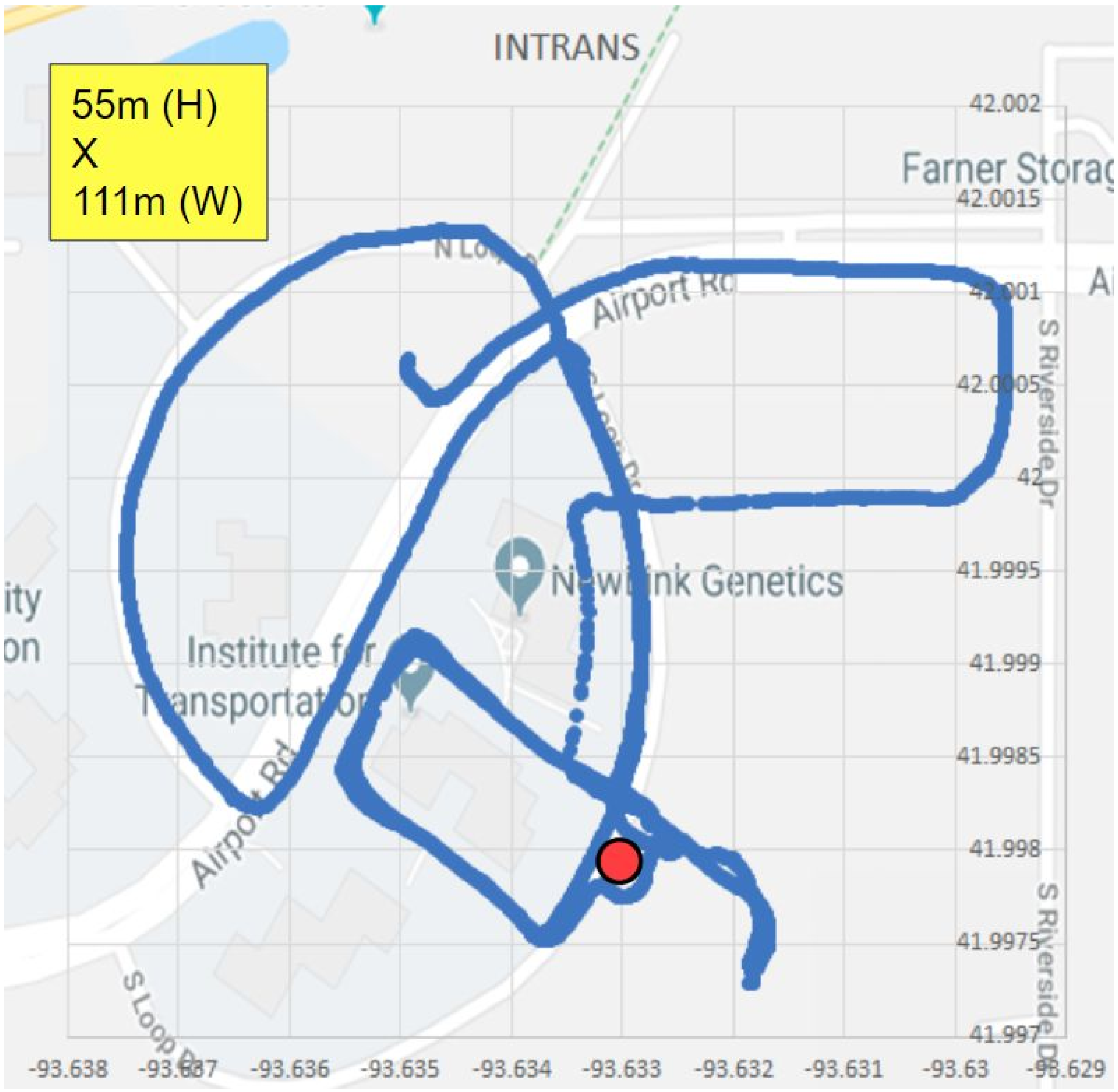
Testing process and testing results

XBee & Adafruit GPS Testing

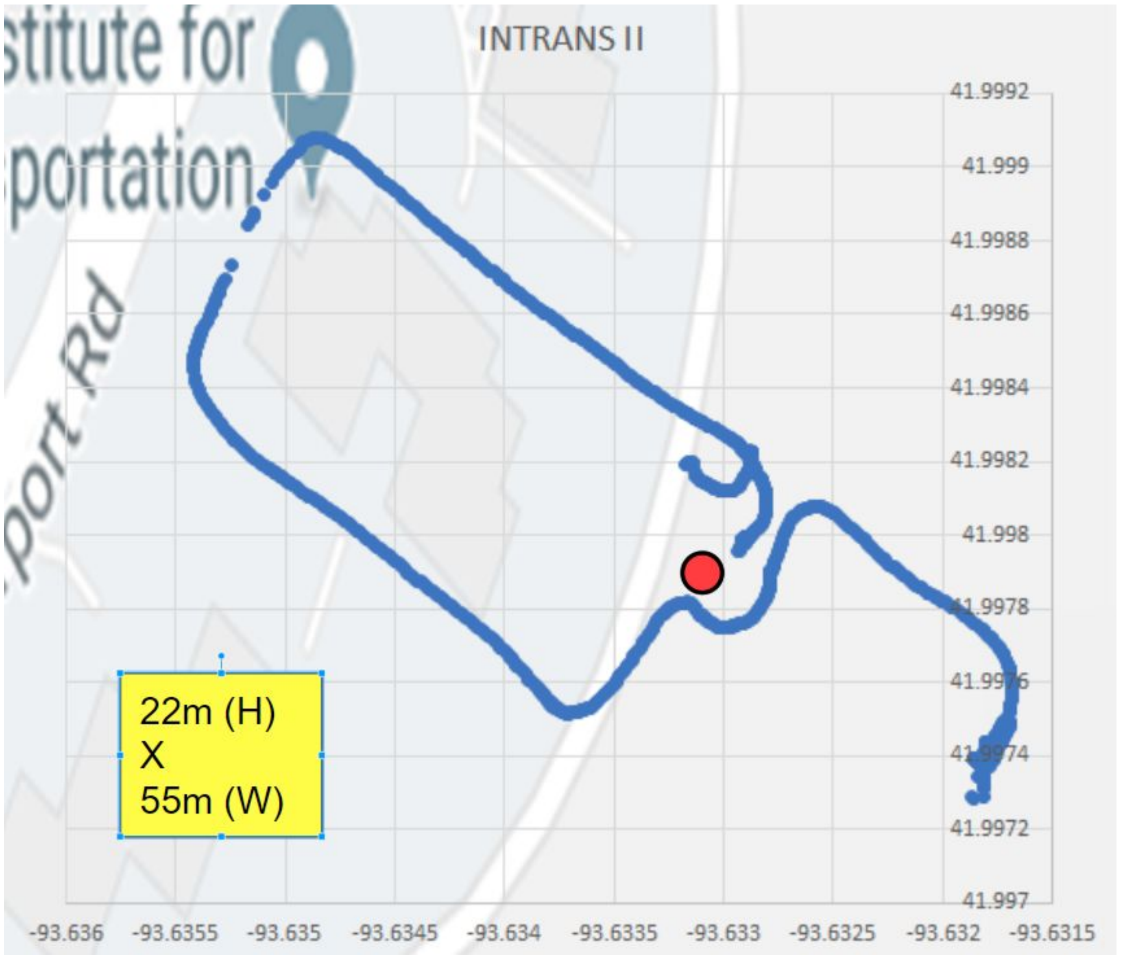
The first test was done with two XBees and two computers. We wanted to see how well they did if we sent information from one building to another. The graph below shows how well the XBee Pro S1 did. Though it seemed pretty good we could tell it had its limitations so we got the XBee Pro S3B and its tests are the ones below.



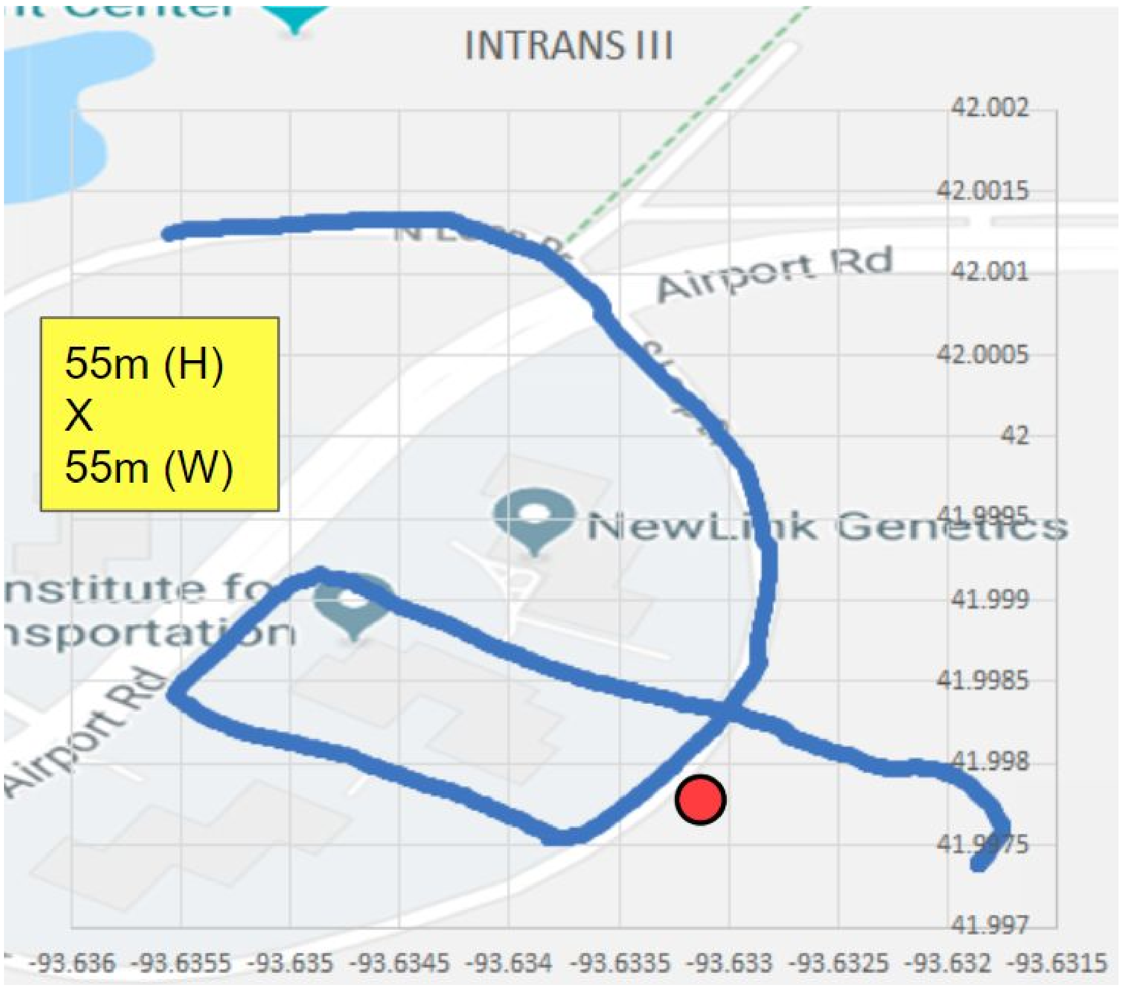
We spent the weekend figuring out how to properly connect to the Adafruit GPS. On Monday April 9th, we decided to take out the GPS along with the new XBee system out to test several things such as the frequency at which the GPS was outputting the data, make sure no packets were being lost in transmission, the distance at which the Xbees were too far apart to receive any information, and how well the devices did when the vehicles were travelling at faster speeds.
First we tested the distance at which we still got information from the gps using the XBees. They came rated at 900 MHz with 9 mile line of sight and 2000 feet for urban area. In order to test this we parked one of the cars at the red dot in the pictures below. The first picture shows all the route data that we took while we were at intrans. As you can see there was a constant inflow of data in most parts except in the middle. The following two pictures were tests to see why this happened and we concluded that it had to do with the antennas both being inside the car at that point.



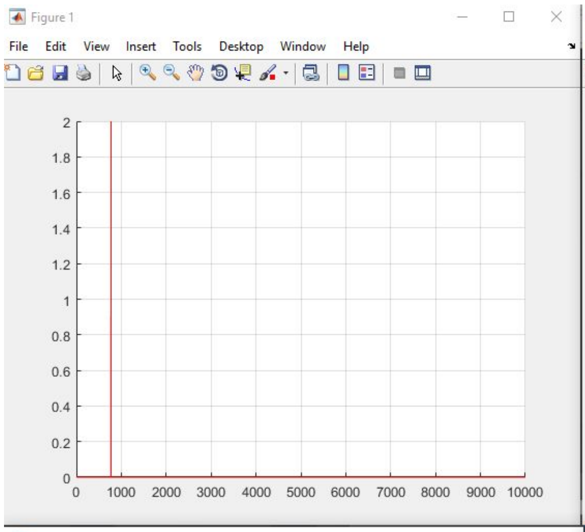
The picture below shows that as we went around a building we lost connection. This was at roughly 250 meters away. During this turn we had both antennas inside the car so we decided to redo the loop with one car still at the red dot but with both antennas outside of the car.

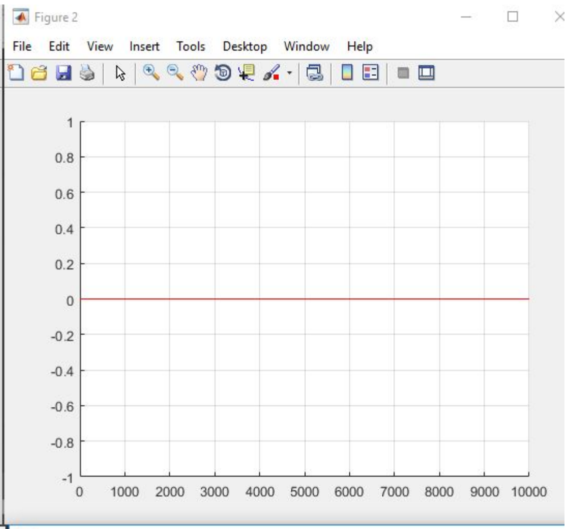


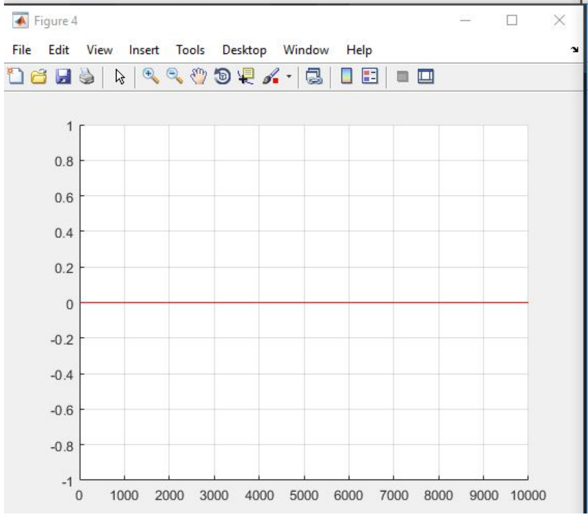
The picture below shows what happened when we had both antennas outside the car. As you can see we didn't lose any amount of data as we went around the building. The line going up farther north cannot be used for this same purpose since in that one the following car was not parked anymore and started following the car. This was part of the second part of the experiment where we drove at different velocities around Intrans and checked to see if there was a loss in data received.

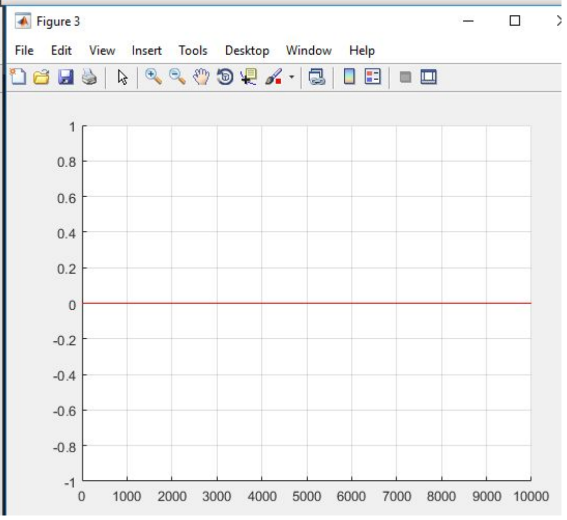


Since the speed limit around Intrans is not very large and data was being received consistently we decided to use our old code from Matlab to test how well it was transmitted at higher speeds on the highway. We tested several values going from 30mph up to 70 mph. The data at each gave the following graphs (all returned the same graph):





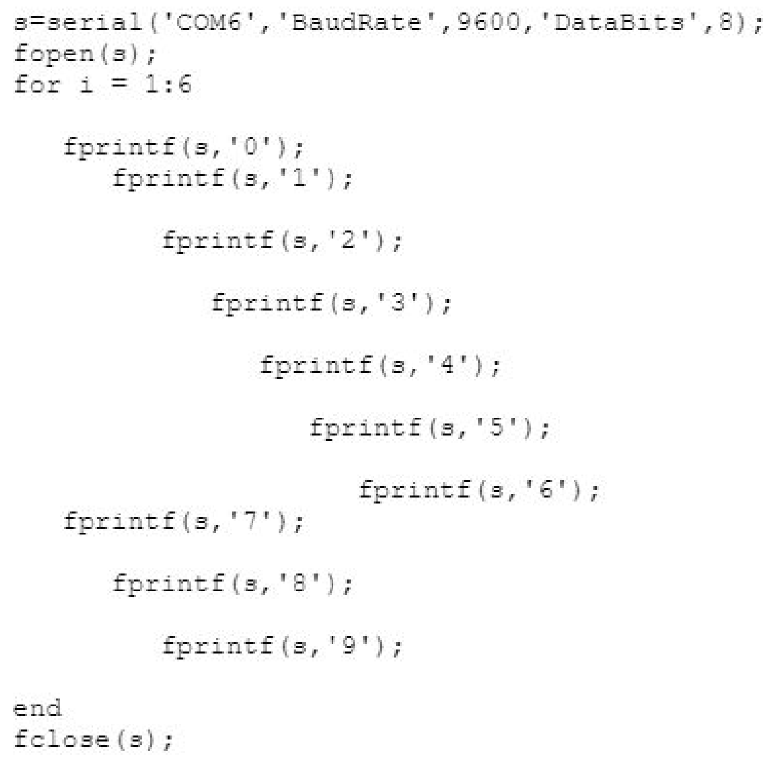




The following code was created in Matlab to create the above graphs:

```
tic;
s = serial('COM4','BaudRate',9600,'DataBits',8);
fopen(s);
f = fopen('output.csv','w');
for i = 1:10000
j = fscanf(s);
fprintf(f,j);
end
fclose(s);
toc;
```

Receiving Loop



```
s=serial('COM6','BaudRate',9600,'DataBits',8);
fopen(s);
for i = 1:6

    fprintf(s,'0');
        fprintf(s,'1');

            fprintf(s,'2');

                fprintf(s,'3');

                    fprintf(s,'4');

                        fprintf(s,'5');

                            fprintf(s,'6');
    fprintf(s,'7');

        fprintf(s,'8');

            fprintf(s,'9');

end
fclose(s);
```

Sending Loop

From top left to bottom right on the four graphs above the code the tests were 65 mph, 60 mph, 55 mph and 50 mph. The pulses that can only be found in the first graph signify lost data. As we can see there was consistently little to no data lost at these high speeds which makes us conclude that we can transmit this information at relatively high speeds. We also checked the amount of time it required for our program to finish sending the 10,000 characters that we were sending and the results are as follows:

```
Elapsed time is 30.120037 seconds.
>> ReceivingLoop
Elapsed time is 26.556675 seconds.
>> ReceivingLoop
Elapsed time is 28.530673 seconds.
>> ReceivingLoop
Elapsed time is 22.465335 seconds.
>> ReceivingLoop
Elapsed time is 27.752628 seconds.
>> ReceivingLoop
Elapsed time is 29.068317 seconds.
>> ReceivingLoop
Elapsed time is 22.337616 seconds.
>> ReceivingLoop
Elapsed time is 23.468265 seconds.
>> ReceivingLoop
```

```
Elapsed time is 26.365071 seconds.
>> ReceivingLoop
Elapsed time is 23.138971 seconds.
>> ReceivingLoop
Elapsed time is 32.510003 seconds.
>> ReceivingLoop
Elapsed time is 22.865831 seconds.
>> ReceivingLoop
Elapsed time is 22.661788 seconds.
```

All these times are relatively close to each other. Because of this we can attribute it to the nature of our own test where we have a receiving and sending loop that have to coordinate together and hence would cause the elapsed time to vary by a few seconds due to human error. If there was data lost we would have seen large amount of differences between times such as over 10 seconds.

Lastly we also checked several things about the GPS. First we checked to see if the GPS was actually sending information at 10Hz. Once we got all the data back we noticed that in reality information was actually being sent out every 5 Hz and the reason we would get information at 10 Hz was because it was echoing every piece of information and sending it twice. Over the next week this became one of the top priorities to fix.

We have also checked for the parameters of the RigRunner early on to see what limitations that it may have. Below is a small table of the information that we managed to gather:

Node	V	Ohms	Current mA
5	12.11	130	6
4	12.11	130	5.8
3	12.11	130	5.8
3	12.11	110	5.9
3	12.11	90	6
3	12.11	70	6.3
3	12.11	50	6.4

The ohms above is a load that we gave to the node. Below we have what was actually recorded as the input equivalent resistance to each node:

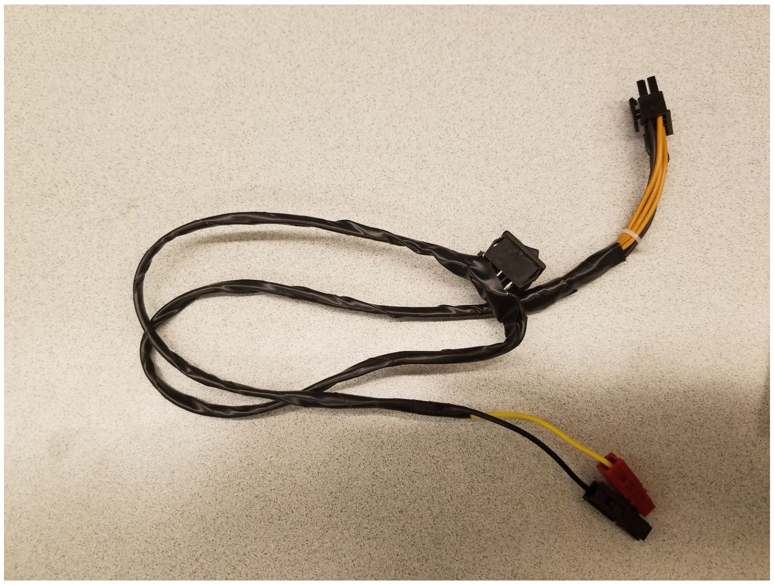
2	3	4	5

6.62Mohm	6.84Mohm	6.74Mohm	6.65Mohm

When we got the information above we had limited equipment so we came back later on to get more measurements. One thing that we were worried about was how the current would react when several different loads were placed on all the ports. We tested this by connecting loads on port two and port three. Then we tested the the current and the voltage for different values of resistance. The table below shows the results:

Node 2 Load(Ohms)	Node 3 Load(Ohms)	Current load 2 Amps	Current load 3 Amps	Voltage load 2	Voltage load 3
120	120	0.095	0.098	12.05	11.54
120	100	0.094	0.117	11.85	11.9
120	80	0.097	0.143	11.84	11.79
120	60	0.096	0.192	11.6	11.9
120	40	0.096	0.279	11.8	11.8
120	20	0.096	Too High	11.78	11.7

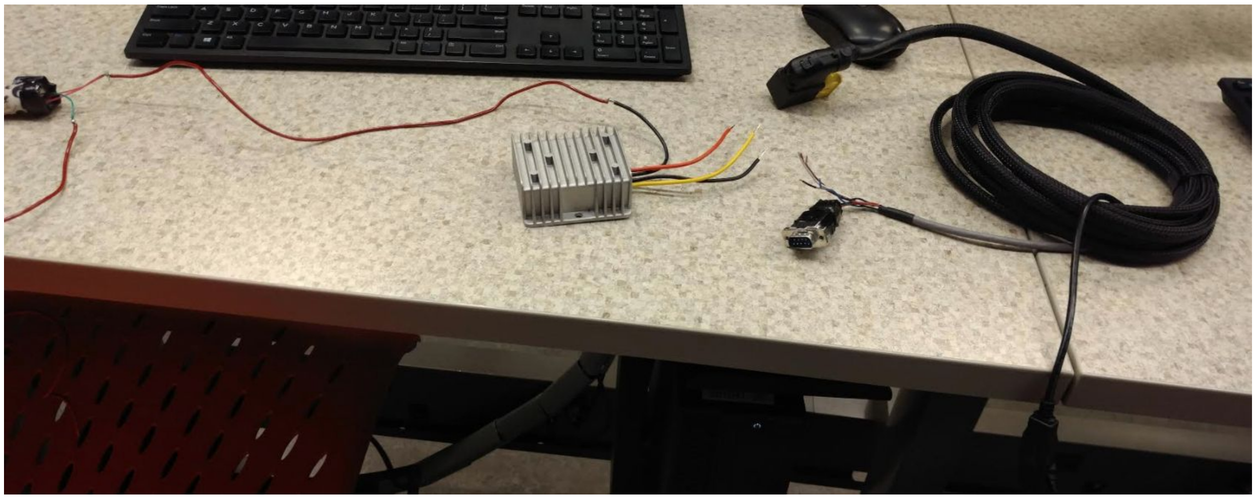
From our experiment we noticed that within the ranges of the values that our components will have we will have the appropriate currents applied. We also noticed that the load in one port does not affect the value of either the current or the voltage in another port. This means we will not have to worry one component failure affecting another component failing. These parameters helped us choose the right connectors with confidence for the devices. Below is the one we created for the PX2:



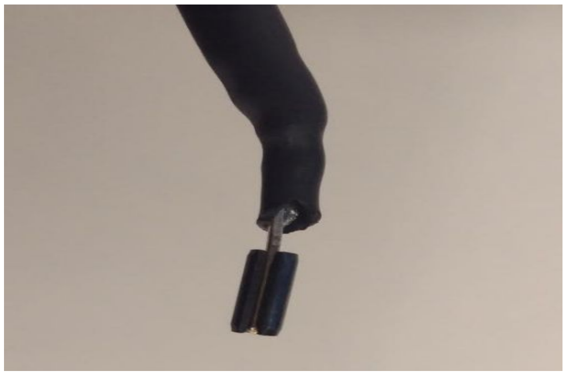
The Pins are below:

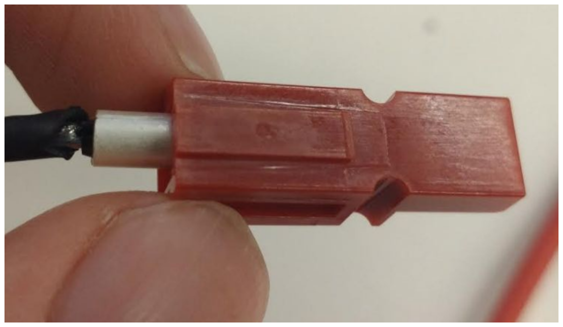
Pin	Description
1	12 Volts
2	12 Volts
3	12 Volts
4	12 Volts
5	Just floating (not attached to anything)
6	Ground
7	Ground
8	Ground
9	Ground
10	Ground

The red and black end is how the devices connect to the RigRunner. We connected several other cables with some of these and for the ESR and Lidar we also added a 12 to 24 volt converter as the one shown below:



The connectors clip on easily but must be soldered on to the connectors as shown below:





Placing your work in the context of:

• Related products

DSRC
DSRC (Dedicated Short Range Communications) is a two-way short distance wireless communications capability that permits high data transmission critical in communications based active safety applications. DSRC is commitmented for active safety communications contributes to safer driving. Vehicle safety applications that use vehicle-to-vehicle (V2V) and vehicle to infrastructure (V2I) communications need secure, wireless interface dependability in extreme weather conditions, and short time delays; all of which are facilitated by DSRC.

V2V applications utilizing DSRC could significantly decrease the chance of traffic accidents and give real time warning for drivers to imminent hazards. Since DSRC is featured for fast network acquisition, low latency, high reliability and safety, it could fully satisfy our requirements.

APP
https://turtler.io/news/location-sharing-in-google-facebook-and-twitter
Basically, this will use cellular data for transmission of location information. This method has many options; one option is Google maps location sharing which just needs a Google Maps

app. This app will drive the GPS of the cell phone, and then the location information will be sent and shared with other devices. Furthermore, there are some other similar apps which have the same properties, like Facebook Messenger or Twitter.

GPS Tracker
https://www.youtube.com/watch?v=bPFtLThcNNY
http://www.electroflip.com/products/gps-tracking-devices
This device can keep the location of your vehicle being sent directly to your phone. It can be set to sent at several different intervals throughout the day. The video is not very clear due to the highest rate at which the device can send its information. However, It’s pretty clear that it’s not meant to be a constant send. This is seen by the fact that this device stays idle for several hours or minutes and just periodically responds to movements of the car or location. It is really good when your car is stolen or when you want to know if your car has been moved. Another thing that distinguishes it from our device is that it works with the customers’ sim cards. This is similar to an idea we had earlier. We had to dismiss it because we believe that having the signal to find a satellite and inform user its location through that satellite is more prone to error than having two devices talking to each other directly.

GPS Tracker For Android
https://gpstrackerforandroid.com/
This application needs two cell phones with the android operating system. It gives by default the location of one of the phones to the other phone every minute. The rate of transmission can be changed; however the technical limitation of transmission speed will be at 10 Hz. This again runs into a similar problem as the GPS tracker above that depends on service. It also does not have a direct way to attach itself to the car and eventually we would like the vehicles to have their own built in tracking program instead of relying on someone having a phone.

• Related literature

https://www.allaboutcircuits.com/projects/gps-transmission-with-the-hc-12-transmitter/
The GPS allows users to accurately determine the location of objects on the Earth. The HC-12s can transmit the information from a GPS receiver with no additional programming or circuitry. GPS coordinates can be transmitted to remote locations with just a GPS receiver, an HC-12 transceiver, and a battery. Remotely transmitted coordinates would have to be received by another HC-12 transceiver and then processed with a microcontroller or computer.

Appendix I – Operation Manual

Raspberry PI 3 Model B

The RPi is set up to automatically execute the Python script when the device powers on. This has been set up by editing the rc.local file. Before powering on the device, it's required to have the GPS and XBee attached to the RPi before turning it on. There is a TTL to USB adapter which converts the TX, RX, VIN, and GND pins to a simple USB port. This is placed between the RPi and the GPS as described in the figure below. The GPS unit itself will also have two attachments connected to it. The external active antenna is plugged into the RF adapter cable. The RF adapter cable is then plugged directly into the GPS unit.

The XBee is plugged into the USB adapter which allows for the XBee to be plugged into one of the USB ports on the RPi. In order for the XBee to transmit, the monopole antenna will also need to be attached. Images for each adapter and device are included below.

Figure: Raspberry PI 3 model B

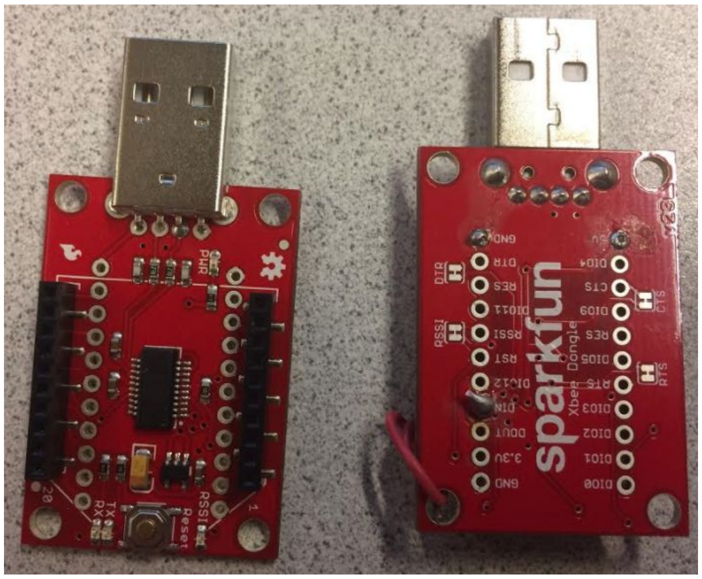


Figure: SparkFun XBee Explorer Dongles

Figure: Adafruit Ultimate GPS Breakout - Version 3

Figure: XBee Pro S3B

Figure: XBee monopole antenna extender

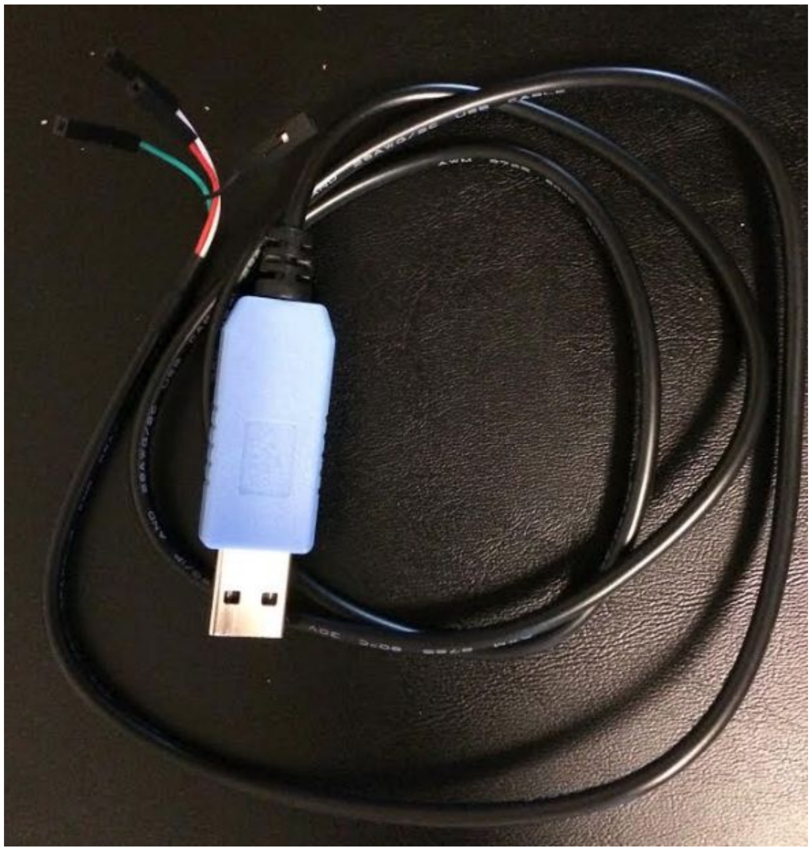


Figure: TTL to USB adapter

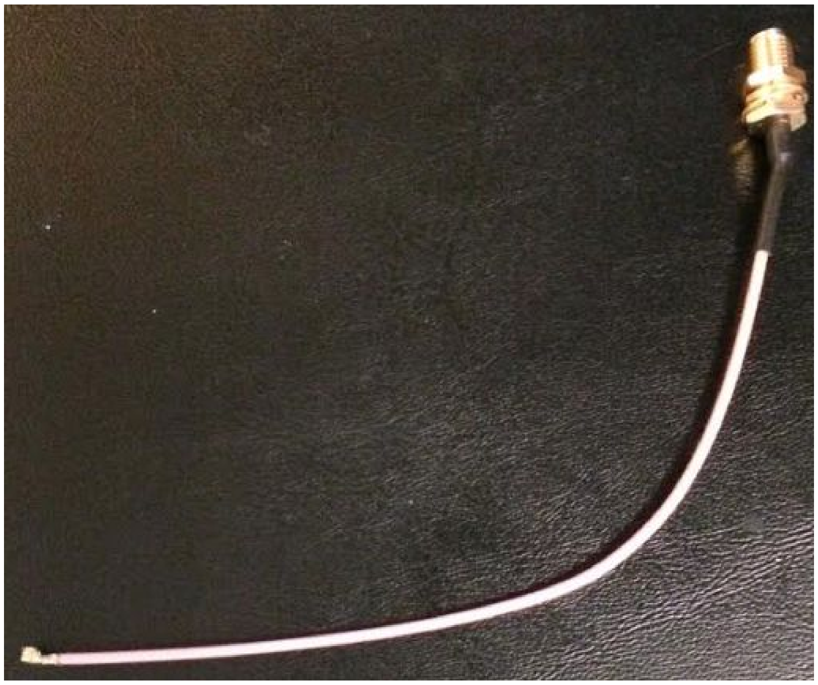


Figure: RF adapter cable

Figure: External active antenna

Figure: Monopole antenna

The two figures below shows the high-level overview of everything attached for the transmission side.

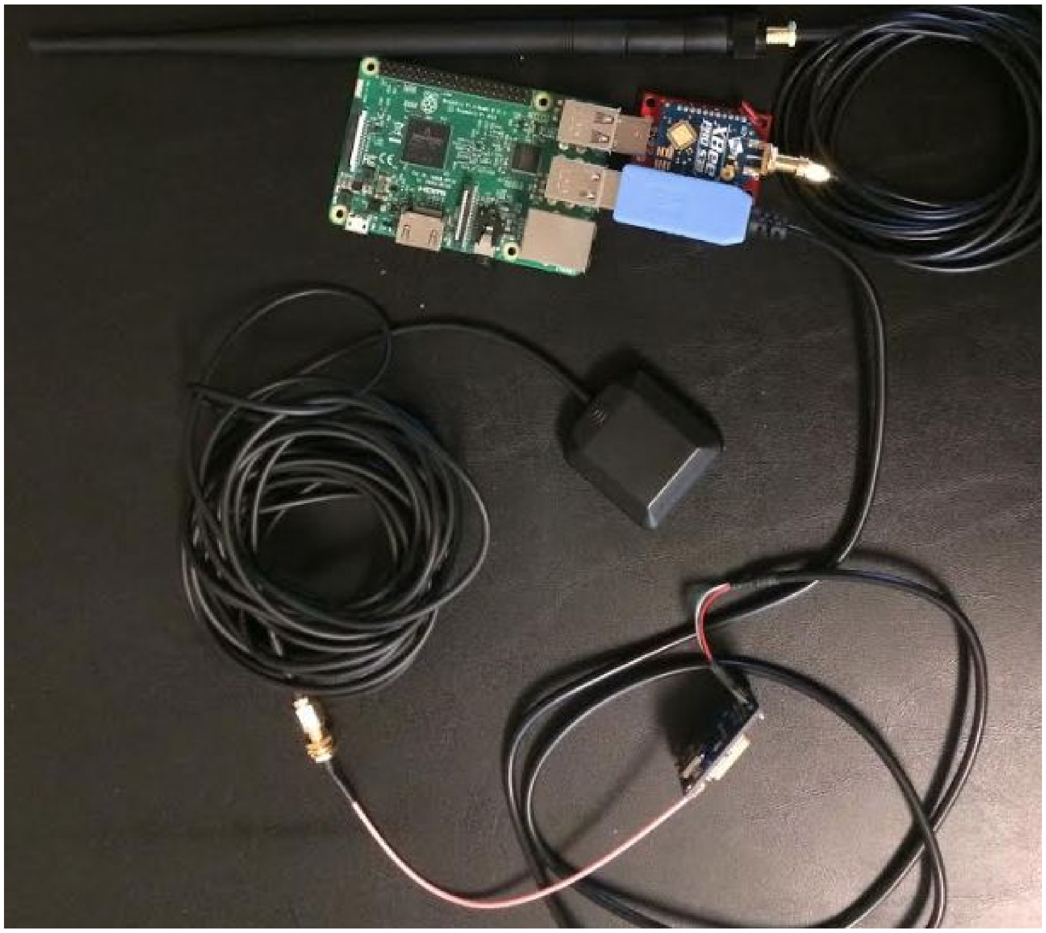


Figure: Overall setup - hardware

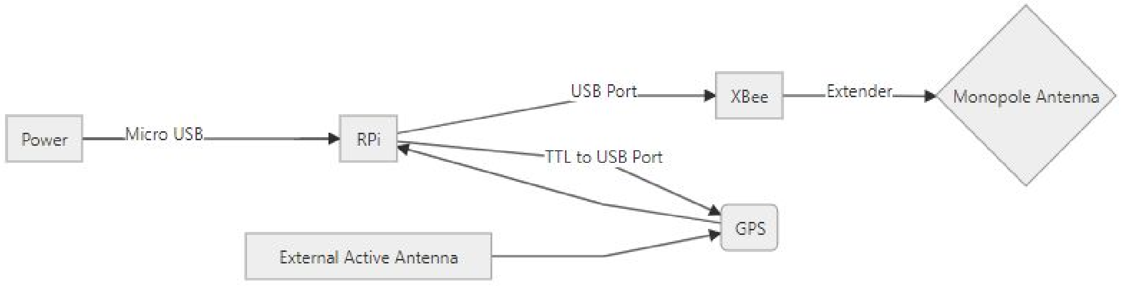


Figure: Overall setup - sketch

Dependencies

- Python 2.7 -> https://www.python.org/downloads/
- pySerial -> https://pythonhosted.org/pyserial/
- Adafruit_CircuitPython_GPS -> https://github.com/adafruit/Adafruit_CircuitPython_GPS

How-to run
In order to start transmitting the GPS data, the XBee and the GPS along with all of their attachments need to be attached to the RPi before turning it on. After turning on the RPi, the script will run automatically and start transmitting the data to be received by the XBee attached to the Nvidia PX2.

Nvidia Drive PX2

For this project, we were required to use ROS. We have created two nodes to run on the Nvidia Drive PX2. The first node publishes the data received from the GPS through a serial connection to the XBee. The received data is then published to the ROS master as a JSON formatted string. The second node that parses the data published to the ROS master for other teams to use the included data from the lead vehicle. This parsed data includes the latitude, longitude, and time-stamp from the GPS itself.

In order to setup the receiving side, the XBee will need to be attached to a SparkFun XBee Explorer Dongle which converts the connection into a single USB port. This allows it to be plugged directly into a USB port on the Nvidia Drive PX2. The XBee also needs the monopole antenna attached to it in order to received bytes from the XBee on the lead vehicle. Images of new devices have been attached below:

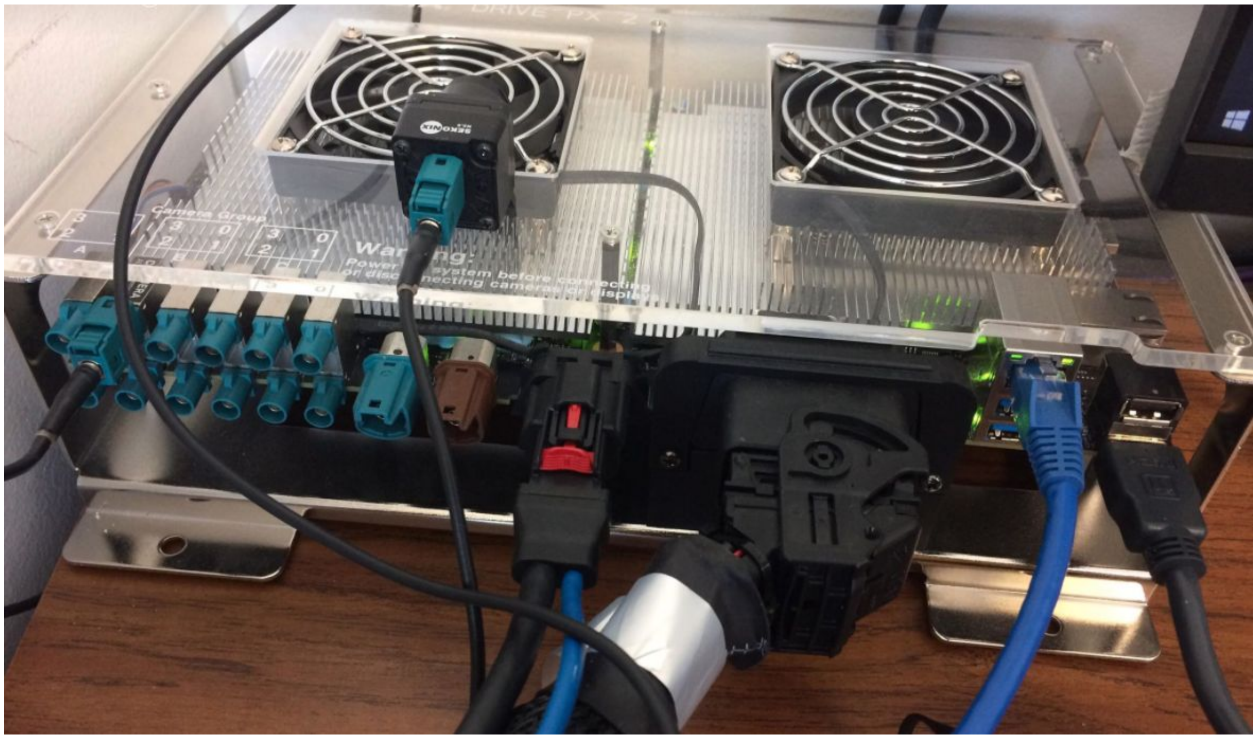


Figure: Nvidia Drive PX2

The two figures below shows the high-level overview of everything attached for the receiving side.

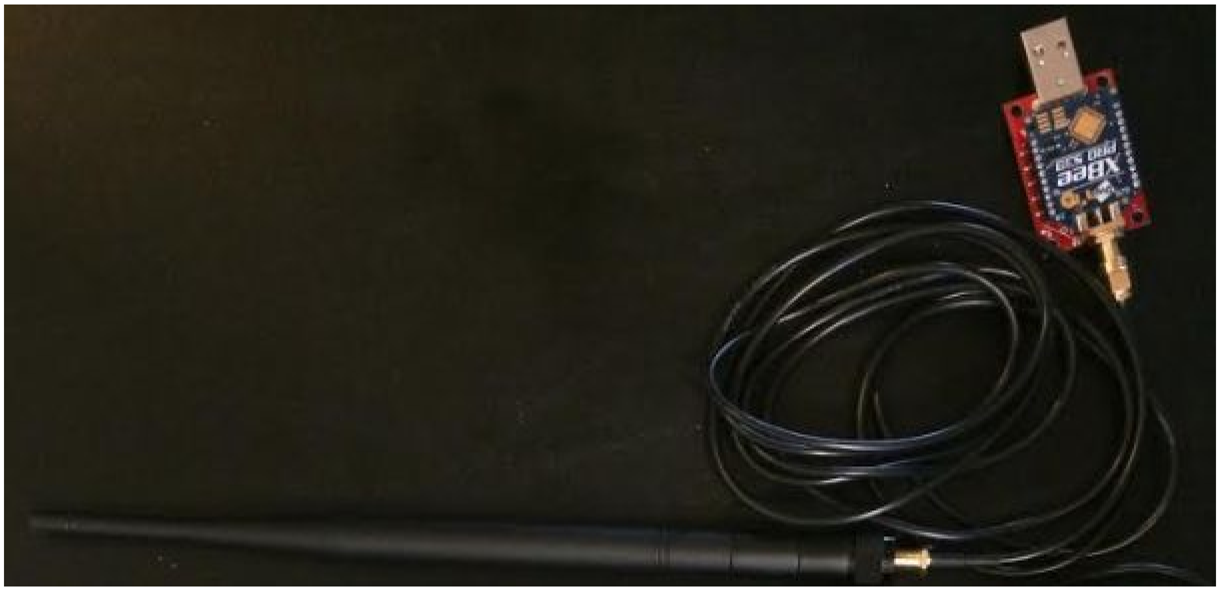


Figure: Overall setup - hardware

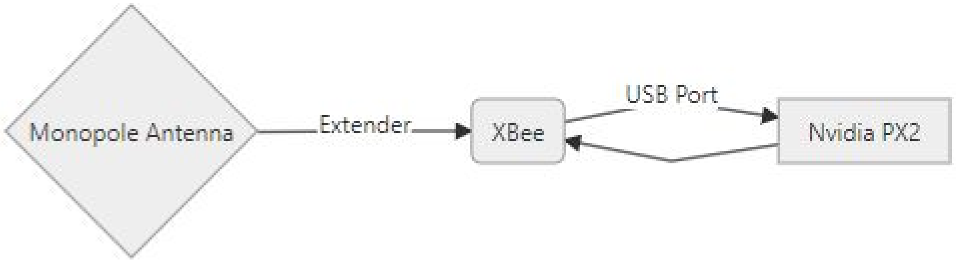


Figure: Overall setup - sketch

Dependencies

- Python 2.7 -> https://www.python.org/downloads/
- pySerial -> https://pythonhosted.org/pyserial/
- Ubuntu 16.04 -> https://www.ubuntu.com/download/desktop
- ROS Kinetic Kame -> http://wiki.ros.org/kinetic
    - std_msgs
    - rospy
    - python-rosinstall
    - python-rosinstall-generator
    - python-wstool
    - build-essential

How-to run
In order to run the ROS nodes, the commands below need to be entered. These commands also assume the ROS package created has been named ‘lead_gps’.

This is the first command which needs to be entered in the terminal to start the ROS service.

```
$ roscore
```

A new terminal will be required and the following command starts the publisher node which will publish any data retrieved from the XBee to the ROS master.

```
$ rosrun lead_gps gps_publisher.py
```

Optional: A subscriber node has been written for other teams to use for accessing the published data. This will start the subscriber.

```
$ rosrun lead_gps gps_subscriber.py
```

Software Licensing

Throughout this project, we have used several open-source libraries listed below. The table includes their name, version we used, license, and the URL to the repository.

Name	Version	License	URL
pySerial	3.4	BSD-3-Clause	https://github.com/pyserial/pyserial
Adafruit_CircuitPython_GPS	Mar 11, 2018	MIT	https://github.com/adafruit/Adafruit_CircuitPython_GPS

Appendix II: Alternative/ other initial versions of the design

1) The DSRC

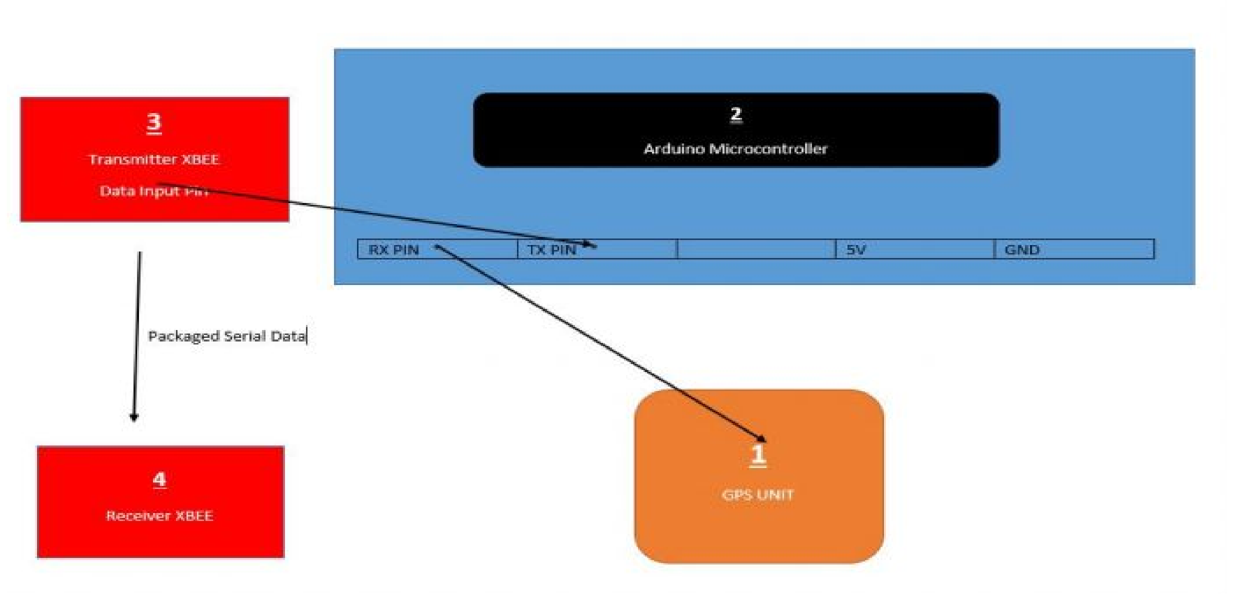
Our first idea was implementing a DSRC device into the following car. Since the DSRC device was the common transceiver for cars, we looked into certain devices that had the DSRC frequency of over 5 GHz. This would enable our transceiver system to have an operating range of over 1 mile. Our original approach to this method was having this device and hooking it up to the GPS where the serial data of the GPS would be sent to the lead car and potentially also receive GPS coordinate data from the lead car as well. Having this transmission distance seemed to be important to us at first and we invested some time into looking into buying the device. When we finally did some searching we realized that there were two problems: one could not simply order it and must have permission from the government and the second was that these things came at a hefty price. Our cheapest option was at nearly 800 USD which seemed to be far beyond our expected budget. This first step in development was not in vain; by looking into a distant range transceiver device, we were able to get a better idea for what must be done. Our job as the ECPE team was to create a reliable transmitter system that could exchange GPS serial data between the lead and following car.

2) The Mobile Wireless Data

After our unsuccessful attempt to purchase a DSRC transmitter, we looked at all sorts of alternatives ranging from routers to other smaller methods. One method we did look into was using our wireless mobile data as a way to receive and transmit serial data. The pros of this method was that it gave us a long range larger than the DSRC; the downside of this was that there would be delays in transmitting and receiving data. At first we thought that the delays would be negligible as long as we had the long range; however, after a meeting with our advisor, we realized that these delays could cause the two cars giving different coordinates from that actual position so we decided to scrap this method. They also depend on service due to their alternative route through a satellite compared to other transceivers that just talk directly to one another. By looking into the mobile data method, we were able to find how serial data is extracted from external devices. This knowledge would prove to be indispensable in all of our future attempts when we use different transceivers to extract the GPS' serial data on the following car. This attempt set up our first ground rule: we want accuracy over long range.

3) The Arduino & XBEE

This step into our first semester of senior design was the most important. For this process, we started to order parts and have more thorough methods of test and trial. After searching around for transceivers, we set our eyes on the XBEE device which would be our main device throughout the project. Since this was the first prototype transmission system we would test the GPS on, we were expecting to gain some valuable experience with implementing hardware. Our first prototype diagram can be shown below:



In the diagram, the GPS unit is connected to the receive pin of the arduino. From our experience gained in using mobile data, we learned that we can simply receive the serial code from the GPS and then the serial data would go to the Arduino Uno microcontroller. The microcontroller was used to program and format the incoming serial data into raw coordinates. This is where we ran into our first realization: the serial data has its own format for the device and must be organized properly in order for it to be decoded. Learning this, we revised our arduino code multiple times to fit the format of the GPS. The serial data format of the GPS is shown below:

PREAMBLE	BID	MID	LEN	DATA	CHECKSUM

This format for the GPS serial code had to be accommodated by the arduino, where we would have to write the code to decode each block in the serial format shown.

In the hardware aspect, we also learned a very important lesson in hardware compatibility: protocol. When hooking up the Arduino to the GPS, we used a pin to USB connection, where we would cut open the wire of a USB cable and have wire on one side with USB port on the other. Since the GPS used a USB port connection our method was connect the USB pins to the RX of the arduino and then hook it up to the GPS. This would result in giving us faulty data which we struggled to find the reason at first. Later we realised that the pins and USB use different protocols; which would then lead us to the final prototype of using a Raspberry Pi.

Appendix III: Other Considerations

To sum up with our project, we would like to remind our possible solutions of communication between lead vehicle and following vehicle. What we mentioned for solutions were on three ways in terms of using different tools to see how data is going. In first case we have tried to use two microcontrollers, data receiving from GPS and transmitting to the other was really unstable. Sometimes it gave data back from GPS but it was showing something wrong. In second case we have used Raspberry Pi on the lead vehicle equipped GPS. it gave us interesting solution because it has fixed the problem that we have had in first case. By running ROS that was being able to perform with Raspberry Pi, data communication was getting better than the first case. Also, we have attached XBee to transceiver part so that we were able to transmit data from GPS on lead vehicle to the following vehicle with wider range as well. The last option was to use original product that we have already bought. It was XBees which can receive and transmit data back between themselves. Actually it was really nice tools for communication between two objects but it could not provide GPS with suitable power source to run with PX2.
It was concluded by experiments we have done in two semester and we actually have been done several times of attempts to experiment how to source power through PX2 connected with RIGrunner which is located to underneath the passenger seat on following vehicle. The main problem for sourcing power that we have faced was to make sure all requirements from all sensors and PX2 that have different amount of current and voltage values are matched each other so that they should be powered by an electric car which is chargeable by battery. All attempts and experiments we have done are performed by Vermeer applied technology hub where is located on the way to go Iowa State University Research Park. Vishal who is our project manager also worked at Institute of Transportation right nearby Vermeer applied technology hub.

We would like to say one more thanks a lot because everytime when we need to test something for data transmission and figuring out power source come from RIGrunner, he always dealt with whether we could enter into there or not. It was not accepted without slashing a card that can be possessed by the faculties. Making a schedule with any faculties managed the hub was mandatory then group members who take cars gladly shared their own cars to visit there. In the beginning of second semester, we suddenly encountered unexpected situation that our project manager temporarily. He was staying his country due to some reasons for a while so that we could not make the process move forward at this time. However, we have catched up what we need for powering and data transmission so that we finally could wrap up with final process of our project.

It was really delightful that we have picked the topic about autonomous vehicle that is pretty sensational issue today. We hope and think it is possible that learnings and skills by knowledges how to communicate vehicle to vehicle wirelessly could be utilized later when it comes to do research and project related to self-driving in the jobs and graduate school.

Appendix IV: Code

Introduction

Inside the boxes below, is the program written to transmit the GPS data from the lead to the following vehicle. Below each box is description of that section of code as we progress through the entire program.

External Libraries

```
# https://github.com/adafruit/Adafruit_CircuitPython_GPS
import lib.adafruit_gps as adafruit_gps
```

This is a simple library that we discovered online that assisted us with parsing NMEA data sentences from a serial GPS. Apart from the data struct, this library has a send_command method, update, and has_fix property. All will be described in further detail upon use in following program.

Native Libraries

```
import serial
import time
import serial.tools.list_ports as port_info
import traceback
import subprocess
```

Python built in libraries and their project specific uses:

- Serial - Used for serial communication with XBees and GPS.
- Time - Used for manipulation of time objects.
- Serial.tools.list_ports - Used to pull the information from the devices plugged into the RPi’s USB ports.
- Traceback - Reports errors back to the system, needed because of try-catch loop in program.
- Subprocess - Used for checking the baud rate of the connected devices. This is done by calling “stty -F <port name>” programmatically to check the output of the port.

Gather Component Information

```
all_ports = port_info.comports()
XBee_hwaddr = "0403:6015"
GPS_hwaddr = "10C4:EA60"
XBee_Serial = None
GPS_Serial = None
```

The all_ports variable stores all of the USB connected device data. The XBee and GPS hardware addresses are hardcoded to help with auto detection mentioned in the next segment. The serial variables are there to store the serial connections to each device after being initialized in the next section.

Finding and Initializing Devices

```
# Iterate through each port and find the XBee and GPS with auto detection
for usb_port in all_ports:
    if "VID:PID" in usb_port.hwid:
        port_hwid = usb_port.hwid.split(" ")[1].split("=")[1]
        port_name = usb_port.device

        # Check to see what baudrate the GPS is currently running
        port_info = subprocess.check_output(["stty", "-F", port_name])
        port_baud = int(port_info.split(";")[0].split(" ")[1])

        if port_hwid == XBee_hwaddr:
            # Create the serial connection to the XBee
            XBee_Serial = serial.Serial(port=port_name, baudrate=port_baud)

            # Show what settings have been detected - used for debugging
            print("XBee:\t" + str(port_name) + " - " + str(port_baud) + " baudrate")
        elif port_hwid == GPS_hwaddr:
            # Create the serial connection to the GPS
            GPS_Serial = serial.Serial(port=port_name, baudrate=port_baud)

            # Show what settings have been detected - used for debugging
            print("GPS:\t" + str(port_name) + " - " + str(port_baud) + " baudrate")

# Ensure the GPS unit was found
if not GPS_Serial:
    print("Error: GPS not found")
    quit()

# Ensure the XBee was found
if XBee_Serial is None:
    print("Error: XBee not found")
    quit()

# Creating a GPS module using the repository linked above
gps_inst = adafruit_gps.GPS(GPS_Serial)
```

```
print("Initializing GPS")
```

Iterates through the four ports of the RPi to find the GPS and XBee by matching the hardware addresses to components listed earlier. After finding a connected device, the baud rate is found and the XBee or GPS serial object is created with found baud rate. This has to be done to ensure that commands sent to the devices are at the correct rates. If either of the devices are not found a message will print which, otherwise the program continues on.

Initialize The GPS

```
# Ensure the GPS is running at baudrate of 115200
gps_inst.send_command("PMTK251,115200")
GPS_Serial.baudrate = 115200

# Ensure the GPS to RMCONLY mode
gps_inst.send_command("PMTK314,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0")

# Ensure the GPS to 10Hz mode
gps_inst.send_command("PMTK220,100")

print("GPS Initialized")
```

In this section, we are setting the GPS to have an 115200 baud rate, RMC Only mode is active, and that the GPS is outputting data at ten hertz. Since we are only worried about our position data; latitude, longitude, and time, RMC is the smallest data packet with the required information. This helps cut down the weight of a transmission. The send_command function, basically formats the parameters and sends it to the GPS over a serial connection.

Parsing and Transmission

```
# Enter infinite loop to constantly parse GPS output
try:
    while XBee_Serial.isOpen() and GPS_Serial.isOpen():
        # Check for any updates from the GPS
        gps_inst.update()

        # Check and ensure the GPS has a fix
        if not gps_inst.has_fix:
            print("GPS has no fix")
            time.sleep(1)
        else:
            # Extract the time components and format it into a human-readable string
            gps_hour = gps_inst.timestamp_utc.hour
            gps_min = gps_inst.timestamp_utc.minute
            gps_sec = gps_inst.timestamp_utc.second
            gps_millisec = gps_inst.timestamp_utc.microsecond / 1000
            gps_time = str(gps_hour) + ":" + str(gps_min) + ":" + str(gps_sec) + ":" +
str(gps_millisec)

            # Create the dictionary including the timestamp, latitude, and longitude to be sent over
            data_dict = {'time': gps_time, 'lat': gps_inst.latitude, 'lon': gps_inst.longitude}

            # Convert the dictionary into a string for transmission
            str_eq = str(data_dict)

            # Transmit the string formatted dictionary and a byte after including the size to ensure
            # nothing was lost during the transmission
            XBee_Serial.write(str_eq + "\n")
            XBee_Serial.write(chr(len(str_eq)))
            print("Sent: " + str_eq)
except:
    traceback.print_exc()
finally:
    XBee_Serial.close()
    GPS_Serial.close()
    print("Program finished")
```

The update method provided by the external library returns data as it is received. If the GPS hasn't processed any new data, this method will return false. The has_fix property works almost the same way except that it returns false if it can't find any position data. Once the GPS has a fix, the time componentes get extracted and reassembled into a human-readable format. The external library did not provide milliseconds variable in its GPS structure, so we had to commute using the provided microseconds. The data is then written to the XBee using serial communications for transmission.