Final Project Report - Team 1

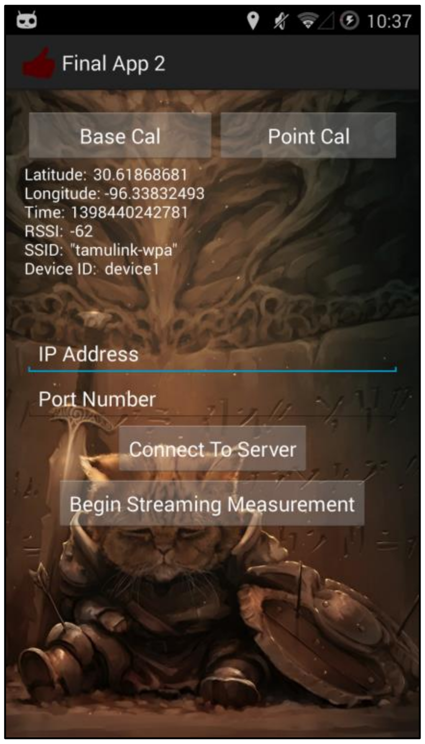
Clayton Crawford - Team Leader
Jeff Terrell - Hardware Contact
Jason Moore - Algorithm Contact
Jeff Jensen - App Contact
Zach Partal - Server Contact

Project Overview

In a perfect world, Wi-Fi radio signals would be consistently strong no matter where the user is located. Unfortunately, we live in the real world where buildings, trees, etc. inhibit the transmission of Wi-Fi radio waves. The focus of this project is to provide a means for a user to always receive the strongest Wi-Fi signal as they move around an area. In order to achieve this, an antenna made from a tin can, dubbed cantenna, is mounted on a stepper motor and is programmed to track the user that is receiving transmissions from it. The project is split into three phases, the last of which is a demonstration of the system as a whole.

Development: Phase I

Android Application - Jeff J., Clayton



Latitude: 30.61868681
Longitude: -96.33832493
Time: 1398440242781
RSSI: -62
SSID: "tamulink-wpa"
Device ID: device1

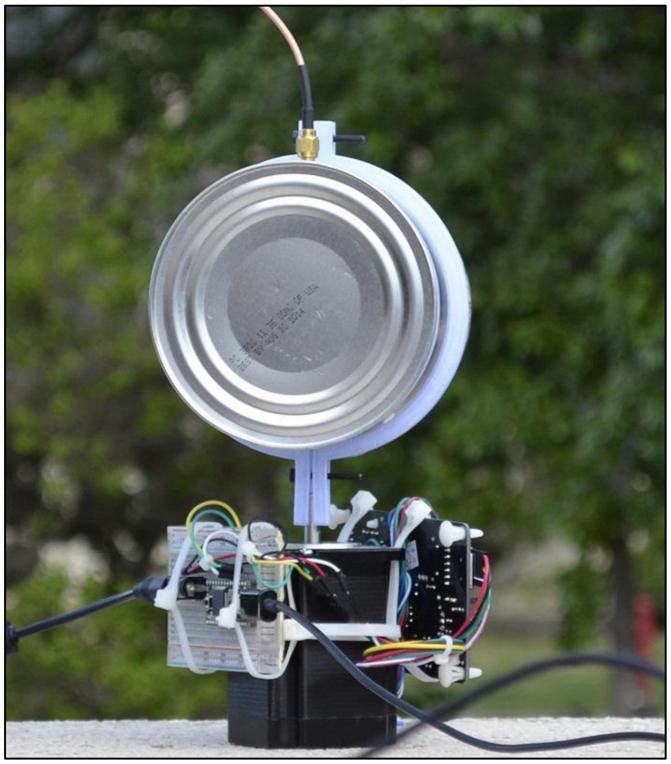
The Android application (seen to the right) is the main data collection method. The app has the ability to collect GPS and RSSI data pairs, store the information on an internal SQLite database, and then offload the data to our server. For Phase I, our team went to the test area outside to walk around and take GPS/RSSI measurements in order to form a mapping of the area to see where we receive strong and weak signals.

Motor Control - Jeff T.

Equipment List:

- 2 WiFi “Cantenna”
- Lin Engineering 5718 High Torque Stepper Motor
- Cytron Technologies SD02B Enhanced 2A Stepper Motor Driver
- Teensy 3.0 microcontroller
- Custom mounting hardware for antenna to motor shaft
- 2 WiFi dongles with SMA attachment
- SMA cables

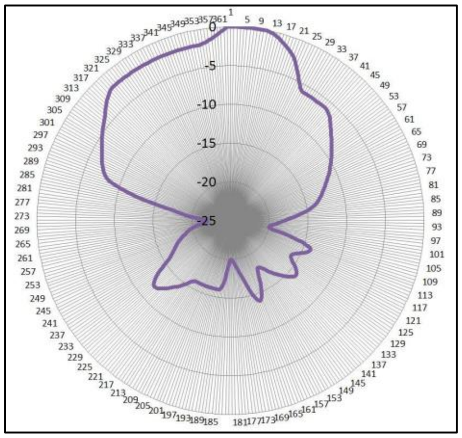
The motor mounting hardware is composed of two cantenna clamps and a motor shaft adapter. The motor shaft adapter was designed in SolidWorks 2013 and exported to .stl format for 3-D printing. The cantenna clamps were 3-D printed using a supplied set of CAD files. During the 3-D printing process for both parts, some technical difficulties with the 3-D printer were encountered approximately 60% into the print. The parts were salvageable and it was cost-effective to go ahead and use the incomplete parts. Some modifications were made in order to affix the cantenna/clamp assembly to the motor shaft adapter. The adapter originally had a slot cut out to which the clamp assembly would mate. Since the print was incomplete, the slot was not constructed, and the parts were bonded using super glue. Eventually, this revision of the mounting hardware failed and we printed new, complete parts. These new parts have been reliable throughout the testing of the system.



Because of unreliable operation of the stepper motor driver using the UART control interface, a set of manual control pins on the driver are used to control pulse frequency and direction of rotation. For a stepper motor operating with no microstepping, one pulse corresponds to a rotation by one step. The 5718 motor has 200 steps and the non-microstepping angular resolution of the motor is calculated by dividing one revolution in degrees by the number of steps, meaning that each step represents 1.8 degrees of rotation. The project requirements call for motor resolution of one degree. This is not feasible without UART control for this particular stepper motor driver. In real-time system operation, accuracy will suffer over time as a result of the limitation on angular resolution.

The control method implemented utilizes the PWM capability of the microcontroller to set the pulse frequency and duty cycle of the signal sent to the PULSE pin on the stepper motor driver. The number of pulses is determined by the angle received from the server. This angle corresponds to a position between -180 and +180 degrees. Upon receiving this angle, the stepper motor driver will send a number of pulses to the motor that corresponds to the angle/position. The direction of rotation is controlled by the DIR pin on the stepper motor driver. This pin is driven by a separate digital output pin on the microcontroller. The control software determines if the angle is positive or negative, and then sets the DIR pin high or low to rotate the motor clockwise or counterclockwise, respectively.

Antenna Characterization - Jeff T., Zach, Clayton, Jason

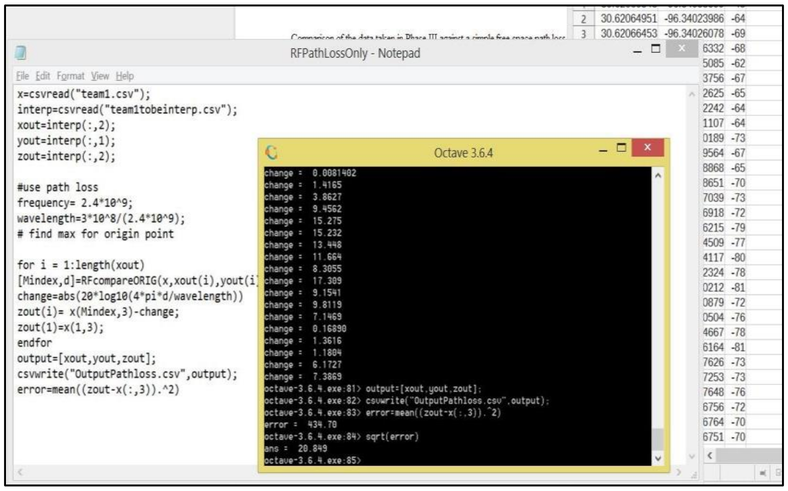


- AUT (antenna under test) mounted to motor → connected via SMA cable/WiFi dongle to laptop running software to retrieve RSSI data from AUT
- motor is rotated incrementally from 0 to 360 degrees and associated RSSI data is recorded
- Normalized polar radiation plot is constructed (as seen right.)

Development: Phase II

Algorithm - Jason

The algorithm is based on bilinear interpolation of points. Bilinear interpolation operates by defining triangular areas using known points as vertices. Points have three values: longitude, latitude, and RSSI. All three values are defined at vertices, while bilinear interpolation is used to define the RSSI values for selected points where RSSI is unknown. It is assumed that RSSI varies uniformly along longitude and latitude. RSSI is equal to X*longitude, Y*latitude, + Z where X Y and Z are constants. X Y and Z are determined by solving a system of equations using the known vertices. If a point to be interpolated falls outside of any of the triangular areas defined by any combination of known points, then free space path loss is assumed to interpolate these points. Free space path loss follows the equation



```
x=csvread("team1.csv");
interp=csvread("team1tobeinterp.csv");
xout=interp(:,2);
yout=interp(:,1);
zout=interp(:,2);

#use path loss
frequency= 2.4*10^9;
wavelength=3*10^8/(2.4*10^9);
# find max for origin point

for i = 1:length(xout)
[Mindex,d]=RFcompareORIG(x,xout(i),yout(i
change=abs(20*log10(4*pi*d/wavelength))
zout(i)= x(Mindex,3)-change;
zout(1)=x(1,3);
endfor
output=[xout,yout,zout];
csvwrite("OutputPathloss.csv",output);
error=mean((zout-x(:,3)).^2)
```

```
change =  0.0081402
change =  1.4165
change =  3.8627
change =  9.4562
change =  15.275
change =  15.232
change =  13.448
change =  11.664
change =  8.3055
change =  17.309
change =  9.1541
change =  9.8119
change =  7.1469
change =  0.16890
change =  1.3616
change =  1.1804
change =  6.1727
change =  7.3869
octave-3.6.4.exe:81> output=[xout,yout,zout];
octave-3.6.4.exe:82> csvwrite("OutputPathloss.csv",output);
octave-3.6.4.exe:83> error=mean((zout-x(:,3)).^2)
error =  434.70
octave-3.6.4.exe:84> sqrt(error)
ans =  20.849
octave-3.6.4.exe:85>
```

```
2  30.62064951  -96.34023986  -64
3  30.62066453  -96.34026078  -69
6332 -68
5085 -62
3756 -67
2625 -65
2242 -64
1107 -64
0189 -73
9564 -67
8868 -65
8651 -70
7039 -73
6918 -72
6215 -79
4509 -77
4117 -80
2324 -78
0212 -81
0879 -72
0504 -76
4667 -78
6164 -81
7626 -73
7253 -73
7648 -76
6756 -72
6764 -70
6751 -70
```

$$Path\ Loss = \left(\frac{4\pi D}{\lambda}\right)^2$$

where λ is the wavelength. The distance, D, is calculated from the defined point with the highest RSSI value.

Server - Zach, Jeff J.

The server is a central part of the final project. The software running on the server is responsible for communicating with the Android device, processing the data from the Android device, running the tracking algorithm, storing data, and delivering commands over serial to the motor control Teensy. Due to the wide range of functionality required from the server it is architectured in such a way that individual tasks can be run in their own threads while still maintaining access to the data being fed in from the Android device. The server software is divided into three main threads that run in parallel. The Worker thread, the DatabaseHandler thread, and the ServoDriver thread. The functionality of each of these threads as well as any helper classes are explored in this report.

The server software's entry point is located in the Server.java class. Upon initialization the server begins an unending loop of listening for new sockets being created on its port. The creation of a new socket on the server's port signals the connection of an Android device. With the connection of an Android device the Server class passes the socket into a worker thread.

Each worker thread is connected to an individual Android client. For this specific project only one client would ever be connected to the server. The worker thread initialization begins with the creation of two thread safe queues. The thread safe queues are used to transfer information to the DatabaseHandler and ServerDriver classes that will soon be launched. The workers primary task is to constantly be listening for, and grabbing data packets which are being sent at a rate of one every five seconds by the Android device.

Secondly, the worker thread includes the functionality of a very important part of the antenna tracking algorithm. The first two messages which are sent by the Android device are used to calibrate the tracking algorithm. The first point received is the base station coordinates. These coordinates are the latitude and longitude of the cantenna which forms the common point for all calculations. The second point received is used as a calibration point. The calibration point along with the base station coordinates create a vector which will then be used for determining the angle at which the antenna must rotate to face the Android device.

At this point the DataPackets object shall be discussed. The DataPackets object is a custom serializable object which consists solely of a list of fields for storing data. The six fields in the DataPackets for the final project are as follows: time, device_id, ssid, rssi value, longitude, and latitude. The Android device packages up sensor readings into a DataPacket and sends the data serially over the network to the server.

Each DataPacket that is received and processed by the Worker thread is sent to two places, the DatabaseHandler queue and the ServoDriver queue. The DatabaseHandler is a thread which is launched at initialization of the Worker thread. The DatabaseHandler thread has a single purpose, to take the DataPackets which the Worker captures and insert them into a local SQLite database. To do this the server uses an sqlite-JDBC driver which allows for the use of Java's built in SQL facilities. The DatabaseHandler threads functionality is simple. First the latest DataPacket in the queue is removed. A createInsertSQL function is used to convert the DataPacket into a properly structured SQL insert query which matches the local SQLite databases schema. An SQL statement with the insert SQL is created and the statement is executed on the local database. This process is repeated until a DataPacket with the ssid value set to "quit" is received. Thus gracefully ending the DatabaseHandler thread.

The ServoDriver thread works similarly to the DatabaseHandler thread in that it loops continuously retrieving values from a queue until the "quit" packet is received. Where the ServoDriver thread differs though is in what it does with the DataPackets when they are retrieved.

Before continuting the SerialHandler class must be explained. The SerialHandler class is a helper class to the ServoDriver class. The SerialHandlers main function is to provide methods for communicating with the servo driver hardware over a serial connection. This is accomplished through the use of the RXTX Java library. The SerialHandler class uses the RXTX library to create and open a serial communication link to the Teensy and provide an easy to use sendData function which accepts a string as its parameter.

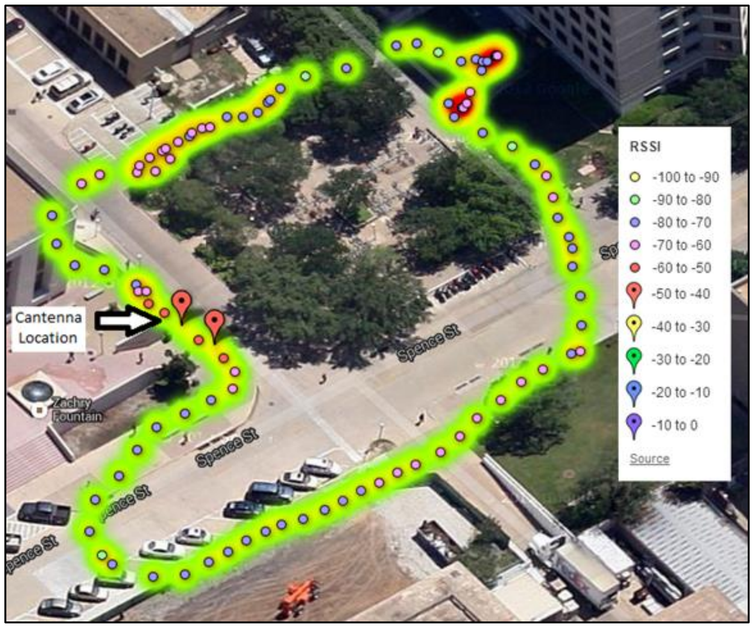
With the functionality of the SerailHandler class explained the rest of the ServoDriver thread can be discussed. Upon retrieval of a DataPacket from the queue the latitude and longitude fields are separated and along with the base station coordinates and the calibration point are passed into the getMotorAngle function in the ServoDriver class. The functionality of the getMotorAngle function is explained in great detail in the algorithm section of the report. Upon execution the getMotorAngle function returns angle to which the motor must rotate to. Using the SerialHandler's sendData method the angle is sent to the Teensy and the motor rotates to its specified position.

While the server software is fully functional there are several areas where improvements could be made. A quick improvement to be made would be to remove the tracking algorithm code from the ServoDriver class and place it into its own separate class. This would make it simple to swap in different algorithms for testing or different tracking situations. Another possible improvement would be to expand the robustness of the server. Currently sudden disconnects of clients cause exceptions and are not handled gracefully. Finally, a search and reconnect function could be added to the ServoDriver. This function would activate upon loss of connection with the Android device. Upon disconnection the server would instruct the servo to begin a slow sweep rotation until the connection with the Android device is reestablished.

Tracking Algorithm

The tracking algorithm is executed within the server and operates through the use of vector math. During a calibration process a vector is created by taking a base coordinate at the antenna and a calibration coordinate at an arbitrary distance directly ahead of the direction the antenna is facing. This vector is then normalized to a unit vector. As the server receives new coordinates from the Android device it can create a second vector by using the base coordinate and the received coordinate. The angle between the two vectors is then calculated through a cross product operation. As the angle calculation function only produces angles between -90 and 90 degrees, logic was put in place to calculate angles that are between -180 and 180 degrees to get a full range of tracking. The calculated angle is then able to be sent to the motor driver and the antenna tracks the Android device.

Results: Phase III



Phase III was a test of all the subsystems put together. One of the team members took an Android device running the previously mentioned application and walked a predefined path around the cantenna. The app captured GPS/RSSI data points every five seconds and pushed the points to the server in order to control the motor direction. The screenshot to the right is a capture of our GPS and RSSI data pairs uploaded to a Google Fusion Table. We can see that when the line of sight with the cantenna and the Android device is somewhat broken, the signal strength decreases (as expected.)

Comparison of the data taken in Phase III against a simple free space path loss model was conducted and its ability to interpolate removed data values was evaluated. The mean error based on simple path loss was 20.89 dB. The algorithm used the highest RSSI value as the antenna location and interpolated missing data based on $t$he path loss equation used in Phase II, where distance D was determined by the haversine function and wavelength λ was assumed to be 0.125m at 2.4 GHz.

Conclusion

Overall, the project was a success. The team was able to have each of the parts (Android application, server, and motor) communicate together simultaneously without any issues. The only issue that occurred is that the motor seemed incapable of handling small changes in rotation. The motor would make a visible jitter, however the position remained stuck. This was remedied by having the person walking the predefined path make a larger change in location i.e. walk faster.