Bloom Bus Tracking Project

Daniel Pany
danielcpany@gmail.com

Riorden D. Weber
riodweber@gmail.com

Bloomsburg University of Pennsylvania
Bloomsburg, PA 17685

13 December 2016

Abstract

This paper is for the documentation of a prototype web app which may later be released as a mobile and desktop application which tracks Bloomsburg University of Pennsylvania's campus busses. The project's timeframe occurred over the Fall Semester of 2016, a total of thirteen weeks. The project was a success. We urge this prototype to be pursued as a project for the University's COMPSCI 480 class.

Thesis: Create a system that would show a live map of a vehicle driving around the Bloomsburg University campus.

Keywords: bus, tracking, GPS, Bloomsburg University, node, xbee, arduino, computer science

1 Introduction

As reflected anywhere in modern day society, most people would prefer the world to work as efficiently, seamlessly, and quickly as possible. This is no exception when it comes to the students of Bloomsburg University. The University's campus is divided into two parts: upper campus and lower campus. While lower campus contains all of the academic halls and classrooms, upper campus is home to many of the University's upperclassmen. In order to travel between the two, you could walk the near mile of sidewalk with an exhausting hill, or do what most students would, and simply take the bus. This works fine a majority of the time, however, it's common for the busses to feel like they're be running late, or not at all. So from a student's perspective it can be an unpleasant experience trying to catch the bus consistently. Then, there's the frustration of getting to the bus stop right as it is pulls away.

It was this frustration that sparked an idea: it would be a great feature if the University could track where the buses would be at any given time. This feature would allow the students to plan for the bus better, effectively saving them time, reducing their anxiety, and minimizing their exposure to inclement weather.

After missing the bus too many times, a solution was theorized. This concept was further explored in the Spring of 2015 among Bloomsburg's Computer Science ACM (Association of Computing Machinery) Chapter.

1.1 Original Concept

The concept was to have little standardized devices, or modules, be housed inside each of the busses. These devices would consist of a GPS unit and a transmitter. The GPS unit would receive the bus's GPS coordinates, pass them on to the transmitter, which would then send them to a central server, listening for incoming messages (Figure 1). The server would store the latest coordinates of each of the busses, and host a web app for users to connect to. The web app would then display a map of the University with each of the busses at their respective locations in real time. The app would be accessible from a computer or mobile device (Figures 2 & 3).

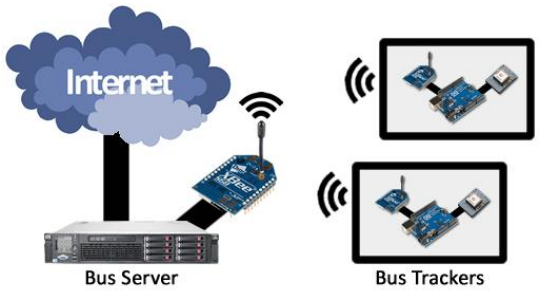


Figure 1
Bus trackers communicating with the server via XBees.

Figure 2
A conceptualization of how the Bus Tracker would appear on computers.

Figure 3
A conceptualization of how the Bus Tracker would appear on a phone.

1.2 Prototype History

During the summer of 2015, Daniel Pany, a Digital Forensics major and Computer Science minor, took the idea under his wing and designed a primitive prototype. The prototype involved the use of a portable module and server. The module consisted of an Arduino microcontroller, Xbee radio, GPS unit, and a shoebox, with everything being held together with sticky tack. After several headaches with initial configurations and software bugs, the prototype ended up working.

Pany took the module on several walks around his neighborhood, while also successfully transmitting the GPS coordinates of his location to a server. Upon return, the coordinates were modified into a Google Earth document. This allowed the coordinates to be viewed as a full path (Figure 4). While the path's sharp turns and missing sections showed imperfections, the prototype worked as intended.

In the Fall of 2015, the Bloomsburg Computer Science ACM chapter presented this prototype to Dr. Curt Jones, the head of the Mathematical and Digital Science Departments at Bloomsburg University.

While interested, Dr. jones asked for a further proof of concept. He asked to see a live demo of a map showing a vehicle driving around campus. This ended up being the goal of our project.

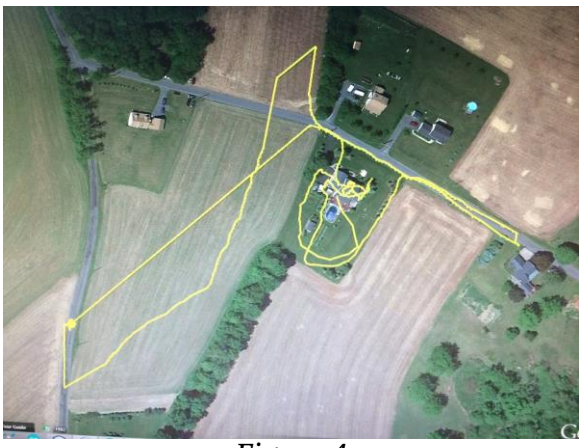


Figure 4
Coordinates from the first prototype displayed in Google Earth.

1.3 COMPSCI 491 - "Special Topics"

The project would be given the opportunity of becoming a school funded project. In the Spring of 2016, it was announced that the class "Special Topics" would be offered for students to choose their own topic of study. This class would be held under the direction and guidance of Dr. Diane Barrett, a Digital Forensics professor.

In the beginning of the 2016 Fall semester, Rio Weber, a Digital Forensics major and Computer Science Minor, joined the project. Weber is an officer of ACM, and helped conceive the original concept when it was first being developed. Together, Daniel Pany and Rio Weber registered for the COMPSCI 491 (Special Topics) class.

1.4 Project Scope

This project was given the duration of the 2016 Fall semester to be completed. The goal, as requested by Dr. Jones, was to further the existing prototype to the point where we could see a live map of a vehicle driving around campus.

2 Literature Review

In preparation of our project, we felt it would be best to have statistics to back up our claims that students wanted a bus tracking system and that it would be beneficial to the University.

In the Spring of 2016, a group led by Wade Cooper, a Business and Finance major, completed a study on upper campus bus statistics. In addition to surveying the students, his group also surveyed how accurately the busses follow the expected time of arrival. The study concluded a few things, stating "Bloomsburg University students want a GPS tracking system for the buses," "the benefits outweigh the costs," and "a GPS tracking system can benefit the students, school, and bus drivers."

Nearby Bloomsburg is West Chester University of Pennsylvania, and they have an already working, yet proprietary bus tracking system. They use a service called DoubleMap, which shows users the movement of each bus in real time. It allows users to distinguish bus routes by color coding them. Finally, it lets the users know about delays, cancellations, and holiday services. This allows the students to plan better, which saves them time, reduces their anxiety, and minimizes their exposure to lousy weather.

3 Methodology

It was very important to us that everything we worked on would become open-source. Seeing as this project may be taken under the wing of some promising students in COMPSCI 480, we have attempted to explain our decisions and process for how we went about creating this project.

3.1 Divide and Conquer

Our overall strategy to develop this proof of concept prototype was to compartmentalize it. From the start, we knew we had to split the project into two distinct overall parts, separate but equal. The success of the project would be determined on the proper integration of these two parts. Pany would lead the configuration and development of the Bus Modules, while Weber would create the server and design its interface.

3.2 Part One - The Bus Modules

Due to prior research before this semester began, we already had a primitive bus tracking module prototype to work from (Figure 5). This module consists of three core parts: a GPS receiver, a microcontroller, and a radio transmitter. For the GPS receiver, we used an Adafruit Ultimate GPS Breakout v3. For the controller, we used an Arduino Uno microcontroller. For the radio transmitter, we used an XBee-Pro S2C ZigBee Radio with a built-in wire antenna attached to an Adafruit XBee Adapter Kit, which allowed us to interact with the XBee through USB or via Arduino. Also, we used a USB battery block to power the module, but the main source of power is intended to be the buses themselves through a cigarette lighter receptacle to USB converter.

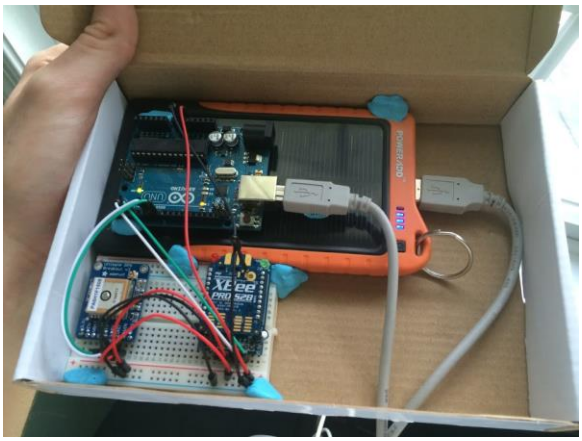


Figure 5
The original prototype we modeled our Bus Modules after.

For our continued research this semester, the University purchased additional components for a second bus module to be built and tested. The additionally ordered parts consisted of an Arduino Uno, an Adafruit Ultimate GPS Breakout v3, an XBee-Pro S2C ZigBee Radio, and an Adafruit XBee Adapter Kit. Upon their arrival, we soldered the parts together in Hartline Science Center's soldering lab with special permission from Dr. Nathaniel Greene, a Physics professor. After we had our parts soldered and ready for use, we started configuring them.

3.2.1 XBee & ZigBee

To clarify, an XBee Radio, commonly referred to as just an XBee, is a small physical device that can work under many different networking protocols. The term Zigbee refers to a commonly used XBee networking protocol. This protocol comes with many benefits; a few of which being the long distance between radios, low power usage, and self-healing mesh networking capabilities.

XBees have many different ZigBee firmwares to choose from, and it was essential for us to understand the main ones for us to know which ones to

choose. The three main firmwares, or "functions," are Coordinator, Router, and EndPoint. The Coordinator is the heart of an XBee ZigBee network. Without it, a network will fail to form. Routers and EndPoints connect to a Coordinator, and are able to communicate with each other. There can only be one coordinator in a network, while there can be many Routers and EndPoints. EndPoints are used only to send data to Routers or Coordinators. Routers act as both EndPoints and relays to other Routers and the Coordinator. It's also important to note that they act as EndPoints and relays at the same time. It is because of this relay functionality that the XBee ZigBee mesh networking topology is capable of being autonomously self-healing. If one Router acting as a relay goes down, any other Router can act in its place.

These XBee ZigBee firmwares are capable of being put into AT mode, also known as "Transparent Mode," or API mode. Data sent to an XBee in AT mode is immediately forwarded to a preconfigured XBee destination address. This means that XBees in AT mode can only work in pairs. API mode, which stands for "Application Programming Interface," is used to send network frames/packets with common network protocol features. If Routers are not put into API mode, the self-healing network capabilities described earlier will not function.

We chose the XBee-Pro S2C ZigBee model particularly because they were the most cost-effective model. They advertised a full mile of range between XBees when communicating outdoors, and that is all we needed for our proof of concept prototype. This model of XBee came with pins that were too small and closely compacted to work in a breadboard-style setting. To accommodate for this, we equipped each XBee with an Adafruit XBee Adapter (Figure 6), which is perfect for working with an Arduino Uno, as well as a CP2102 cable.

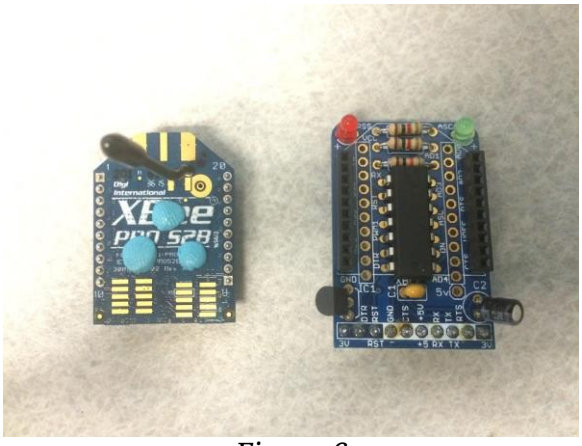


Figure 6
An XBee-Pro on the left, and the XBee Adapter on the right.

To communicate with an XBee for firmware changes and configurations, we resorted to using a CP2102 cable. In order to use the cable on any computer, the appropriate CP2102 drivers must be installed. After many frustrations with using a typical FTDI cable and attempting to use the Arduino Uno as a conduit, we found that these communication methods for configuring the XBee firmware were unreliable, and should not be used in place of a CP2102 cable.

We used the XBee configuration software, XCTU, to change each XBee's firmware and configurations. Since we had three XBees to work with, we set two XBees to have the Router API firmware functionality, and one XBee to be the Coordinator. A picture of general XBee configurations used in this project can be seen in the Appendix section 8.2.
Since there would be no way of identifying an XBee's firmware by looking at it, we labeled each one with dabs of sticky tack that corresponded to a key we created (Figure 7). We were able to test and troubleshoot sending

messages from Router to Coordinator by using XCTU's Frame Generator and Frame Interpreter features.

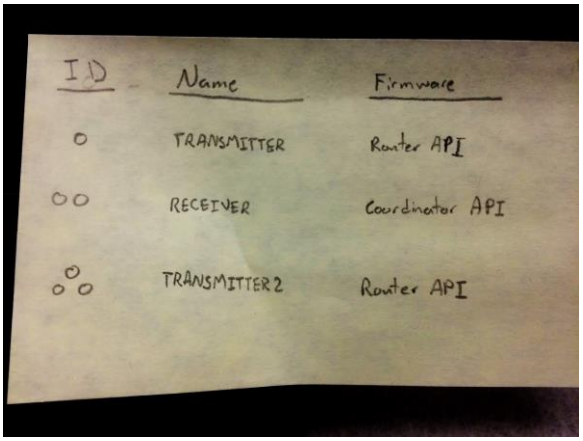


Figure 7
Our key for identifying each XBee by drops of sticky tack.

In order to research the actual distance that two XBees could be away from each other and still transmit traffic, we ran tests at a few different places. The first test we tried was in the building we were developing the prototype, McCormick. We also ran tests on the University Quad. Results on this can be found in the Findings section of this report.

3.2.2 "Ultimate GPS Breakout"

The Adafruit Ultimate GPS Breakout v3 should only be thought of as a GPS coordinates receiver, and not also a transmitter. If it were also a transmitter, then we wouldn't have to worry about sending the coordinates to a server using XBees. In truth however, this GPS unit does not "receive" GPS coordinates. It triangulates where in the world it is by listening to the signals of at least three of the many GPS satellites orbiting the Earth, constantly transmitting their locations. Using more than three satellites and advanced calculations, this unit can calculate its altitude, speed, and other interesting statistics.

These GPS units are small (Figure 8), and use minimal energy. They come with a built-in antenna, but it is possible to upgrade them with a larger external antenna. The unit's attached pins are perfect for working with an Arduino in a breadboard.

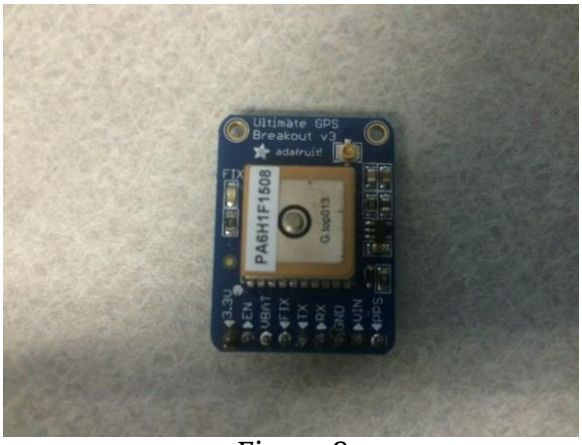


Figure 8
An Ultimate GPS Breakout.

In order to research the accuracy of the GPS units, we ran tests at a few different places. We were constantly testing the unit's accuracy by simply walking the module around the University Quad. We also tested it by taking a bus ride and logging generated coordinates. Results on this can be found in the Findings section of this report.

3.2.3 Arduino Uno

An Arduino Uno is a microcontroller which can be used to act as the brains between the XBee and the GPS unit. It is the Arduino which interacts with both of them, and allows the GPS coordinates from the Ultimate GPS unit to be sent to the XBee Router, and then sent off to the server (Figures 9 & 10).

By using the Ultimate GPS Arduino library, and connecting the appropriate communication and power pins by wire, we were able to interact with and receive coordinates from the GPS unit. We modified a pre-made example of Arduino-Ultimate GPS code to only gather the necessary information in the format we wanted.

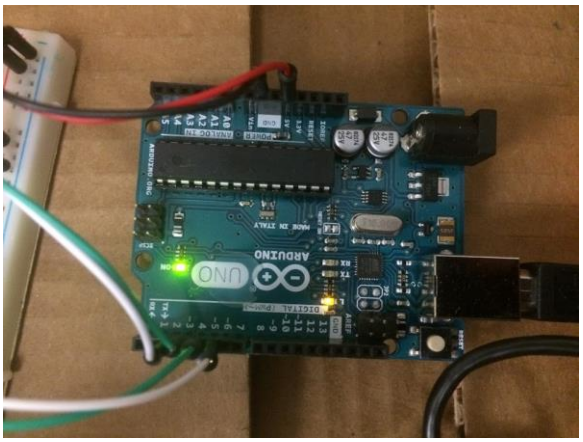


Figure 9
A close-up of an Arduino Uno as used in the current working prototype.

Since we have attached adapters to our XBees, we are able to connect them to the Arduino and tell them to send certain data to the server. In order to interact with our Router API configured XBees, we need to send our message in ZigBee protocol. To do this, we had to reverse engineer the format by looking at and understanding ZigBee packets. Then, we were able to construct a ZigBee frame generator, which takes any message we give it as input and returns a ZigBee frame, ready to be sent to the XBee for transmission. We included this code in the final Arduino sketch.

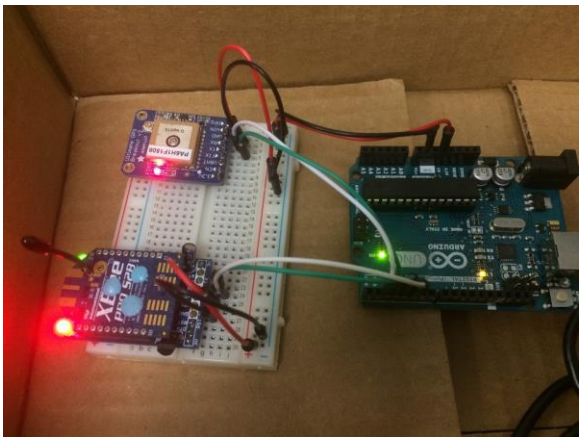


Figure 10
The Arduino Uno interacting with the XBee and the GPS unit.

It is possible to use an Arduino as a conduit for XCTU to interact with the XBees directly, as if the XBee was being connected to a computer via FTDI cable. As discussed earlier, this should not be used as the means for firmware and configuration changes, but can and should be used to troubleshoot sending and receiving packets.

3.3 Part Two - The Server

For a project like this, the purpose of a server is very well defined. The server is to act as the portal that synchronizes the app. It is where the user is served an interface, and where the data is parsed and stored. To handle all this, a well-known server "stack" was used. A server stack is a set of technologies that are known to work well together and meet industry standards. There are many different "stacks" but for this project we choose the "MEAN stack."

The MEAN stack is an acronym which stands for MongoDB, Express, Angular, and Node. This stack was used because it met all necessary requirements.

The rest of section 3.3 will be a breakdown of each technology, what they do, and why they were chosen.

3.3.1 Node

```
30    "dependencies": {
31      "body-parser": "^1.15.2",
32      "connect-mongo": "^1.3.2",
33      "express": "^4.14.0",
34      "express-session": "^1.14.1",
35      "firebase": "^3.5.1",
36      "history": "^4.4.0",
37      "mongodb": "^2.2.9",
38      "mongoose": "^4.6.0",
39      "react": "^15.3.2",
40      "react-dom": "^15.3.2",
41      "react-router": "^3.0.0",
42      "serialport": "^4.0.1",
43      "xbee-api": "^0.4.3"
44    },
45    "devDependencies": {
46      "babel-core": "^6.18.2",
47      "babel-loader": "^6.2.7",
48      "babel-plugin-add-module-exports": "^0.2.1",
49      "babel-plugin-react-html-attrs": "^2.0.0",
50      "babel-plugin-transform-class-properties": "^6.18.0",
51      "babel-plugin-transform-decorators-legacy": "^1.3.4",
52      "babel-preset-es2015": "^6.18.0",
53      "babel-preset-react": "^6.16.0",
54      "babel-preset-stage-0": "^6.16.0",
55      "webpack": "^1.13.3"
56    }
```

Figure 11
All Node Module dependencies. - package.json

The MEAN stack runs on node. Node is a JavaScript runtime built on Google Chrome's V8 engine. Node is a non-blocking input/output model that makes it extremely lightweight and efficient. Because of Node's non-blocking input/output model, Node can handle multiple connections and keep them alive without much processing. This was the most important reason for choosing Node. The project requires a constant real-time up update. This is necessary because in order to get a live feed of the university busses GPS location, it must be in real time. In order for the feed to be live, the server must coordinate a simultaneous connection from the database, to the user at all times. This alone requires two connections, and as more users are on the site, the more connections to the server must keep alive. This is precisely the reason why Node was the best choice for running the server. Node is capable of maintaining many connections with ease.
Node also has the largest ecosystem of open source libraries in the world. To see all the libraries used for this project see (Figure 11). The most important packaged library used for project was Express. This is because express is what turns the computer into a web server.

3.3.2 Express

```
1   var express = require('express');
```

Figure 12
Express instance creation code. - server.js

While Node is the engine that is used to power the server, something must be used to serve the site to the client. This is where Express comes in. Express is the web framework that serves the HTML to the client. HTML stands for Hypertext Markup Language, which is the language of the internet. Every time a browser makes a request to a server for a website, the server responds back with HTML. It is Express's job to handle these requests and execute the proper modules.

```
paniel:~/workspace (master) $ node server.js real
eth0 172.17.32.97
SERVER RUNNING... PORT: 8080
```

Figure 13
Express server running on port 8080

The choice to use Express as the framework for the app was less strategic than the logic that went into choosing Node. Express was chosen because it is part of the MEAN stack technologies. It has been tested and optimized to be used with Node and is also the most common HTTP server to be used in tandem with Node.

Express is also known for having a myriad of HTTP utilities and middleware. In the project quite a number of them were used.

3.3.3 Database

In the original architecture for the project, MongoDB was to be the database that would hold the data of the bus GPS coordinates. MongoDB is part of the MEAN webstack, so it is well tested and works well with Node because MongoDB stores data in JSON files. JSON* stands for JavaScript Object Notation, and since Node uses JavaScript, the combination is seamless.

* to understand what JSON format looks like, see Figure 11.

```
1  module.exports = function(server_IP) {
2      return { 'url': 'mongodb://' + server_IP };
3  };
```

Figure 14
Connection to MongoDB. - database.js

While the Mongo database (shown in Figure 14) is implemented, the use of MongoDB has not been. For this project the database needs to have realtime read and write capabilities, this in itself would be a massive project, one for which there was no time. Since the purpose of the project was to establish a proof of concept, rather than create a local version, concentration was place elsewhere where it would be more useful.

Since the requirement for a realtime database still exists, an excellent alternative was used called Firebase.

3.3.4 Firebase

Firebase is also a JSON database that is provided and maintained by Google. It is a real-time database and backend as a service. It provides the project an API that allows the application data to be synchronized to multiple clients at once. Google Firebase was a perfect fit for this project, however, Firebase does have its limitations. This project only uses Firebase's free version. The free version only allows 100 simultaneous connections. Also, only 1GB of storage is provided.

For this project these limitations was not an issue as the project is still in alpha. For the project to proceed to beta phase, a internally created version of Firebase must be constructed and implemented.

```
41
42  firebase.initializeApp({
43     serviceAccount: "server/BloomBus-0096d2641a16.json",
44     databaseURL: "https://bloombus-68ea7.firebaseio.com",
45     databaseAuthVariableOverride: {
46        uid: "my-service-worker"
47     }
48  });
49
50  // Creating Database Reference
51  var db = firebase.database();
52  var bus_dataRef = db.ref("/bus_data");
53
```

Figure 15
Firebase authentication key and data node reference. - xbee-api.js

The implementation of Firebase is straightforward. All that is required is to install the module to Node, then create a Firebase reference point to a data node. As seen in (Figure 15) the starting node is "bus_data." All data for the GPS bus coordinates are appended to this node, making it easily accessible to the client.

3.3.5 MVC

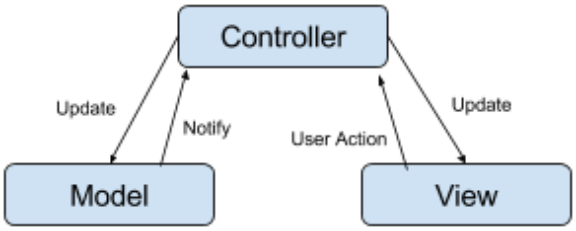


Figure 16
MVC structure

MVC stands for model view controller. It is a software design pattern for creating web applications. It is a web architecture that isolates the application logic from the user interface. This creates a faster and more seamless presentation.

MVC consists of three parts.
The model, which is the lowest level and is responsible for maintaining data. The

view, which is responsible for displaying the data to the user and allowing the user to take actions on said data. And lastly, the controller, which controls the interactions between the model and view. This is what allows the view to change, presenting new data to the user, or in this case, updating the position of the bus on the map. There are many types of MVC's and the MEAN stack suggests that Angular be used. However for this project React.js was chosen to be the MVC. React.js is a front-end JavaScript library for rendering data as HTML that is maintained by Facebook.

Initially React.js was chosen over Angular.js for no other reason than experimentation. However as time went on it became more clear that React was the right choice. It is important to note that either Angular or React would have sufficed for this project, however React was the better choice in the end because React also allows for transpiling to React Native, which essentially allows a webapp (browser app) to be turned into a native app (iPhone app). React was also chosen because of its growing influence in the web. Weber also had experience with Angular and was quite adept in using it, and wanted to compare the two.

3.3.6 Ratchet

```
23
24    <!-- Ratchet CSS -->
25    <link rel="stylesheet" href="ratchet/css/ratchet.min.css">
26    <link rel="stylesheet" href="ratchet/css/ratchet-theme-ios.min.css">
27
```

Figure 17
Ratchet includes for CSS and Javascript in HTML - index.html

The very last part to the server side of this project was to pick a front-end framework. There were two options. Option one was to use Twitter's Bootstrap, option two was to use Ratchet. For this project Ratchet was the obvious choice. This is because Ratchet is specifically designed for creating "native-looking" web apps, and this project will hopefully be transpiled into a native app when completed.

Ratchet allowed easy HTML page creation, the end result of which is displayed in (Figure 18).

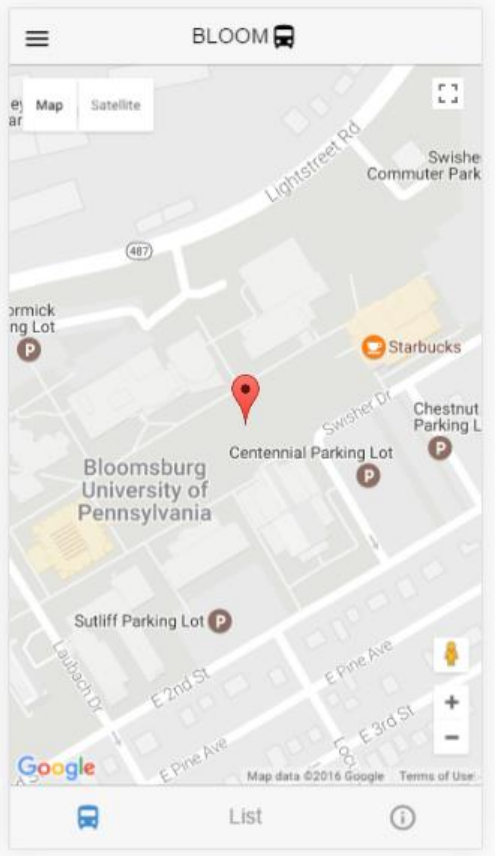


Figure 18
The App web interface.
See more in the appendix.

3.3.7 Google Maps

This may be the most important as well as most complicated part of the front-end application.

Google Maps is the medium that allows the location of the GPS modules to be presented. Incorporating Google Maps into the app was a challenge. This is because Google Maps requires exact

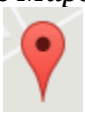
dimensions (Figure 19), and the display to be visible, and an instance of it to be created, all on the execution of the app.

```
47
48  #map {
49      width: 100%;
50      position: absolute;
51      left: 0;
52      top: 45px;
53      bottom: 51px;
54      overflow: hidden;
55  }
```

Figure 19
Google Maps element dimension style - custom.css

The logic that runs Google Maps is provided by Google which can be found at maps.googleapis.com/maps/api/js.
After this file is included Google Maps needs an "instance" script. The majority of front-end logic for this app is located in this script. It is here that the app drops a pin, connects to Firebase), and updates the location of the pin based on the latest coordinates received (Figure 20).

Google Maps "pin"



Firebase instance

```
var database = firebase.database()
.ref('bus_data/' + getNewTime());
```

Figure 20

3.3.8 Cloud9

Now that all the technologies used for the project have been selected, there must be a place to develop and host it. Cloud9 was selected as the only contender to be the platform our project would be built and housed on. Cloud9 IDE is an online integrated development environment. It also allows users to simultaneously change and write code together on the same file, much like Google Docs. Lastly, Cloud9 provides a domain name for testing and demoing the app. For this project the web address was bloombus-server-paniel.c9users.io though it may be inactive now.

Compared to developing it all locally, this was like floating on Cloud9, pun intended.

3.4 Communication

The communication between the modules and the server starts and ends with the XBees.

Along with getting the Arduino to construct the network layer frames using the ZigBee protocol, we came up with an application layer protocol for the actual data we want to send from the modules to the server. The protocol is discussed in more detail in the appendix, but it comes with data such as identity of the bus module, the current time, latitude and longitude, and even more troubleshooting information. The protocol was designed to be easily parsed with regular expressions. An example of the protocol looks like this:

```
IDEN(1),TIME(20:55:40),DATE(10/31/16),FIXQ(1%1),LALO(41.004873%76.268203),SKAA(1.48%36.44%184.80),SATL(4);
```

To test sending and receiving packets using this protocol, we used XCTU's Frame Generator (Figure 21). Using this tool, we were able to troubleshoot how the server would respond to different packets.

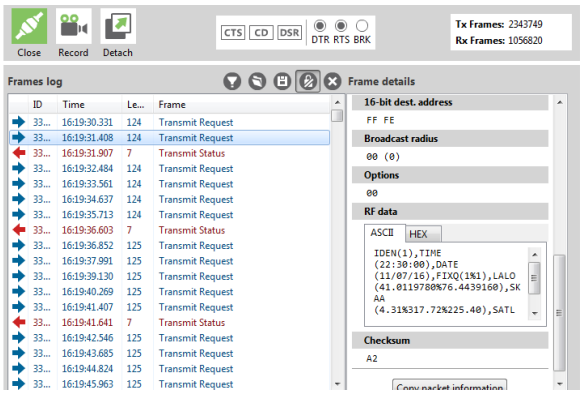


Figure 21
XCTU being used to transmit pre-generated frames.

When an XBee has successfully sent or received a packet, the red LED on the adapter lights up. When the data is received by the Coordinator, it is forwarded to the server (Figure 22). The server then takes that data and uploads it to a database, to be forwarded on to every client with a connected session.

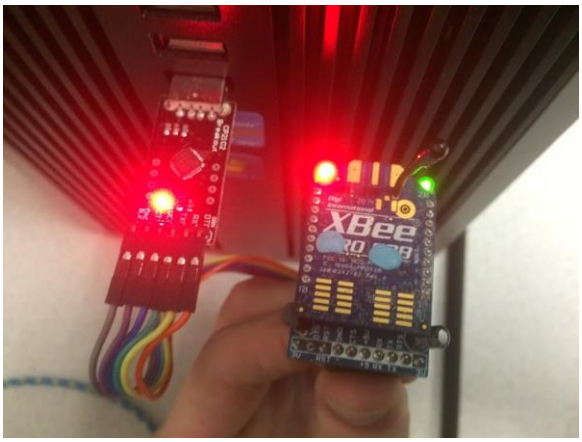


Figure 22
The Coordinator XBee connected to the server with a CP2102 cable.

4 Findings

Our proof of concept was able to be completed within our scope. We have successfully built two prototypes of the bus tracking modules and developed a server which is capable of receiving bus GPS coordinates and displaying them on a map live.

Unfortunately, we have found a few problems with our proof of concept that must absolutely be mentioned and considered if this project is developed further. We found that the range of our XBees only came to be about a tenth of what was advertised (one mile). Of course, this estimate may be the case if there was absolutely nothing in between the XBees, but there will always be buildings, hills, or trees between the busses and the server. This tenth of a mile is laughable when considering the max distance should be enough to accommodate the entire upper campus bus route.

We have also found that the XBee signals barely go through walls, and even windows. When testing our prototype in the University library, our server only seemed to get a signal when the bus tracking module was in direct line of sight. Sometimes, the signal would bounce off walls of other buildings, but it didn't ever seem to be enough.

On the positive side, we discovered that the XBee's capability to form a self-healing mesh networking worked wonderfully. We were successfully able to set up one XBee as a relay to extend the range to the server for a bus tracking module, even when that relay itself was also sending coordinates to the server (Figure 23).

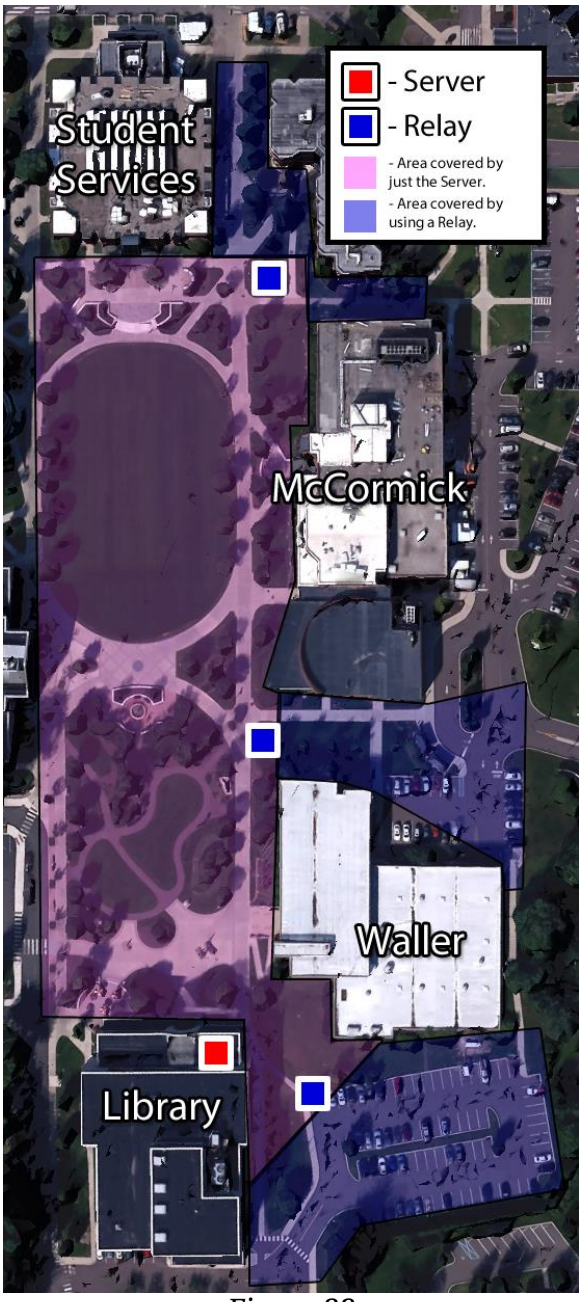


Figure 23
A map of tested XBee coverage from the server and extended by relays.

Also, it's worth noting that our GPS Unit was mostly accurate, but sometimes got thrown way off. When gathering data from a bus tracking module while riding a bus, we found that half way through the loop, the coordinates noticeably shifted. The worst part, was that the shift stayed until we rebooted the GPS, which was importantly only five minutes later.

One last thing we discovered was that our bus tracking website doesn't work on older iPhones. This is due to older browsers not being up to date with the latest JavaScript operations.

5 Recommendations

If this project were to be continued in COMPSCI 480 or by another group that is passionate about seeing this idea be fully implemented into our University, we have a few recommendations for fixing the problems in our findings.

To solve the rather embarrassing problem of small reception distance between XBees, we recommend getting more powerful XBees that can go longer distances. An XBee-Pro 900 XSC S3B Wire (Figure 24) is capable of reaching a signal 28 miles away with a High-Gain Antenna. Since it operates using a 902 to 928 MHz frequency range, the low signal is more easily able to penetrate through walls and other obstructions.

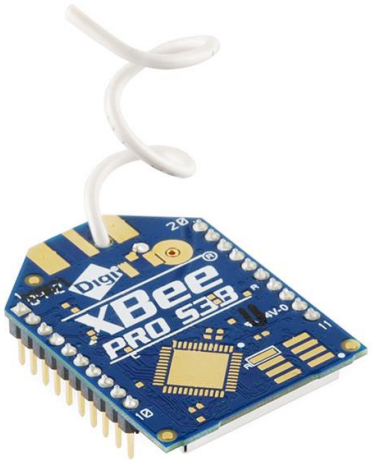


Figure 24
The XBee-Pro 900 XSC S3B Wire, a more capable XBee than our's.

For the GPS inaccuracies, we would highly encourage getting an external antenna. While we're sure any external antenna would do wonders for the unit's precision, we suggest the Adafruit GPS External Antenna and their SMA to uFL Adapter Cable (Figure 25). A software approach to fixing the hardware inaccuracies be forcing the busses snap to roads.

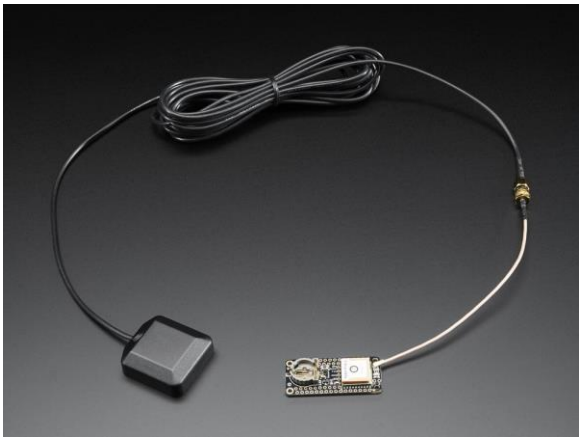


Figure 25
An Adafruit GPS unit connected to an external antenna via an adapter.

Also, we have a recommendation for getting the bus application to operate on older iPhones. It may be possible to resolve this by writing an internal version of Firebase. However, this approach may run the risk of using deprecated methods to run the app. If the app is ever to be ported to a native app using React Native, this issue must be resolved.

Lastly, we have a few features we'd like to add to the actual bus tracking application. We'd like to allow for showing additional busses, and selecting a single bus. When a single bus is selected, we'd allow google maps to lock the bus into the middle of the app and follow it as the bus moves without requiring panning.

6 Conclusion

With the final goal of this project being to make a system capable of tracking a car driving around campus, we can definitively say we completed this goal, even if not ideally. With only the weaker equipment to work with, we were not able to achieve a great distance of range between the bus tracking modules, and the server. It would have been ideal to see the range between the modules and server to stretch across campus, as opposed to just a tenth of a mile away from the library.

However, we believe the distance between XBee units and the accuracy of the GPS unit can be improved with the recommendations previously stated. We hope to see this project fully implemented in the future, and that this document helps to pick up where we left off.

7 References

[1] Brandon Kester, Tyler Specht, Nathan Daley, Matthew Gonzalez, and Daniel Pany (2016). “Bus Tracking Proposal.” Retrieved from: https://docs.google.com/document/d/1wfwctd0I48CctnpOZQ7kpVuwjqEexofUsrX8ZMV09nA/edit?usp=sharing

[2] Wade Cooper, Desiree Dooley, Nick Hoffman, Chris Lombardi, Kieran Raabe, Zach Lytle, Geoffrey Matz, Geoffrey Shapiro (2015). “GP-YES Research Presentation.” Retrieved from: https://docs.google.com/presentation/d/1FxtCKn2D0VU7lYLeEtT7P2sW6AW91hePtacHvtZECic/edit?usp=sharing

[3] Zac Dannelly (2015). “AT vs. API (What, Why, How).” ARMmbed Blog. Retrieved from: https://developer.mbed.org/users/dannellyz/notebook/at-vs-api-when-why-how/

[4] “XCTU [Software & Documentation].” Digi. Retrieved from: https://www.digi.com/products/xbee-rf-solutions/xctu-software/xctu

[5] Jack (2013). “XBee Basics.” Youtube [tunnelsup channel]. Retrieved from: https://www.youtube.com/watch?v=odekkumB3WQ&list=PL1VKPu9fL6A8J93XSi1dI5AbAj8VdEXpp

[6] Jack (2012). “XBee S2 Quick Reference Guide/Cheat Sheet and Video Tutorials to Getting Started.” TunnelsUP Blog. Retrieved from: https://www.tunnelsup.com/xbee-guide/

[7] “XBee Buying Guide.” Sparkfun. Retrieved from: https://www.sparkfun.com/pages/xbee_guide

[8] “XBee-Pro 900 XSC S3B Wire [Product Page].” Sparkfun. Retrieved from: https://www.sparkfun.com/products/11634

[9] “Adafruit Ultimate GPS [Product Page].” Adafruit. Retrieved from: https://www.adafruit.com/product/746

[10] (2013). “Adafruit Ultimate GPS [Explanation].” Youtube [Adafruit Industries channel]. Retrieved from: https://www.youtube.com/watch?v=6A99O-mCZSM

[11] “SMA to uFL Adapter Cable [Product Page].” Adafruit. Retrieved from: https://www.adafruit.com/products/851

[12] “External GPS Antenna [Product Page].” Adafruit. Retrieved from: https://www.adafruit.com/products/960

[13] (2016). “Adding a Google Map.” Google Maps API Documentation. Retrieved from: developers.google.com/maps/documentation/javascript/adding-a-google-map

[14] (2016). “[Firebase] Installation & Setup.” Firebase Documentation. Retrieved from: https://firebase.google.com/docs/database/web/start

[15] “Node.js [Software & Documentation].” Retrieved from: https://nodejs.org/en

[16] “Express.js [Software & Documentation].” Retrieved from: http://expressjs.com

[17] “Cloud9 [Software].” Retrieved from: https://c9.io

8 Appendices

8.1 Arduino Code

```
/**
 * Written by Daniel Pany
 * Fall Semester, 2016
 * Version 1.2
 * Intended for use with the Bloom Bus Tracking Project.
 *
 *
 * The purpose of this Arduino Sketch is to allow an Arduino Uno to act as a bridge between an Ultimate GPS Breakout v3 and an XBee-Pro.
 * This code initializes communications with both the GPS Unit and the XBee.
 * Then it enters a loop, where it continually queries the GPS Unit for data, and sends it to the XBee to be transmitted to the Bus Tracking Server.
 *
 * This sketch uses code found on the Adafruit GPS Library "Parsing" example.
 *
 */

#include <Adafruit_GPS.h>
#include <SoftwareSerial.h>

int TRACKER_ID = 3; // Used to identify which Bus Tracking Module is being operated.
//No two bus arduinos should have the same ID, and this MUST be changed accordingly when updating Arduinos.



// ------------------- GPS Setup Code ------------------- //

// If you're using a GPS module:
// Connect the GPS Power pin to 5V
// Connect the GPS Ground pin to ground
// If using software serial (sketch example default):
//   Connect the GPS TX (transmit) pin to Digital 3
//   Connect the GPS RX (receive) pin to Digital 2

// This SoftwareSerial object is used to communicate with the GPS unit
SoftwareSerial mySerial(3, 2);

// This GPS object allow us to interact with the GPS unit
Adafruit_GPS GPS(&mySerial);



// Set GPSECHO to 'false' to turn off echoing the GPS data to the Serial console
// Set to 'true' if you want to debug and listen to the raw GPS sentences.
#define GPSECHO  true

// this keeps track of whether we're using the interrupt
// off by default!
boolean usingInterrupt = false;
void useInterrupt(boolean); // Func prototype keeps Arduino 0023 happy



// ------------------- XBee Setup Code ------------------- //

//This SoftwareSerial object is used to communicate with the XBee
SoftwareSerial xbee(4, 5); // RX, TX // GRN, WHT
```

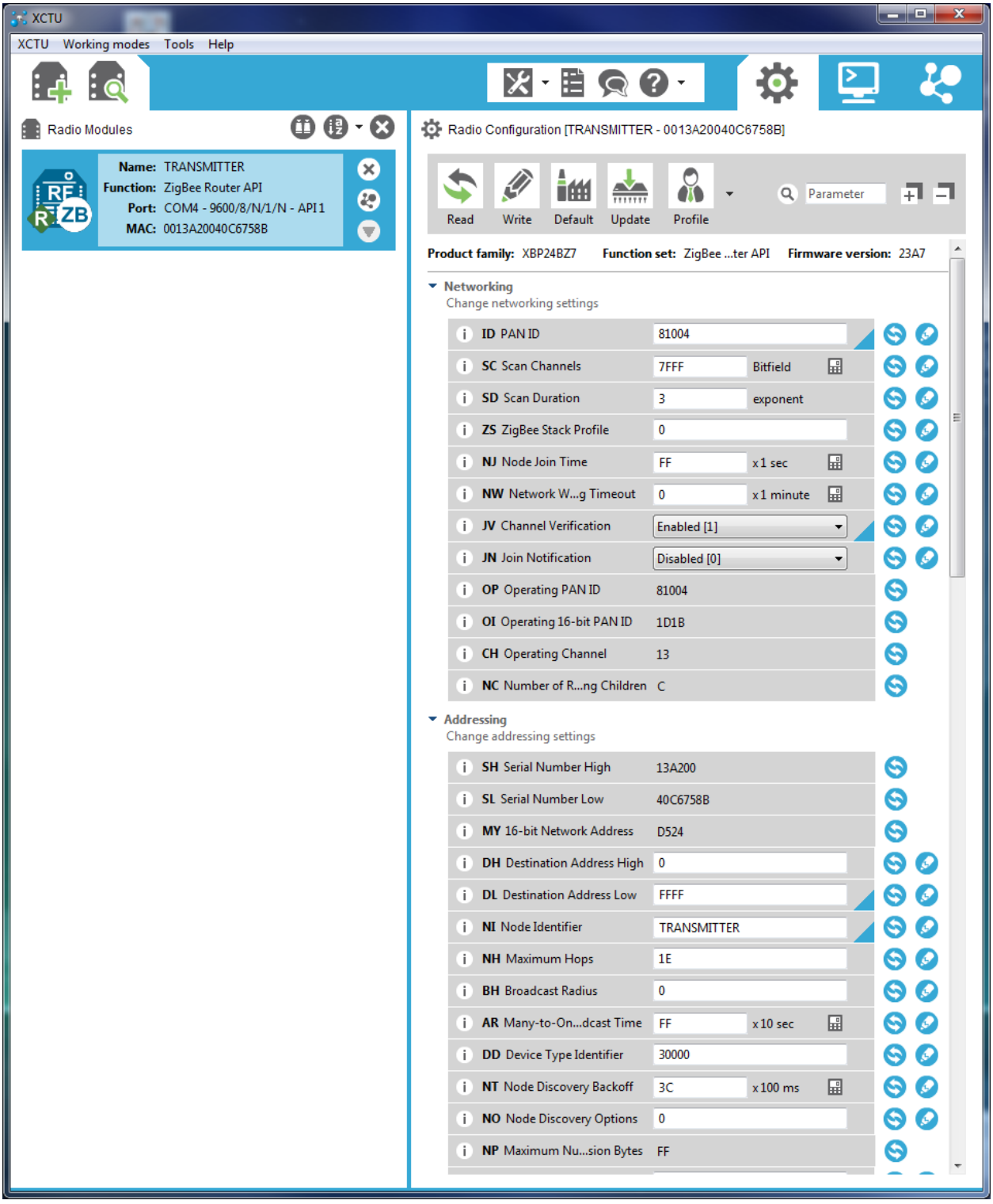
```
byte temp[256]; //This array is used to construct the entire XBee ZigBee Transmit frame.




// ------------------- Bus Tracker Custom Functions ------------------- //

//This function allows the creation of XBee Transit Frames with messages of up to 238 bytes in length
void constructFrame(String message)
{
  unsigned msgLen = message.length(); //Message Length
  unsigned frmLen = msgLen + 14; //Frame Length
  byte frmLenLow = (char) frmLen; //
  byte frmLenHigh = (char) (frmLen/256);

  temp[0] = 0x7E; //Start Transmit
  temp[1] = frmLenHigh; //Length High
  temp[2] = frmLenLow; //Length Low
  temp[3] = 0x10; //Frame Type
  temp[4] = 0x01; //Frame ID
  for (unsigned i = 5; i < 13; i++) temp[i] = 0x00; //8 bytes for the 64-bit dest. address
  temp[13] = 0xFF; //16-bit dest addr. high
  temp[14] = 0xFE; //16-bit dest addr. low
  temp[15] = 0x00; //Broadcast Radius
  temp[16] = 0x00; //Options
  for (short i = 17; i < 17+msgLen; i++) temp[i] = message.charAt(i-17); //Dedicate as many bytes as needed for the message to
be sent.

  char checksum = 0x00; //Checksum of frame
  for (short i = 3; i < 3+14+msgLen; i++) checksum = checksum+(char)temp[i]; //Sum every byte of the frame from byte 3 to
the last byte.
  temp[17+msgLen] = 0xFF-checksum; //Append the resulting checksum.

}



// This function takes an integer, and pads it with a leading 0 if it is less than ten.
// Example Case 1:  8:23 -> 08:23
// Example Case 2:  11:2 -> 11:02
// Example Case 3: 11:20 -> 11:20
String timePad(int timer)
{
  if (timer < 10) return "0" + String(timer);
  return String(timer);
}




// ------------------- Main Setup Function ------------------- //

// The "main" function for Arduino.
void setup()
{

  // Begin communications with the computer. (Optional. This is only for troubleshooting)
```

```
  Serial.begin(9600);
  Serial.println("Bus Tracker Communications has begun.");

  // Begin communications with the XBee/
  xbee.begin( 9600 );

  // Begin communication with the GPS unit.
  GPS.begin(9600);

  // uncomment this line to turn on RMC (recommended minimum) and GGA (fix data) including altitude
  GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCGGA);
  // uncomment this line to turn on only the "minimum recommended" data
  //GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCONLY);
  // For parsing data, we don't suggest using anything but either RMC only or RMC+GGA since
  // the parser doesn't care about other sentences at this time

  // Set the update rate
  GPS.sendCommand(PMTK_SET_NMEA_UPDATE_1HZ);   // 1 Hz update rate
  // For the parsing code to work nicely and have time to sort thru the data, and
  // print it out we don't suggest using anything higher than 1 Hz

  // Request updates on antenna status, comment out to keep quiet
  GPS.sendCommand(PGCMD_ANTENNA);

  // the nice thing about this code is you can have a timer0 interrupt go off
  // every 1 millisecond, and read data from the GPS for you. that makes the
  // loop code a heck of a lot easier!
  useInterrupt(false);

  delay(1000);
  // Ask for firmware version
  mySerial.println(PMTK_Q_RELEASE);
}




// ------------------- Necessary GPS Library Functions ------------------- //

// Interrupt is called once a millisecond, looks for any new GPS data, and stores it
SIGNAL(TIMER0_COMPA_vect) {
  char c = GPS.read();
  // if you want to debug, this is a good time to do it!
#ifdef UDR0
  if (GPSECHO)
    if (c) UDR0 = c;
    // writing direct to UDR0 is much much faster than Serial.print
    // but only one character can be written at a time.
#endif
}

void useInterrupt(boolean v) {
  if (v) {
    // Timer0 is already used for millis() - we'll just interrupt somewhere
    // in the middle and call the "Compare A" function above
    OCR0A = 0xAF;
    TIMSK0 |= _BV(OCIE0A);
    usingInterrupt = true;
  } else {
    // do not call the interrupt function COMPA anymore
    TIMSK0 &= ~_BV(OCIE0A);
    usingInterrupt = false;
  }
}
```

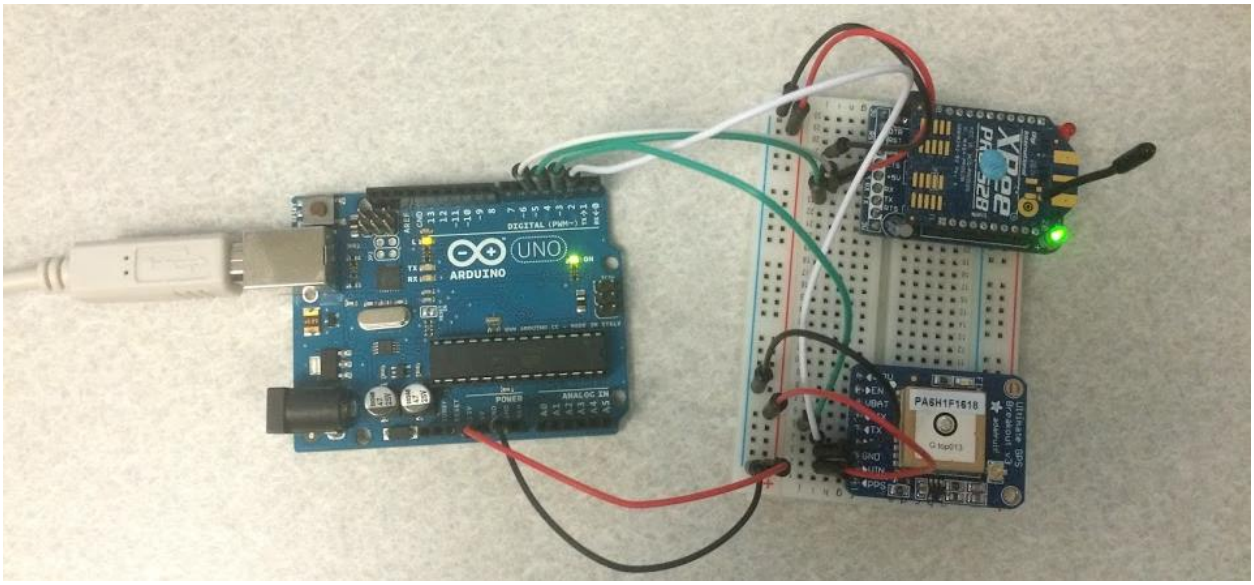
```
// -------------------- Main Loop Function -------------------- //

uint32_t timer = millis();

// This function is called as soon as the setup() function is completed. The code within loop() will repeat as long as the Arduino is powered on or crashes.
void loop()
{
  // _________________ START of Necessary GPS Library Code _________________ //

  // in case you are not using the interrupt above, you'll
  // need to 'hand query' the GPS, not suggested :(
  if (! usingInterrupt) {
    // read data from the GPS in the 'main loop'
    char c = GPS.read();
    // if you want to debug, this is a good time to do it!
    if (GPSECHO)
      if (c) delay(1);//Serial.print(c);
  }

  // if a sentence is received, we can check the checksum, parse it...
  if (GPS.newNMEAreceived()) {
    // a tricky thing here is if we print the NMEA sentence, or data
    // we end up not listening and catching other sentences!
    // so be very wary if using OUTPUT_ALLDATA and trytng to print out data
    //Serial.println(GPS.lastNMEA());   // this also sets the newNMEAreceived() flag to false

    if (!GPS.parse(GPS.lastNMEA()))   // this also sets the newNMEAreceived() flag to false
      return;  // we can fail to parse a sentence in which case we should just wait for another
  }

  // if millis() or timer wraps around, we'll just reset it
  if (timer > millis())  timer = millis();

  // _________________ END of Necessary GPS Library Code _________________ //



  // _________________ START Sending GPS Message _________________ //

  // approximately every 2 seconds or so, print out the current stats
  if (millis() - timer > 2000) {
    timer = millis(); // reset the timer `

    // This string contains the data to be sent to the server. This data exists whether or not the GPS is currentling getting coordinates.
    String gpsString = String("IDEN("+String(TRACKER_ID)+"),")
       + "TIME(" + timePad(GPS.hour) + ":" + timePad(GPS.minute) + ":" + timePad(GPS.seconds) + "),"
       + "DATE(" + timePad(GPS.month) + "/" + timePad(GPS.day) + "/" + String(GPS.year) + "),"
       + "FIXQ("+ String(GPS.fix) + "%" + String(GPS.fixquality) + ")";

    // If the GPS is currently getting coordinates, append this other data to the string.
    if (GPS.fix)
    {
      gpsString += ",LALO("+String(GPS.latitudeDegrees,7) + "%" + String(GPS.longitudeDegrees * -1.0,7) + "),"
      + "SKAA(" + String(GPS.speed) + "%" + String(GPS.angle) + "%" + String(GPS.altitude) + "),"
      + "SATL(" + String((int)GPS.satellites) + ")";
```

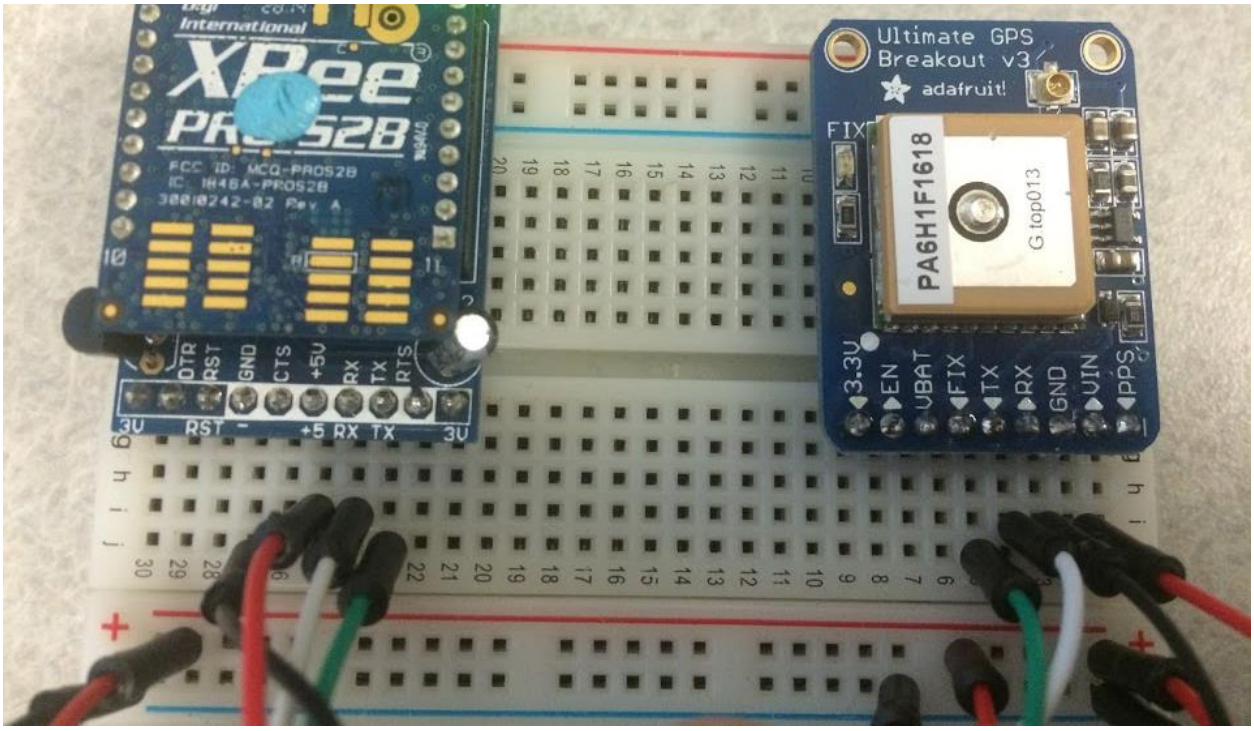
```
    }

    //Start printing troubleshooting information.
    Serial.print("\nTime: ");
    Serial.print(GPS.hour); Serial.print(':');
    Serial.print(GPS.minute); Serial.print(':');
    Serial.print(GPS.seconds); Serial.print('.');
    Serial.println(GPS.milliseconds);
    Serial.print("Date: ");
    Serial.print(GPS.day, DEC); Serial.print('/');
    Serial.print(GPS.month, DEC); Serial.print("/20");
    Serial.println(GPS.year, DEC);
    Serial.print("Fix: "); Serial.print((int)GPS.fix);
    Serial.print(" quality: "); Serial.println((int)GPS.fixquality);
    if (GPS.fix) {
      Serial.print("Location: ");
      Serial.print(GPS.latitude, 4); Serial.print(GPS.lat);
      Serial.print(", ");
      Serial.print(GPS.longitude, 4); Serial.println(GPS.lon);
      Serial.print("Location (in degrees, works with Google Maps): ");
      Serial.print(GPS.latitudeDegrees, 4);
      Serial.print(", ");
      Serial.println(GPS.longitudeDegrees * -1.0, 4);

      Serial.print("Speed (knots): "); Serial.println(GPS.speed);
      Serial.print("Angle: "); Serial.println(GPS.angle);
      Serial.print("Altitude: "); Serial.println(GPS.altitude);
      Serial.print("Satellites: "); Serial.println((int)GPS.satellites);
    }

    Serial.println("\n\nNow Constructing Packet...");Serial.println();

    gpsString += ";";

    // Convert the String to a frame, and put into the global "temp" array created earlier.
    constructFrame(gpsString);

    // Print the resulting frame for troubleshooting purposes
    short frameLength = 18+gpsString.length();
    for (unsigned i = 0; i < frameLength; i++) Serial.print((char)temp[i]);
    Serial.println();Serial.println();
    Serial.println();Serial.println();

    //Send the frame to XBee
    xbee.write(temp, frameLength);

    // _________________ END Sending GPS Message _________________ //

  }
}
```

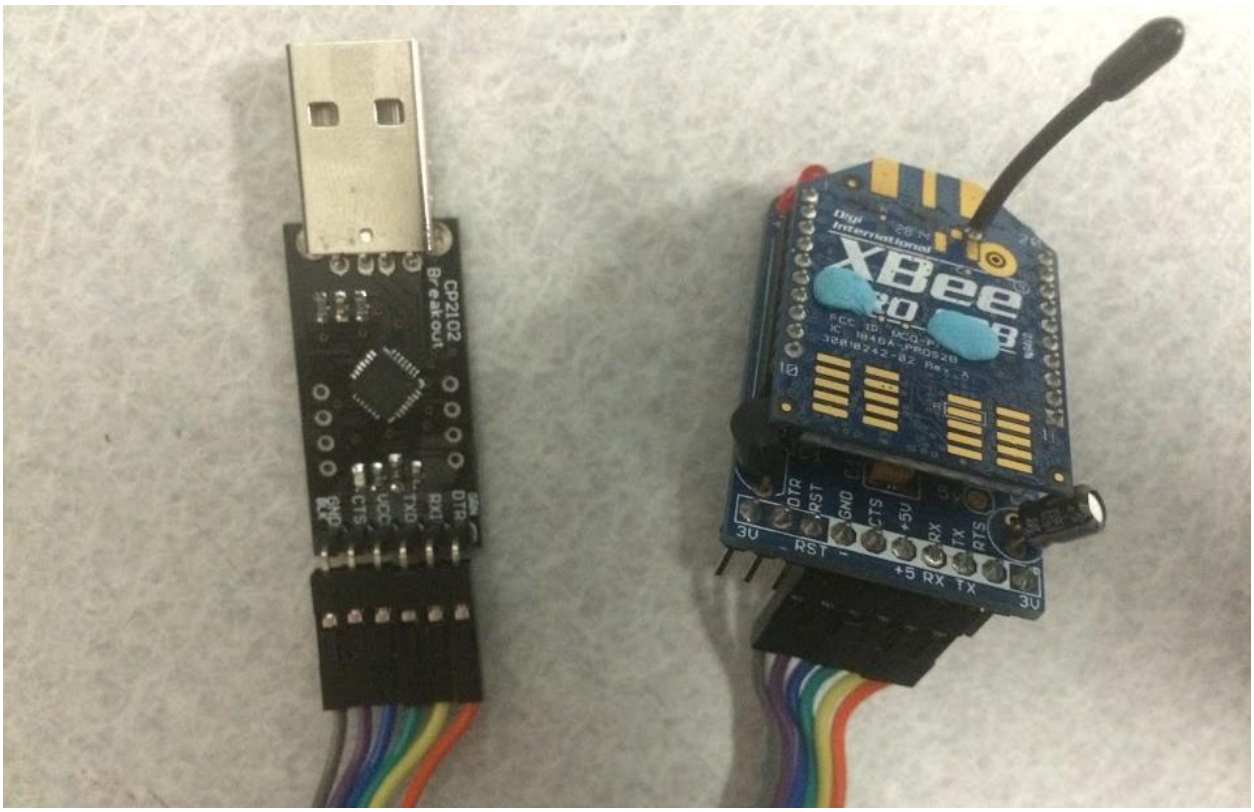
8.2 XBee Configurations



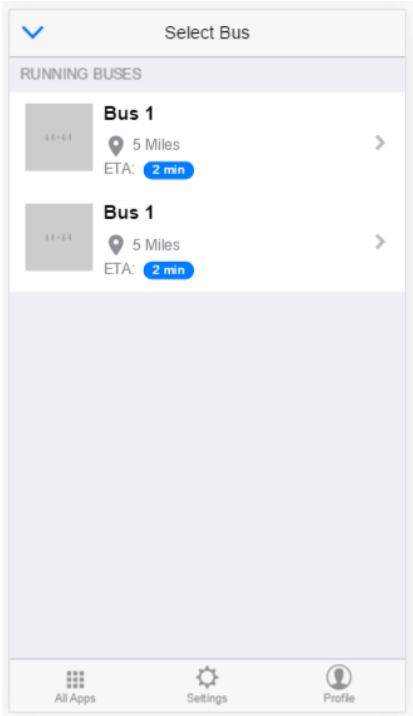
8.3 References for Wiring the Components

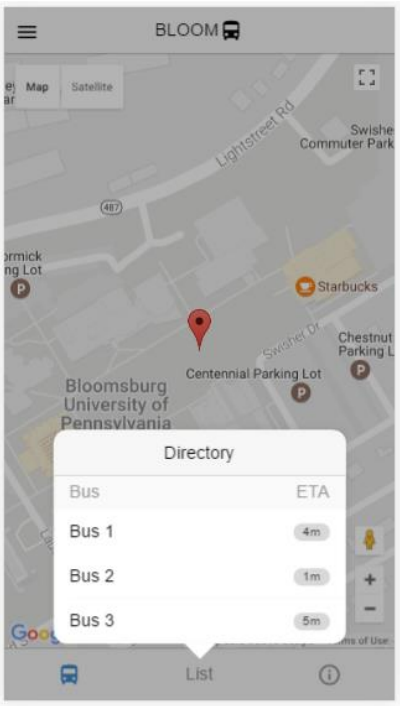


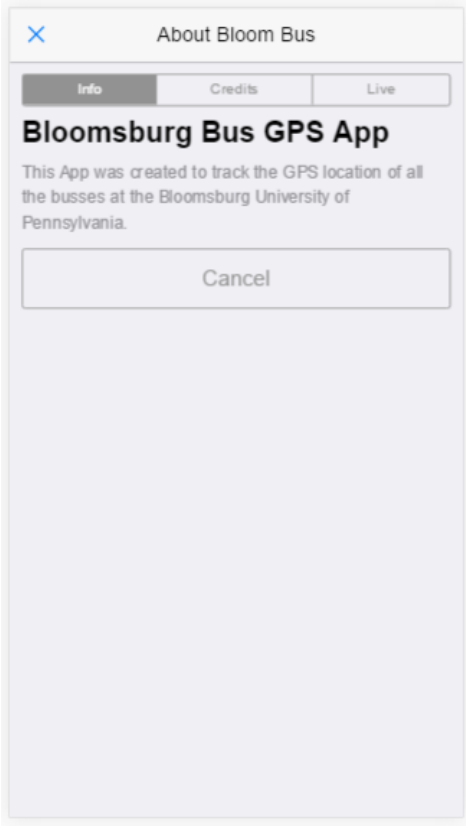


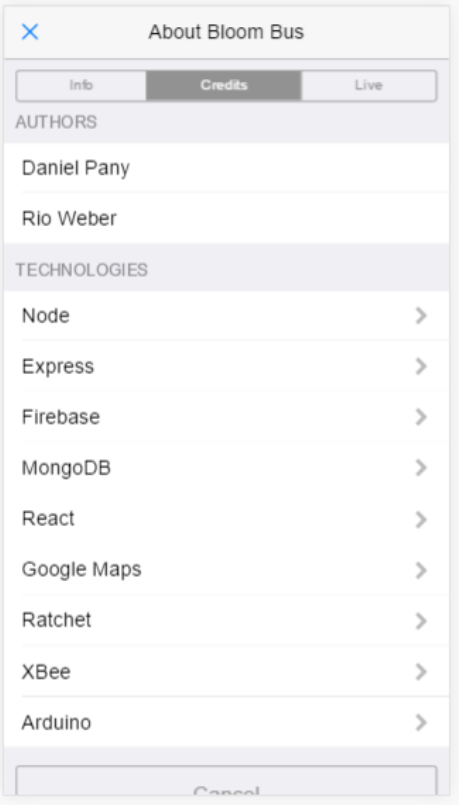


8.4 Web Application for Mobile Devices


Select Bus
RUNNING BUSES
Bus 1
5 Miles
ETA: 2 min
Bus 1
5 Miles
ETA: 2 min
All Apps
Settings
Profile


BLOOM
Map
Satellite
Lightstreet Rd
Swisher Commuter Park
Starbucks
Centennial Parking Lot
Chestnut Parking L
Bloomsburg University of Pennsylvania
Directory
Bus
ETA
Bus 1
4m
Bus 2
1m
Bus 3
5m
List


About Bloom Bus
Info
Credits
Live
Bloomsburg Bus GPS App
This App was created to track the GPS location of all the busses at the Bloomsburg University of Pennsylvania.
Cancel


About Bloom Bus
Info
Credits
Live
AUTHORS
Daniel Pany
Rio Weber
TECHNOLOGIES
Node
Express
Firebase
MongoDB
React
Google Maps
Ratchet
XBee
Arduino