6-9-2020

LeaPi: Wireless Diagnostic Assistant

Jeff Collins

Austin Iverson

Collin Seaman

Joseph Sindelar

SANTA CLARA UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Date: June 9, 2020

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

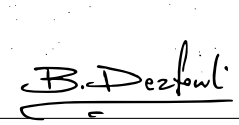
Jeff Collins
Austin Iverson
Collin Seaman
Joseph Sindelar

ENTITLED

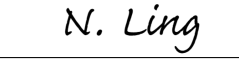
LeaPi: Wireless Diagnostic Assistant

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREES OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING
BACHELOR OF SCIENCE IN WEB DESIGN AND ENGINEERING



Thesis Advisor



Department Chair

LeaPi: Wireless Diagnostic Assistant

by

Jeff Collins
Austin Iverson
Collin Seaman
Joseph Sindelar

Submitted in partial fulfillment of the requirements
for the degrees of
Bachelor of Science in Computer Science and Engineering
Bachelor of Science in Web Design and Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 9, 2020

LeaPi: Wireless Diagnostic Assistant

Jeff Collins
Austin Iverson
Collin Seaman
Joseph Sindelar

Department of Computer Science and Engineering
Santa Clara University
June 9, 2020

ABSTRACT

Nearly every person who uses WiFi on a daily basis has had trouble with a bad connection. Wireless connectivity issues are often difficult to diagnose and fix. Current solutions such as wired extenders, and Mesh WiFi commercial packages are expensive and do not provide the user with a system that suggests placement of mesh units to maximize coverage. Our solution is an inexpensive and open-source diagnostic tool that maps out Wifi quality and informs the user of interference. With a simple, meaningful display, users will find trouble spots in their house, diagnose why IoT devices are not working, effectively place WiFi extenders and mesh nodes, and more.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Nearly every person who uses WiFi on a daily basis encounters connection difficulties. Wireless connectivity issues are also often difficult to diagnose and fix. Existing wireless diagnostic and extender tools are expensive, costing the consumer hundreds of dollars per unit. Current solutions for wireless diagnostics include the RF Explorer[1], an antenna paired with desktop software used to detect wireless interference. However, the Explorer is a complicated tool designed for professional engineers and technicians. Another solution is the HPE Aruba[2], a wall-mounted diagnostic tool for commercial applications. Neither of these solutions can be built upon by a user, and neither option is designed with consumers on a budget in mind. Therefore, there exists a need for a wireless diagnostic tool and software platform that is easy to use, open source, and inexpensive, so that the average person can optimize their home network.

[1] http://rfexplorer.com/wifisurveyor/
[2] https://capenetworks.com/

Chapter 2

Solution

2.1 Our Design

To solve the aforementioned problem, our project utilizes inexpensive materials to build an open-source product that detects and diagnoses WiFi blind spots in the home or office. A user of our device moves around their environment while our portable diagnostic tool collects data regarding packet loss on the network. The diagnostic tool also simultaneously communicates with three other beacon devices at the edges of the network, so that the relative position of the diagnostic tool can be computed. Collected data is then pushed to a locally hosted dynamic graph of network quality in order to advise the user on optimal extender and IoT device placement. This content is accessible through a web app that controls calibration, scanning, and displays the graph in real time on any device with a connection to the network.

Chapter 3

System Overview

3.1 Icons Used

There are three different icons used to represent the various device types in our Full System Diagram. These icons, along with a description of how each is used in the system, are listed as follows.

3.1.1 Access Points

Access Points are represented as the following icon:

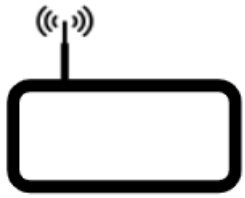


Figure 3.1: Representation of an Access Point

The two types of Access Points are Main and Meshed. The Main Access Point is the WiFi router, while the Meshed Access Points transmit standard Access Point beacon frames that are used in the trilateration process. The trilateration process, and the detailed involvement of each Meshed Access Point, will be discussed in the Trilateration section later on. The Access Points are labelled ”Main AP” or ”Meshed AP” on the diagram to distinguish between the types.

3.1.2 Diagnostic Node

The Diagnostic Node is represented by the following icon:

Figure 3.2: Representation of a Diagnostic Node

The Diagnostic Node is a device that is moved around the room to calculate the network coverage in various locations. It also hosts the web app. It is the main point of user interaction.

3.1.3 Diagnostic Server

The Diagnostic Server is represented by the following icon:

Figure 3.3: Representation of a Diagnostic Server

The Diagnostic Server is tested for packet retention rate between itself and the Diagnostic Node at each user-requested scan point to acquire a metric for network coverage. It is connected to the Main Access Point via Ethernet cable to avoid any packet loss between the two, ensuring that any packet loss measured is between itself and the Diagnostic Node.

3.1.4 Full System Diagram

The following figure is the full system diagram for LeaPi:

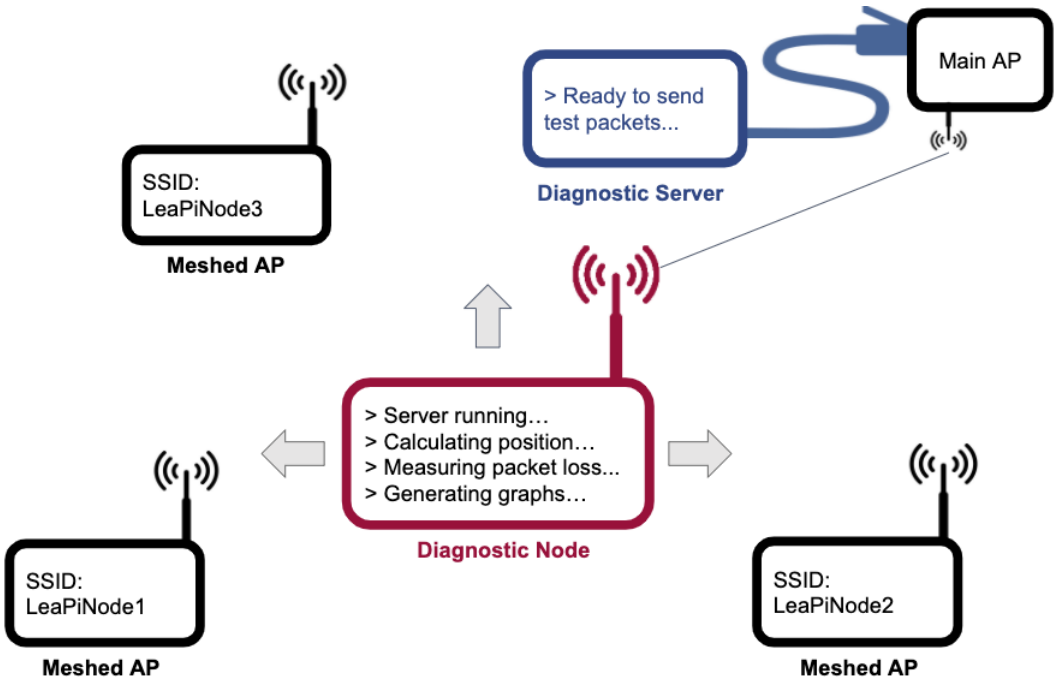


Figure 3.4: Full System Diagram for LeaPi

Chapter 4

Technologies Used

4.1 Introduction

This section outlines the different technologies which will make up our system, divided into categories of use for the app, for the repeaters, and the diagnostics.

4.2 Technology and Software

4.2.1 Web App Technologies

- **Flask** - Framework for web development based on Python allowing for easy integration with our other Python technologies.
- **Plotly** - Plotly is a graphing technology we used with the python backend to graph scan positions on the web app.
- **Bootstrap Styles** - Bootstrap styles is a library used to quickly design responsive sites that we leveraged for our web app.

4.2.2 Diagnostic Technologies

- **Python** - Used network functions included with Python3 for gathering data on network coverage and performance.
- **iPerf** - Used for getting the packet retention rate, a built-in capability of iPerf.

4.2.3 General Technologies

- **GitHub** - This is a repository and version management system to enable code sharing and simultaneous changes to code bases.

4.3 Hardware

- **Raspberry Pi Zero W** - Inexpensive, Linux-based computers we used for the Access Points.
- **Raspberry Pi 4 Model B** - Slightly more expensive version of the Raspberry Pi used as the Diagnostic Node. We opted for the more expensive model here as the Diagnostic Node needs to run the trilateration and packet retention rate calculations.
- **CanaKit WiFi Dongle** - Used on the Raspberry Pi units to extend WiFi range and enable communication between the Meshed Access Point network and the Diagnostic Node.

Chapter 5

Web App Outline

5.1 User Interface

Pictured below is our web app for the user to interact with the system. Below the header is our heatmap, visible in the screenshot with a few configured diagnostic nodes in the bottom left and right and a few scanned points. In the top right of the heatmap are tools for interacting with the graph like zooming in and out or panning. Below the heatmap is a status/result string which is updated throughout use to let the user know what their next step in the process is. In the screenshot this status/result string says 'Scan complete,' indicating that the user has just scanned, and they now have the option to scan more points or use the clear function to start over. Below the status line are buttons for allowing the user to interact with the back end of the system. The four buttons are 'Clear,' 'Scan,' 'Calibrate Node 2,' and 'Calibrate Node 1.' This user interface was styled by linking with bootstrap css and using some of their classes for streamlining.

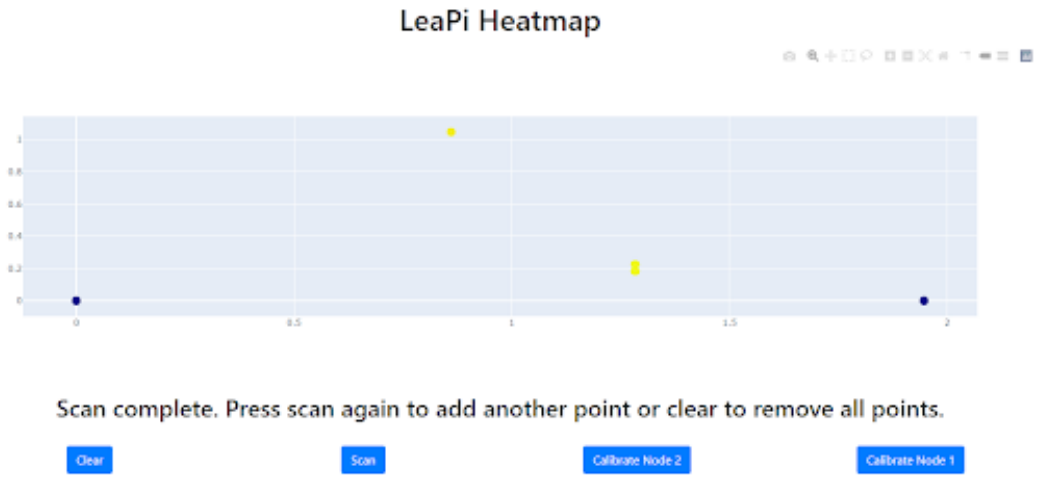


Figure 5.1: LeaPi web app user interface

5.2 Activity Diagram

Pictured below is the user's activity diagram when interacting with our LeaPi web app. They first begin at the far left on the 'Initial Startup' tile. Here the system will first ask the user to calibrate node two then node one if they have not been calibrated already. Then, the user will move to a point in real space they haven't yet scanned on the graph. Once they have done this they can click the 'Scan' button and add that point to the heatmap. At this step they then have the choice to repeat the previous two steps as required until their heatmap is populated as desired, or click the 'Clear' button to clear the heatmap and re calibrate the nodes.

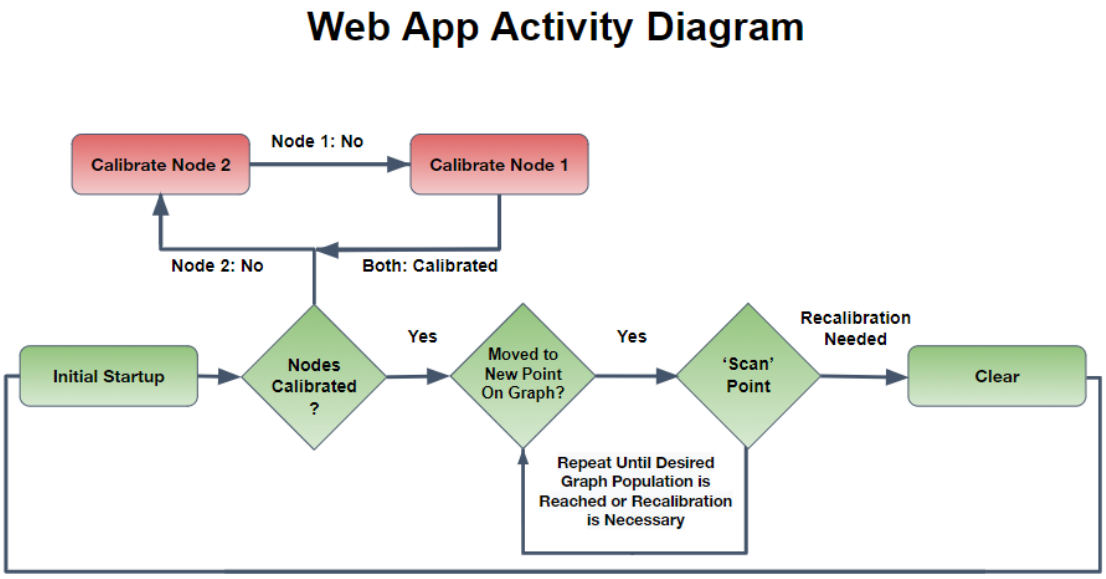


Figure 5.2: User's activity diagram for interacting with LeaPi web app

5.3 Implementation

5.3.1 Templates

In Flask we can utilize html templates via the Jinja2 template engine to render dynamic content. This template engine gives our html files capability beyond their traditional markup language limitations by interpreting our code rather than just compiling it. With this ability we can manipulate variables or update dynamic elements within a larger wrapper that stays static. At the time of serving the content to the user, these templates are then interpreted and translated to standard html for viewing in a browser. Our app consists of one such wrapper template called 'wrapper.html' which contains our header, result

string, and buttons in standard html format and styled with bootstrap as these are all static elements.

Within this wrapper file, between the 'LeaPi Heatmap' heading and the result string however, is a div with only the code "{% include 'index.html' %}". These escape characters, '{%', indicate to the template engine that we are no longer reading html but code to be interpreted. The next part which reads, "include 'index.html'", tells the Jinja2 engine to concatenate this 'index.html' file in the middle of our 'wrapper.html' file to the final output html served to the user's browser. The following escape characters, '%}' indicate to the compiler that we are back to reading html and it can continue directly printing it to the output html file served to the user after compiling, rather than breaking to interpret it. In our case, the 'index.html' file being rendered within our 'wrapper.html' is a graph generated by Plotly as mentioned in the previous chapter and updated accordingly by our main 'app.py' file that Flask uses to route the app's urls as discussed in the next subsection.

5.3.2 Main File - app.py

Pictured below this subsection in Figure 5.3 is part of our flask app's main file run at execution, 'app.py.' This part of the file shown defines the app's url routes that can be accessed and what alterations are performed to the graph or calibration after each button press by the user.

Route 1 - '/'

The first route defined, '/', defines what happens when the root page is requested. The app checks if a file called '.distance' exists, which would indicate that the distances between the nodes have already been calculated and therefore, and we already have calibration data on file. In this case the app renders our existing calibration data into an 'index.html' displayed in the wrapper template as previously discussed, indicating to the user with the result string that it has loaded their previous calibration and they may continue to scan or recalibrate.

Route 2 - '/calibrateNode2'

If the nodes have not yet been calibrated the app renders the template with our result string indicating the user needs to calibrate node 2 and then node 1. An obedient user will then click on the button to calibrate node 2, which sends a request to the route, '/calibrateNode2'. This route calls our triangulate python module's 'calibrate2()' function which will

determine the distance of sides 12 and 23, and renders the template with a new result string indicating the user now needs to calibrate node 1.

Route 3 - '/calibrateNode1'

If the user has not yet calibrated node 2 before node 1, the '/calibrateNode1' route's calibration will not work as triangulate's 'calibrate1()' function relies on side lengths passed from 'calibrate2()' to find node 3. Once this is finished the 'index.html' in the template is re-rendered with the result indicating to the user that their calibration is now complete and they are ready to scan.

Route 4 - '/scan/

If they then click the scan button, the '/scan' route is requested. This route calls our graph module's 'generate()' function, passing 'scan' as a parameter. This will take a list of x,y coordinates of our nodes as well as packet loss percentages to find our new point's location and color respectively. After generating a new Plotly 'index.html' graph file with the updated point(s), it is re-rendered within the 'wrapper.html' template alongside the new result string telling the user they now have the option to continue scanning until they'd like to clear the graph and recalibrate.

Route 5 - '/clear'

Finally, clicking the clear button which requests the '/clear' route. This route also calls 'generate()' from our graph module, however passing 'clear' as a parameter. After receiving this string as a parameter the 'generate()' function clears the graph of all points and the calibration data of all coordinates, re-renders the template with our newly emptied graph 'index.html', and tells the user in the result string that their graph has been cleared.

```python
@app.route('/')
def hello_world():
    if os.path.exists(".distance"):
        return render_template('wrapper.html', result='Previous calibration found, ready to
        scan.\nTo recalibrate please calibrate Node 2 and then Node 1')
    return render_template('wrapper.html', result='Welcome. Please Calibrate Node 2 and then Node 1')

@app.route('/clear')
def clear():
    generate('clear')
    return render_template('wrapper.html', result="Graph has been cleared.")

@app.route('/calibrateNode2')
def calibrateNode2():
    signal_2_1,signal_2_3 = calibrate1()
    return render_template('wrapper.html', result="Calibrated Node 2. Please calibrate Node 1.")

@app.route('/calibrateNode1')
def calibrateNode1():
    calibrate2()
    return render_template('wrapper.html', result="Calibration complete. Ready to scan.")

@app.route('/scan')
def scan():
    generate('scan')
    return render_template('wrapper.html', result="Scan complete. Press scan again to add another
    point or clear to remove all points.")
```

Figure 5.3: App's main route definitions found in our Flask 'app.py'

5.3.3 Serving the App

The Flask framework runs a server that makes the web app accessible after exporting the FLASKAPP variable, or the name of the file you want Flask to run. This file in our case was 'app.py', so before running the app we have to run the command "export FLASKAPP='app.py'". After this, running the command "flask run" in the directory with which you exported FLASKAPP will begin an instance of your app accessible at localhost through the default port 5000. In order to get an instance of the web app accessible by every machine on the network, you run the command "flask run –host=0.0.0.0". Running the 'flask run' command with this option included then starts a server that listens on your machine's IP to machines on the same WiFi network, again through the default port 5000. If you'd like to specify the port you wish to make the app accessible on, you could run the command "flask run –host=0.0.0.0 –port=2020" for example, which would start an instance of the app accessible to other machines on the same WiFi network through the port 2020 (as illustrated in figure 5.4 below).

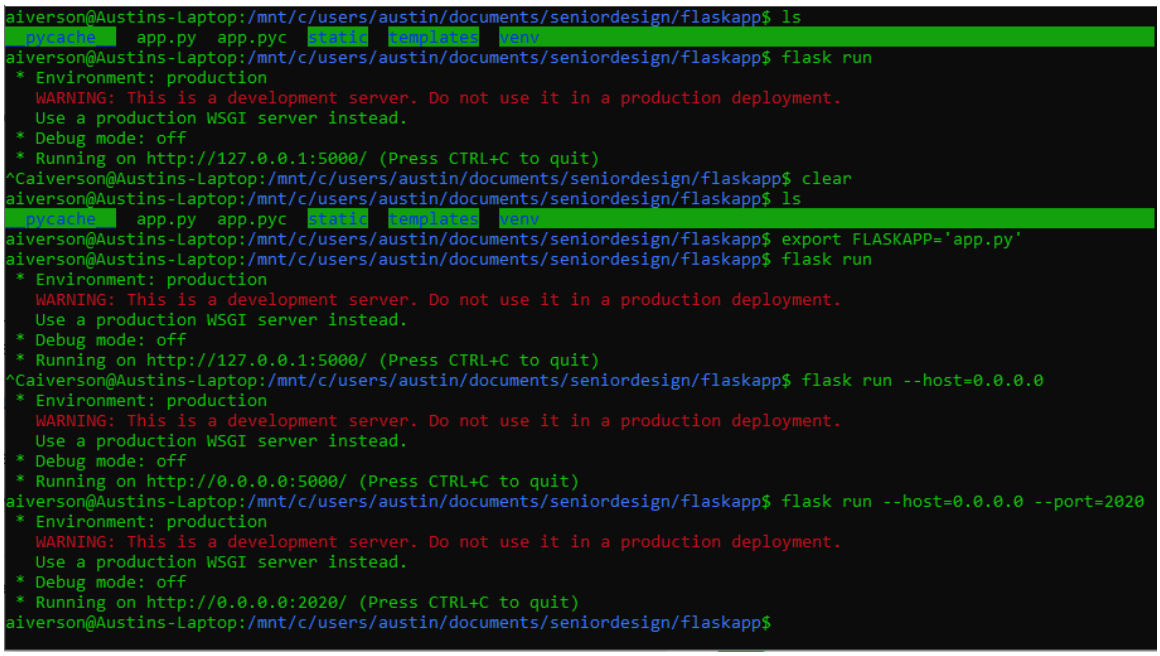


Figure 5.4: Running commands that start the app on various Flask server configurations

Chapter 6

Trilateration

6.1 Theory

Trilateration is essentially finding the intersection of three spheres. Each reference point represents the center of one of the spheres, and by finding the relative position of each reference point and the radius of each sphere, we can calculate their intersection. In our case, the radii of the spheres are the measured distances from some position x,y between the three points to each of the reference points.

6.2 Implementation

We used Raspberry Pi Zeros as our reference points, which are shown in Figure 3.1 below as Nodes 1-3. The Master Diagnostic Node was a Raspberry Pi 4 B that measured the RSSI level of each Pi Zero to find its relative distance from each node.

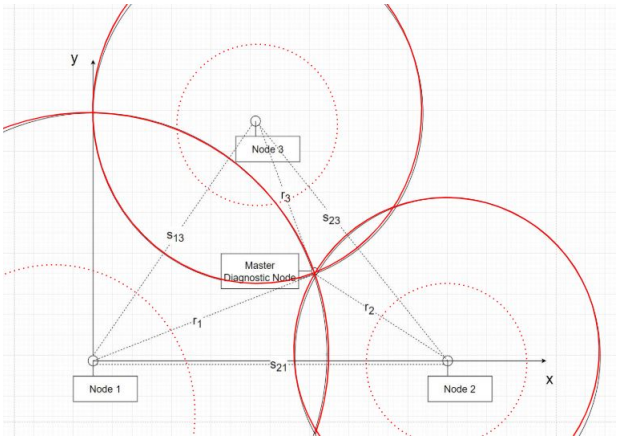


Figure 6.1: Highlighted spheres of RSSI around each node's antenna

Our first challenge was to find all three reference points. We designated Node 1 to be the

point (0,0), and we let the line between Node 1 and Node 2 define the x axis of our graph. As such, finding the position of Node 2 just involves finding the relative distance from Node 1 to Node 2.

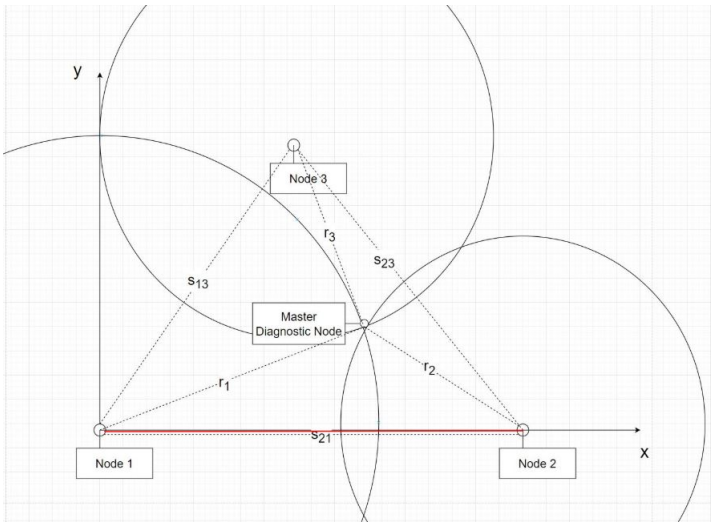


Figure 6.2: Finding position of Node 1 and Node 2

6.2.1 Distance Measuring

To accomplish this, we set the Master Diagnostic Node directly on top of Node 2 and then measured the beacon strength of Node 1. There are a number of reasons we chose this method over measuring RSSI between the two nodes directly. Firstly, if we measured RSSI using the nodes then each node would need to send its measured value to the Master Diagnostic Node (MDN), which would mean switching networks. We wanted to avoid having network changes in our project to ensure that the MDN could always stay connected to the main AP. Secondly, we wanted to avoid discrepancies in measured distance between the Raspberry Pi 4 B and the Raspberry Pi Zero. As long as the measured distance is always the RSSI level between a Pi Zero and a Pi 4, discrepancies due to different Wifi technologies and power levels can be mitigated. Once we measured the RSSI level, we applied the formula:

$$Distance = 2^{(-RSSI-25dBm)/N}$$

Where -25 dBm represents the maximum RSSI that can be achieved from placing the MDN directly on one of the reference points and N is some scaling factor to make distances easier to read and estimate.

6.2.2 Finding Node Positions

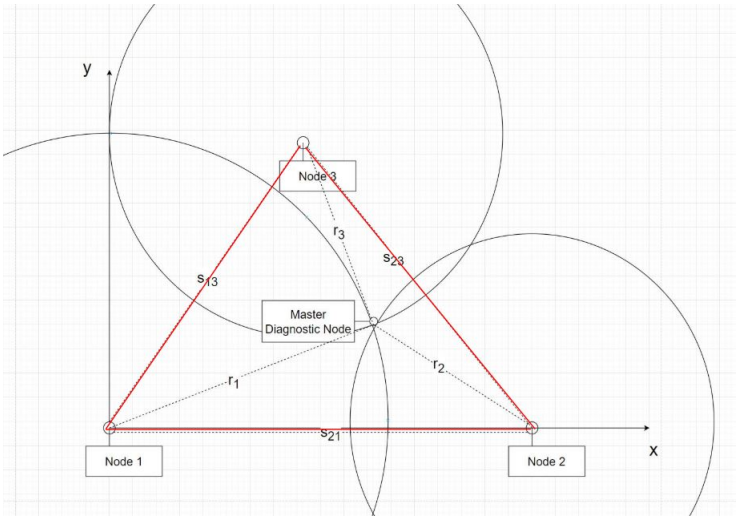


Figure 6.3: Need to find all three sides

We know that Node 1 is at (0,0) and Node 2 is at (s12,0). Now that we know how to measure each side of the triangle, we need to start calculating the relative position of Node 3, designated (V_x,V_y). This is the main challenge of the calibration phase, as we will need to measure all three sides of the triangle and then use the law of cosines to find $\measuredangle 1$:

$$cos(\measuredangle 1) = \frac{s_{13}^2 + s_{21}^2 - s_{23}^2}{2 s_{13} s_{21}}$$

We can then use our calculated value of $\measuredangle 1$ to find V_x and V_y:

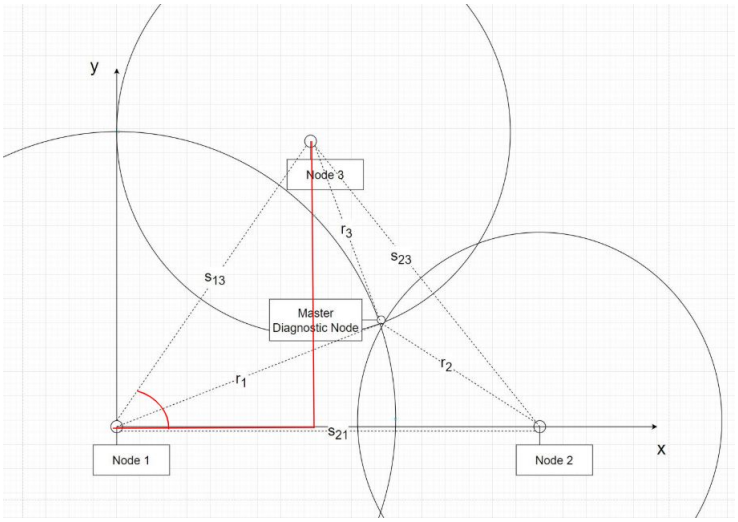


Figure 6.4: Use angle 1 to find V_x and V_y

$$V_x = s_{13}cos(\measuredangle 1)$$

$$V_y = s_{13}sin(\measuredangle 1)$$

6.2.3 Calculating Relative Position of Master Diagnostic Node

Now that we have all three center points, we can measure the radii of all three spheres and calculate the relative position of the Master Diagnostic Node anywhere inside the triangle. The radius of each sphere is simply the distance from its center point to the MDN. We

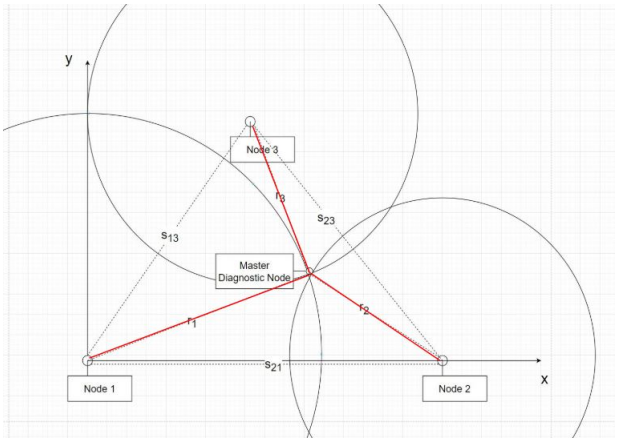


Figure 6.5: Measure distance from each node

can then plug our values into the following equations to find the coordinates of the Master Diagnostic Node:

$$x = \frac{r_1^2 - r_2^2 + s_{21}}{2s_{21}}$$

$$y = \frac{r_1^2 - r_3^2 + V_x^2 + V_y^2 - 2V_x x}{2V_y}$$

Because spheres are in three dimensions, we can also find the value of z for these coordinates, assuming all nodes are at z = 0:

$$z = \pm\sqrt{r_1^2 - x^2 - y^2}$$

This value is discarded in our implementation as superfluous.

Chapter 7

Network Configuration

7.1 Overview

Our network consists of five Raspberry Pi units. Each Raspberry Pi Zero is configured to operate on its own independent ad-hoc network with a unique SSID. The Raspberry Pi 4 Bs are both configured to operate on the main home network.

7.2 Configuring The Network

7.2.1 Main Diagnostic Tool

The diagnostic tool must be connected to the main AP over Wifi in order to measure network quality. To do this, you must change the options in the /etc/network/interfaces file to match the SSID and passkey of the main network. Note that the main network does not need to be connected to the internet. By default, this connection utilizes the wlan1 interface, which is the name given to an external Wifi dongle. The wlan0, or built-in Wifi interface, is used to measure the beacon quality of the positional nodes. This unit also uses the static IP 192.168.1.49 by default for connecting via SSH.

7.2.2 Positional Nodes

The three Raspberry Pi Zeros comprise the three positional nodes. Like the Diagnostic Tool their network settings are configured in the /etc/network/interfaces file. These do not include a built-in Wifi interface, so the external Wifi dongles are designated wlan0. These must broadcast three distinct network beacons, and are configured by default to operate on their own individual ad-hoc networks. The Main Diagnostic Tool then uses the beacon frames from those networks to measure distance. These are not connected to the main AP, but the user can connect to their ad-hoc networks directly and change their operation via

SSH. Note: By default, these networks are unsecured, but it is recommended that the user adds a password by manipulating the /etc/network/interfaces file.

7.2.3 Diagnostic Server

The Diagnostic Server is connected directly to the main AP using an Ethernet cable. This is done to ensure that any packet loss occurs solely between the main AP and the Diagnostic Tool. The AP will then assign the Diagnostic Server a unique IP address, which must be found manually using the AP's interface (usually found by navigating to 192.168.0.1 in a web browser). The user must then replace the IP on line 25 of /LeaPi/scripting/client/-surveyor_c_lite.sh in the Main Diagnostic Tool with the IP of the Diagnostic Server.

7.3 Table of Default Values

These are the default values for SSID and IP for each device. Changing these values should be done with caution. You must keep track of both the SSID and IP of each Positional Node to change their operation directly over SSH.

Table 7.1: Default SSIDs and IPs

Device	SSID	IP Address
Positional Node 1	LeaPiNode1	192.168.4.2
Positional Node 2	LeaPiNode2	192.168.4.3
Positional Node 3	LeaPiNode3	192.168.4.4
Main Diagnostic Node	None	192.168.1.49
Diagnostic Server	None	Determined by AP

Chapter 8

Network Quality Measurement

When it comes to selecting a metric for the network quality measurement, there are several options to consider. We will briefly discuss the pros and cons of each option below, followed by a description of how we implemented our chosen network quality measurement.

8.1 Options

8.1.1 Beacon Strength

One of the options we had was to use beacon strength, the strength of the connection from your device to the access point it is connected to, as our network quality measurement. This is a very obvious and simple solution, as it would be very quick and easy to implement; however, there are a number of things wrong with this approach, the most important of which are laid out as follows:

1. Beacon strength falls rapidly when passing through objects such as walls or furniture, so the true network quality may be falsely represented

2. Beacon strength does not provide any information on whether or not the connection is stable enough to send packets at a steady rate, or what that rate is, which is what we are actually concerned with

Due to these crippling factors, we quickly decided to abandon the idea of using beacon strength as the network quality measurement and looked elsewhere.

8.1.2 Packet Reception Rate (PRR)

The next possible metric we came up with was the packet reception rate, abbreviated as PRR. PRR is a good metric for use as the network quality measurement because it provides a numerical value describing the percentage of monitored test packets sent from a client device that reach their destination, a server device, undamaged. There is one problem with using solely PRR as our network quality measurement, however – PRR simply provides the percent of packets that safely arrived, but does not report the transfer speed on the network card itself. Modern WiFi devices have a built-in safety measure to ensure that data being transferred arrives undamaged. If the receiver device observes some corruption or packet loss (dropped packets that never reach their destination), it can reduce the data transfer bandwidth on the card in order to achieve a better PRR. As these corrections happen in real time, immediately once the networking device notices the errors in packets, it is possible that the PRR will only be mildly affected while the actual network quality is lower at that location. Therefore, while PRR handle most conditions, we need another metric to drive the solution the rest of the way home.

8.1.3 Physical-Layer Bitrate

The physical-layer bitrate is the data transfer bandwidth on the card that we mentioned above; as was stated, it is sometimes reduced at the receiver device's network card's discretion to improve PRR at the cost of transfer speed. When comparing this to the maximum rated transfer speed of the WiFi device, we can see if it was reduced to inflate PRR. However, we cannot use the physical-layer bitrate alone as it is not always reduced in situations where severe packet loss will occur.

8.1.4 Final Decision

We decided to first implement PRR, which has the widest and most useful coverage of the three aforementioned options, and then implement the physical-layer bitrate if there was enough time left. Unfortunately, there was not enough time left to implement this after we got the rest of the project working, so including the physical-layer bitrate in our network quality measurement was moved to future work. Our final design for this project uses PRR alone as the network quality measurement, which has been shown to handle the vast majority of network testing conditions (see the Evaluation chapter for more information).

8.2 Implementation

Our implementation of measuring the PRR, using iPerf, can be found in Code Appendix Section 2. A brief explanation of how iPerf works and how we get the PRR from iPerf's output is as follows:

1. iPerf server command runs on the destination side – in the case of this project, that's the Diagnostic Server
2. iPerf client command runs on the source side – this runs on the Diagnostic Node
3. iPerf then sends a set number of packets from the Diagnostic Node to the Diagnostic Server, which then confirms the packets that arrived damaged
4. iPerf reports the percentage of damaged packets out of the total number of transmitted packets to the user on the client side – this is called packet loss
5. We subtract the packet loss from 100% to get PRR, which is then handed off to the back-end of the web interface for processing and display

8.3 Possible Improvements

Our results, discussed in the following Evaluation chapter, show that PRR is a good network quality measurement in our various testing conditions. That said, we definitely recognize the potential of comparing the PRR with the physical-layer bitrate in order to have a more accurate and holistic view of the conditions of the WiFi network. While it is unfortunate that this feature had to be moved to future work, when it is completed, our network quality measurement will work in not just most, but all testing conditions.

Chapter 9

Evaluation

9.1 Trilateration Effectiveness

9.1.1 Setup

To measure the effectiveness of our trilateration method and to get an idea of the tendencies of our algorithm, we placed a six test points on the graph, as shown by the circled points in Figure 9.1.

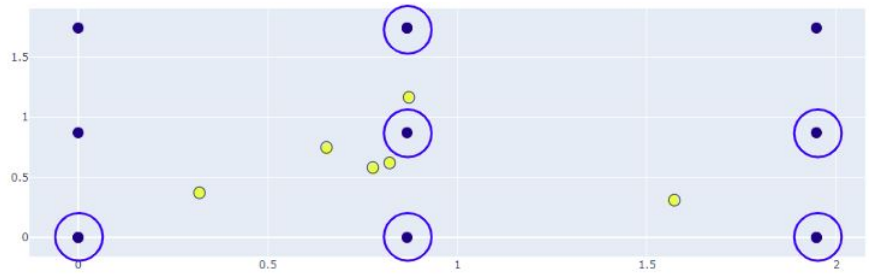


Figure 9.1: Six test points

We then placed the Master Diagnostic Node at points in the room that corresponded to each test point and ran our trilateration algorithm to compare our expected values to our observed values. Our measured points are circled in Figure 9.2.

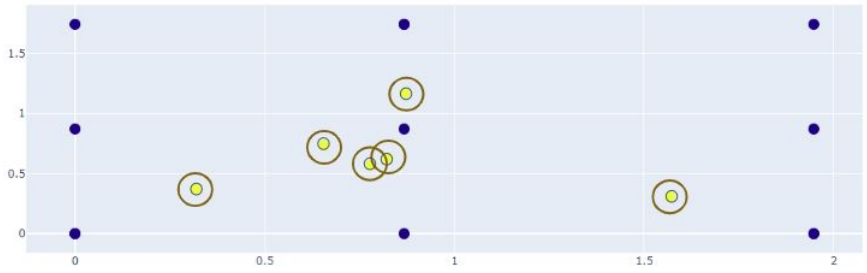


Figure 9.2: Six observed points

9.1.2 Results

After scanning all six points, it was clear that our trilateration method was not entirely accurate. A vast majority of the observed points were off by a clear margin. However, we were able to observe some patterns in the way that points were being calculated. One

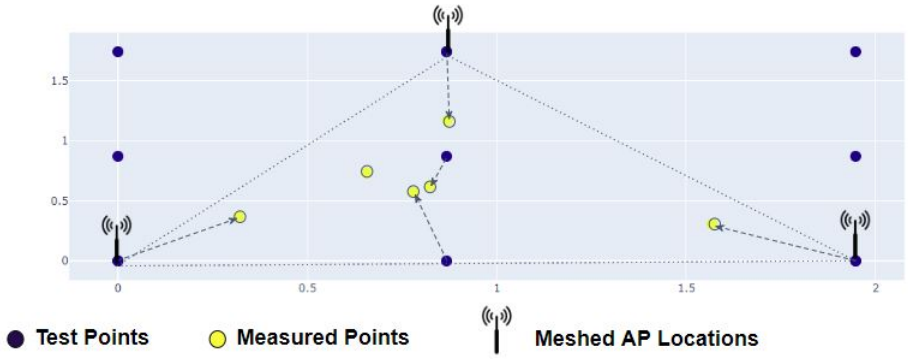


Figure 9.3: Comparing expected points to observed points

pattern that emerged is the tendency for the measured points to deviate toward the center of the triangle, indicating that our maximum RSSI level may have been higher than originally estimated. This tendency is shown in Figure 9.3, where test points inside and at the edge of the triangle had a tendency to appear close to the center of the triangle but still generated reasonable values.

However, when measured outside the triangle, our algorithm generated entirely incorrect results. Figure 9.4 on page 25 shows that the measured values outside the triangle have a particular tendency for error. This is one of the limitations of our implementation, as locations outside the triangle of beacon nodes cannot be measured accurately. One way to mitigate this issue would be to add additional beacon nodes, and to use the closest three beacon nodes as the vertices of the triangle. However, this would be much more difficult to

implement, and so for the scope of our project we decided that our range was adequate.

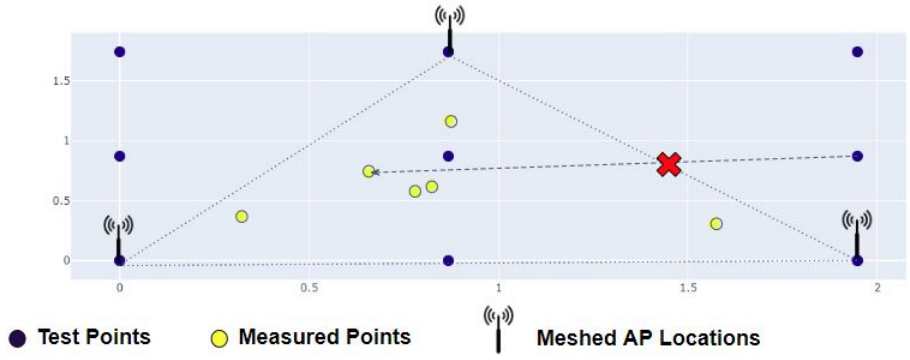


Figure 9.4: Comparing expected points to observed points

9.2 Network Quality Measurement Effectiveness

We tested our system in as much of a variety of room layouts as possible; however, given the nature of the shelter-in-place order currently active, our system was constrained to the rooms and environment within one house. That said, we tested several cases, including through varying amounts of pieces of furniture and walls. As expected, the PRR generally dipped as we placed more objects between the access point and the Diagnostic Node, so this was a good metric for network coverage. On the other hand, we did notice that PRR did not always drop enough to signify poor coverage when it was evident that the ability of the router to quickly deliver packets was hampered. For example, when we tested through several walls, we still got a slightly PRR, but the data was coming through very slowly. We are confident that this is because the WiFi card cut the physical-layer bitrate down to compensate for the packets that were coming in damaged or were lost entirely; therefore, from these results, we determined that physical-layer bitrate must be present for this product to be useful in all housing environments. Additionally, we believe physical-layer bitrate will add an extra layer of accuracy to our network quality measurement, as it will become more evident how badly each object between the router and the device affects the coverage. We have placed it in our list of future work for these reasons.

Chapter 10

Future Work

10.1 Improving User Experience

10.1.1 Network Settings UI

We plan to create a UI for configuring network settings in the future. This interface would allow the user to fine-tune the product to their purposes. This falls in line with our vision for the project from the beginning, as we want to create a product that is expandable, customizable, and accessible for all users. Users will also be able to enable an advanced mode, such that the more technical users will be able to access the raw network quality data and parse it themselves.

10.1.2 WiFi Extender Placement Suggestions

We plan to use the data we already collect to suggest to the user where WiFi extender device are necessary in their network. Additionally, since many people use IoT devices in their homes (such as WiFi cameras and smart doorbells), this would ensure that the IoT devices receive the WiFi coverage that they need to keep the user's home safe.

10.2 Improving Functionality

10.2.1 Network Quality Tests

We plan to add more network quality tests, such as physical-layer data rate, in order to give the user a more accurate coverage measurement at each location. The measurements from these additional quality tests would be combined via an algorithm into one value that the user would easily be able to understand.

10.2.2 Passive Network Monitoring

One functionality we planned to include from the beginning is passive network monitoring via the beacon nodes. This functionality will allow the system to detect and record interference, notifying the user of patterns of interference based on time of day and allowing the user to diagnose why their network seems to work better at certain times of the day; from here, a user can take action to change their network frequency and/or disable the device causing the interference.

10.2.3 Saving and Loading Graphs

A simple capability we plan to include in a future update is the option to save and load graphs. This allows the user to recall data from previous runs and directly compare it to current runs, an essential tool in comparing the coverage of different WiFi configurations and extender placements.

10.2.4 Expand Trilateration Range

Perhaps one of the most crucial updates to this project is to expand the operational range of the triangulation process. As you saw in the evaluation section, scan points outside the area of the triangle formed by the Meshed Access Points have grossly inaccurate position readings. We are prioritizing the expansion of this effective range to eliminate the resulting error in positioning.

Chapter 11

Considerations

11.1 Ethical

From the outset our project brought ethics onto the table of discussion in terms of privacy concerns as we wanted to make it one of our goals that no user data was at risk due to using our network diagnostic tool. We were able to mitigate this concern by making the flask server only accessible to machines on the same WiFi network as the host at run time. This essentially makes the network tool as safe as your WiFi network.

11.2 Health and Safety

A potential health and safety consideration for our project are the possible adverse health effects of bringing additional wireless devices into the home, but any studies of wireless signals and their effects on our health good or bad have been inconclusive.

11.3 Manufacturability

All parts used for our project were pre-manufactured and we purchased them from vendors. Users would also purchase them from a vendor so manufacturability is not much of a concern as everything is pre-manufactured.

11.4 Social

As a social consideration, access to the internet these days is almost a necessity and the quality of our internet access is always improving with society, and vice versa. Our project's main goal is to increase the quality of internet access the average consumer is capable of receiving.

11.5 Environmental

An environmental concern of our project is the potential harm components like raspberry pi's can cause after improper disposal. This is because their components contain metals and plastics which can harm the surrounding ecosystem. These concerns can be mitigated, however, by e-wasting the components when the user is done with them.

11.6 Economic

Economics was one of our main considerations from the outset as well as we were discouraged by the existing economic barriers for entry into the mesh network and diagnostics market. This is why we devised our solution using raspberry pis, some of the world's most inexpensive computers, in order to mitigate this concern.

11.7 Sustainability

Our project is fairly sustainable as it is easily re-configurable, and none of the components are consumed during use. This then means it can be utilized for its intended use an essentially infinite number of times. In addition if the user is done with our product for whatever reason, since the main components are raspberry pi's they could re-purpose them for another project if they are inclined to do so rather than e-wasting them.

11.8 Political

Lastly, since our project has no real political impact this is not much of a concern.

Chapter 12

Conclusion

The following conclusion is a recap of our main points.

12.1 Problem and Solution Addressed

The range of a WiFi network is affected by many factors like antenna type, its environment like walls or furniture, and interference like other 2.4Ghz bands or microwaves. It can be hard to tell what your network's range is exactly, or why something might not be working due to the difficulty of troubleshooting mesh networks. There are currently diagnostic tools on the market however they can start around five hundred dollars and get up to thousands of dollars, making them not too appealing or even attainable to the average consumer. To address this problem we proposed an inexpensive and easy to use diagnostic tool by using inexpensive hardware and making it easily configurable. Our diagnostic tool measures packet loss, determines the position of the user, and utilizes this information to create a heat map of WiFi strength for easier network configuration, quality improvement, and troubleshooting.

12.2 System Overview and Technology

Our system's three components are access points, diagnostic nodes, and a diagnostic server. The access points include both the main access point emitting the network and our meshed access points. The diagnostic node is the main point of user interaction where the web server is hosted and calculates the position of the user using trilateration based on RSSIs of the meshed access points. The diagnostic server sends packets to the master diagnostic node used to calculate packet loss and is hard wired into the router to eliminate errors. The access points are Raspberry Pi Zero Ws, the diagnostic nodes are Raspberry Pi 4 Model Bs, and they are all communicating with CanaKit WiFi dongles. Our app and

server run on flask and was styled with bootstrap, our graph runs on Plotly, our network statistics run on iPerf, and we used GitHub for version control.

12.3 Evaluation of our Present and Future

To evaluate the effectiveness of our trilateration we measured predetermined test coordinates in real life and compared these expected coordinates to the results of our software. The results showed that we did pretty well, however some measurements tend towards the center of the triange, and points outside of the triangle are not accurately measured. In the future, we could do with improving the user's experience. This could be done by creating a UI for configuring the actual network settings themselves, adding the option to save or load previous graphs, as well as utilizing our heatmap to implement suggestions for the placement of IoT devices and WiFi extenders. We could also improve functionality by adding more network quality tests, passively monitoring the network, and expanding the range of trilateration.

Appendices

.1 Python Code

.1.1 Finding Signal Level (signalLevel.py)

```
import os
import subprocess

OOR = "Out␣of␣Range"
ips = ['192.168.4.2', '192.168.4.3', '192.168.4.4']
addresses = ['LeaPiNode1', 'LeaPiNode2', 'LeaPiNode3']
interface = 'wlan0'
surveyor_location = '../scripting/client/surveyor_c_lite.sh'

def getPacketLoss(bflag_value = 5, lflag_value = 1470):

    bflag_value = str(bflag_value)
    lflag_value = str(lflag_value)
    return 1
#Ignoring this code for now because I forgot the server at
   home
    runCommand = ['./' + surveyor_location, '-b',
       bflag_value, '-l', lflag_value]

    scan = subprocess.Popen(runCommand, stdout = subprocess.
       PIPE, stderr = subprocess.PIPE)
    stdout,stderr = scan.communicate()
    value = stdout.decode()
    print(value)

    return float(value)


def getSignal(essid):
    scanCommand = ['sudo', 'iwlist', interface, 'scan']
    lines = ['␣','␣','␣','␣']
    Signal = {
        'strength': 0,
        'address': essid
    }
    print("Scanning...")

    scan = subprocess.Popen(scanCommand,stdout = subprocess.
       PIPE, stderr = subprocess.PIPE)
    stdout,stderr = scan.communicate()
```

```
    lines = stdout.decode().split('\n')

    i = 0
    for line in lines:
        if(essid in line):
            Signal = {
                'strength': lines[i - 2].split('=')[2].split
                   (' ')[0],
                'address': essid
            }
        i = i + 1

    return Signal

def getAllSignals():

    all_signals = []
    for add in addresses:
        signal = getSignal(add)
        all_signals.append(signal)
    return all_signals

if __name__ == '__main__':
    print(getAllSignals())
```

.1.2 Trilateration (triangulate.py)

```
from flask import render_template
import signalLevel as sig
import os, math, time

distance_file = '.distance'
node1 = 'LeaPiNode1'
node2 = 'LeaPiNode2'
node3 = 'LeaPiNode3'

def signal_to_distance(signal):
    return pow(2, (-signal - 25 + 0.1)/36)


def calibrate1():
    signal_2_1 = 0.0
    signal_2_3 = 0.0


    signal_2_1 = (float(sig.getSignal(node1)['strength']) +
       float(sig.getSignal(node1)['strength'])) / 2
    print(signal_2_1)
    signal_2_3 = (float(sig.getSignal(node3)['strength']) +
       float(sig.getSignal(node3)['strength'])) / 2

    signal_2_1 = signal_to_distance(signal_2_1)
    signal_2_3 = signal_to_distance(signal_2_3)

    with open(".sig_2_1", "w") as f:
        f.write(str(signal_2_1))

    with open(".sig_2_3", "w") as f:
        f.write(str(signal_2_3))

    return signal_2_1,signal_2_3


def calibrate2():
    signal_1_3 = 0.0

    if os.path.exists(".sig_2_1"):
        with open(".sig_2_1", "r") as f:
           signal_2_1 = f.read()
        signal_2_1 = float(signal_2_1)
```

```
        with open(".sig_2_3", "r") as f:
            signal_2_3 = f.read()
        signal_2_3 = float(signal_2_3)
    else:
        print("Need to calibrate Node 2 First")
        return
    signal_1_3 = (float(sig.getSignal(node3)['strength']) +
       float(sig.getSignal(node3)['strength'])) / 2

    signal_1_3 = signal_to_distance(signal_1_3)

    cos_1 = (signal_1_3 * signal_1_3 + signal_2_1 *
       signal_2_1 - signal_2_3 * signal_2_3) / (2 *
       signal_1_3 * signal_2_1)

    if cos_1 > 1 or cos_1 < -1:
        print('Calculated values do not form triangle, try
           again')
        return

    node_3_x = signal_2_3 * cos_1
    print(cos_1)
    node_3_y = signal_2_3 * math.sin(math.acos(cos_1))

    distance_info = (node_3_x,node_3_y,signal_2_1)

    with open(distance_file, "w") as f:
        f.write(str(distance_info))

    return distance_info


def calibrate():
    response = ''
    response = input("Place Diagnostic Tool at Node 2, then
       press enter...")

    signal_2_1 = 0.0
    signal_2_3 = 0.0
    signal_1_3 = 0.0

    signal_2_1 = (float(sig.getSignal(node1)['strength']) +
       float(sig.getSignal(node1)['strength'])) / 2
    print(signal_2_1)
```

```
    signal_2_3 = (float(sig.getSignal(node3)['strength']) +
       float(sig.getSignal(node3)['strength'])) / 2
    print(signal_2_3)

    response = input("Place␣Diagnostic␣Tool␣at␣Node␣1,␣then␣
       press␣enter...")

    signal_1_3 = (float(sig.getSignal(node3)['strength']) +
       float(sig.getSignal(node3)['strength'])) / 2
    print(signal_1_3)

    signal_2_1 = signal_to_distance(signal_2_1)
    signal_2_3 = signal_to_distance(signal_2_3)
    signal_1_3 = signal_to_distance(signal_1_3)
    print("Distances␣2-1,␣2-3,␣1-3:")
    print(signal_2_1, signal_2_3, signal_1_3)
    cos_1 = (signal_1_3 * signal_1_3 + signal_2_1 *
       signal_2_1 - signal_2_3 * signal_2_3) / (2 *
       signal_1_3 * signal_2_1)

    if cos_1 > 1 or cos_1 < -1:
        print('Calculated␣values␣do␣not␣form␣triangle,␣try␣
           again')
        return

    node_3_x = signal_2_3 * cos_1
    print(cos_1)
    node_3_y = signal_2_3 * math.sin(math.acos(cos_1))

    distance_info = (node_3_x,node_3_y,signal_2_1)

    with open(distance_file, "w") as f:
        f.write(str(distance_info))

    return distance_info


def triangulate():
    if(os.path.exists(distance_file)):
        with open(distance_file, "r") as f:
            distances = f.read()
        V = distances.replace('(', '').replace(')', '').
           replace('␣', '').split(',')
        Vx = float(V[0])
        Vy = float(V[1])
```

```
        U = float(V[2])
    else:
            V = calibrate()
            Vx = V[0]
            Vy = V[1]
            U = V[2]

    print(Vx, Vy, U)
    r1 = signal_to_distance(float(sig.getSignal(node1)["
       strength"]))
    r2 = signal_to_distance(float(sig.getSignal(node2)["
       strength"]))
    r3 = signal_to_distance(float(sig.getSignal(node3)["
       strength"]))

    x = (r1*r1 - r2*r2 + U*U)/(2*U)
    if (x < 0):
        x = 0
    y = (r1 * r1 - r3 * r3 + Vx * Vx + Vy * Vy - 2 * Vx * x)
        / (2 * Vy)
    return x,y,U

if __name__ == '__main__':
    x,y,U = triangulate()
    print(x)
    print(y)
```

.1.3 Generating Graph (graph.py)

```
from triangulate import triangulate, calibrate
from signalLevel import getPacketLoss
import plotly.graph_objects as go
from datetime import datetime
import os, pickle, random

saved_graph = '.graphInfo'
graph_location = './templates/index.html'

def generate(response):
    U = 0
    graphInfo = {
                'x_vals': [],
                'y_vals': [],
                'z_vals': []
                }

    if os.path.exists(saved_graph):
        with open(saved_graph, 'rb') as f:
            graphInfo = pickle.load(f)

    # response = input("(1) Press Enter to scan\n(2) Type \"
       calibrate\" to recalibrate distance\n(3) Type \"clear
       \" to clear previous scan results\n(4) Type \"quit\"
       to exit\n")
    if response == 'calibrate' or response == '2':
        U = calibrate()[2]
        response = 'clear'

    if response == 'clear' or response == '3':
        graphInfo = {
                'x_vals': [],
                'y_vals': [],
                'z_vals': []
                }
        print("Graph being reset...")

    if response == 'quit' or response == '4':
        return

    if response is not 'clear' and response is not '3':
        x,y,U = triangulate()
```

```
    if response is not 'clear' and response is not '3':
        z = getPacketLoss()

    if len(graphInfo['x_vals']) == 0:
        graphInfo['x_vals'].append(0)
        graphInfo['y_vals'].append(0)
        graphInfo['z_vals'].append(0)
    elif len(graphInfo['x_vals']) == 1:
        graphInfo['x_vals'].append(U)
        graphInfo['y_vals'].append(0)
        graphInfo['z_vals'].append(0)

    if response is not 'clear' and response is not '3':
        graphInfo['x_vals'].append(x)
        graphInfo['y_vals'].append(y)
        graphInfo['z_vals'].append(z)
    print(graphInfo)

    size_of_markers = 12
    fig = go.Figure()
    fig.add_trace(go.Scatter(x = graphInfo['x_vals'], y =
       graphInfo['y_vals'], mode="markers", marker = dict(
       color=graphInfo['z_vals'], size=size_of_markers)))
    refresh_interval = 7
    reload_script = 'window.onload=function()␣{window.
       setInterval(function()␣{window.location.reload(true)
       },␣1000␣*␣' + str(refresh_interval) + ')}'
    fig.write_html(graph_location, auto_open = False)

    with open(saved_graph, 'wb') as f:
        pickle.dump(graphInfo, f)

if __name__ == '__main__':
    while(1):
        response = input("(1)␣Press␣Enter␣to␣scan\n(2)␣Type␣
           \"calibrate\"␣to␣recalibrate␣distance\n(3)␣Type␣
           \"clear\"␣to␣clear␣previous␣scan␣results\n(4)␣
           Type␣\"quit\"␣to␣exit\n")
        generate(response)
        if response == '4':
            break
```

.1.4 Web App (app.py)

```
from flask import Flask
from flask import render_template
from graph import generate
from triangulate import calibrate1, calibrate2
from flask_bootstrap import Bootstrap
import os
app = Flask(__name__)
app.config.update (
    TEMPLATES_AUTO_RELOAD = True,
    EXPLAIN_TEMPLATE_LOADING = True
)

signal_2_1 = 0.0
signal_2_3 = 0.0
signal_1_3 = 0.0

#heatmap page w scan, clear, calibrate
@app.route('/')
def hello_world():
    if os.path.exists(".distance"):
        return render_template('wrapper.html', result='
           Previous␣calibration␣found,␣ready␣to␣scan.\nTo␣
           recalibrate␣please␣calibrate␣Node␣2␣and␣then␣Node
           ␣1')
    return render_template('wrapper.html', result='Welcome.␣
       Please␣Calibrate␣Node␣2␣and␣then␣Node␣1')

@app.route('/clear')
def clear():
    generate('clear')
    return render_template('wrapper.html', result="Graph␣has
       ␣been␣cleared.")

@app.route('/calibrateNode2')
def calibrateNode2():
    signal_2_1,signal_2_3 = calibrate1()
    return render_template('wrapper.html', result="
       Calibrated␣Node␣2.␣Please␣calibrate␣Node␣1.")

@app.route('/calibrateNode1')
def calibrateNode1():
    calibrate2()
    return render_template('wrapper.html', result="
```

```
        Calibration␣complete.␣Ready␣to␣scan.")

@app.route('/scan')
def scan():
    generate('scan')
    return render_template('wrapper.html', result="Scan␣
        complete.␣Press␣scan␣again␣to␣add␣another␣point␣or␣
        clear␣to␣remove␣all␣points.")
```

.2 Network Quality Measurement Scripts

.2.1 Client (surveyor_c_lite.sh)

```
#!/bin/bash

PKT_LEN=1470
BANDWIDTH=5

for arg in "$@"; do
        case $arg in
        '-l'|'--length')
                PKT_LEN=$2
                shift
                shift
                ;;
        '-b'|'--bandwidth')
                BANDWIDTH=$2
                shift
                shift
                ;;
        *)
                echo "You␣have␣entered␣an␣invalid␣argument."
                exit
                ;;
        esac
done

iperf -c 192.168.1.4 -u -b ${BANDWIDTH}m -t 3 -i 3 --len ${
   PKT_LEN} > temp.txt

line=$( tail -n 1 temp.txt )
# echo $line >> log.txt

IFS='(' read -ra ADDR <<< "$line"
IFS='%' read -ra LOSS <<< "${ADDR[1]}"
PKTLOSS=${LOSS[0]}
PRR=$( bc <<< "scale=2;100-${PKTLOSS}" )

rm temp.txt

echo $PRR

# done
#
```

```
# # calculating average packet retention rate over the four
   scans
# AVGPRR=$( bc <<< "scale=2;(${SCANS[0]}+${SCANS[1]}+${SCANS
   [2]}+${SCANS[3]})/4" )
#
# MAX=$( printf '%s\n' "${ARRAY[@]}" | awk '$1 > m || NR ==
   1 { m = $1 } END { print m }' )
# MIN=$( printf '%s\n' "${ARRAY[@]}" | awk '$1 < m || NR ==
   1 { m = $1 } END { print m }' )
# RANGE=$( bc <<< "scale=2;$MAX-$MIN" )
```

.2.2 Server (surveyor_s.sh)

```
#!/bin/sh

ctrl_c () {
        printf "\nCleaning up...\n"
        [ -z $1 ] && rm log.txt
        echo "Done!"
}

echo "Server started. Use ^C (INTERRUPT) to end."

trap ctrl_c INT

iperf -s -u -i 1 >log.txt
```

.3 HTML Template

.3.1 Graph Wrapper (wrapper.html)

```
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="https://stackpath.
       bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.
       css" integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/
       Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
       crossorigin="anonymous">
    <style>
      form {display: inline;}
    </style>
  </head>

  <body>
    <div class="containter" id ='heatmap'>
      <div class = "row␣justify-content-center">
        <h1 class = "col-xs-12" style= 'padding-top:25px;' >
          LeaPi Heatmap</h1>
      </div>

      <div class = "row␣justify-content-center">
        <div class = "col-12">
          {% include 'index.html' %}
        </div>
      </div>

      <br>
      <div class = "row␣justify-content-center">
        <h2 class = "col-xs-12" style = "padding-bottom:␣25
          px;">{{result}}</h2>
      </div>

      <div class = "row␣justify-content-center">
        <div class = "col-3" align="center">
          <form action='/clear' method="GET">
            <input class = "btn␣btn-primary" type='submit'
              value='Clear'>
          </form>
        </div>

        <div class = "col-3" align="center">
```

```
          <form action='/scan' method="GET">
            <input class = "btn␣btn-primary" type='submit'
               value='Scan'>
          </form>
        </div>

        <div class = "col-3" align="center">
          <form action='/calibrateNode2' method="GET">
            <input class = "btn␣btn-primary" type='submit'
               value='Calibrate Node 2'>
          </form>
        </div>

        <div class = "col-3" align = "center">
          <form action='/calibrateNode1' method="GET">
            <input class = "btn␣btn-primary" type='submit'
               value='Calibrate Node 1'>
          </form>
        </div>
      </div>
    </div>
  </body>
</html>
```