# AREALYTICS

Daniel Gonzalez, Avery Lamp, Ethan Weber, and Moin Nadeem

## Our Team

### Ethan Weber
Hardware, Microcontrollers, Software, Web
http://ejweber.scripts.mit.edu/

### Avery Lamp
Software, iOS Developer, Web
http://averylamp.me/PortfolioLinks.pdf

### Daniel “Gonzo” Gonzalez
Design, Software, Hardware, Web
http://gonzo.mit.edu/resume.pdf

### Moin Nadeem
Software, Systems, Web
http://moinnadeem.com/

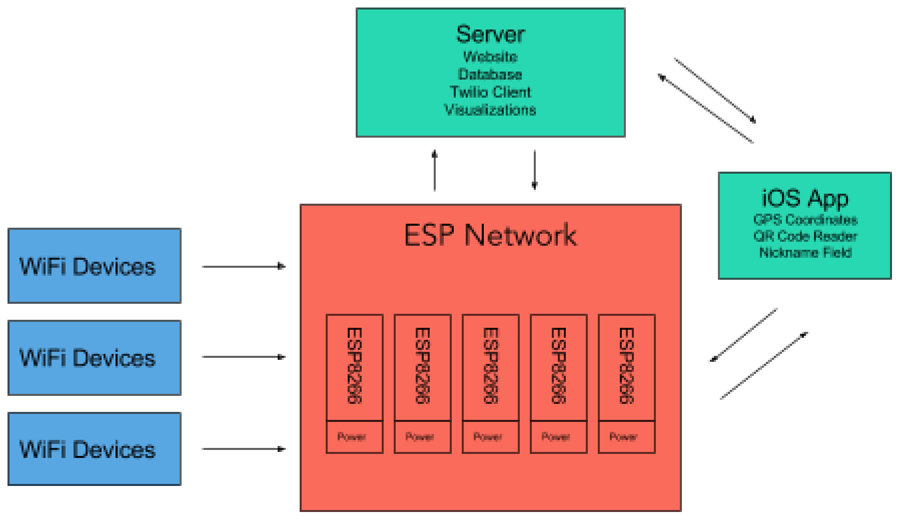
# An Introduction

For places that have a large flow of human traffic throughout the days and nights, such as universities and convention centers, it's helpful for people running those institutions to have a good idea of where people travel most and how they move in certain times of the days. There was no efficient, simple way to accomplish this task until Arealytics came to life. Now we are able to passively track people's devices throughout various buildings and spaces. We can display this information on a dashboard system tailored towards a customer's needs: real-time data about customers, max flow rates of walkways, location hotspots, and we are currently leveraging the millions of data points available to create a predictive analytics solution.

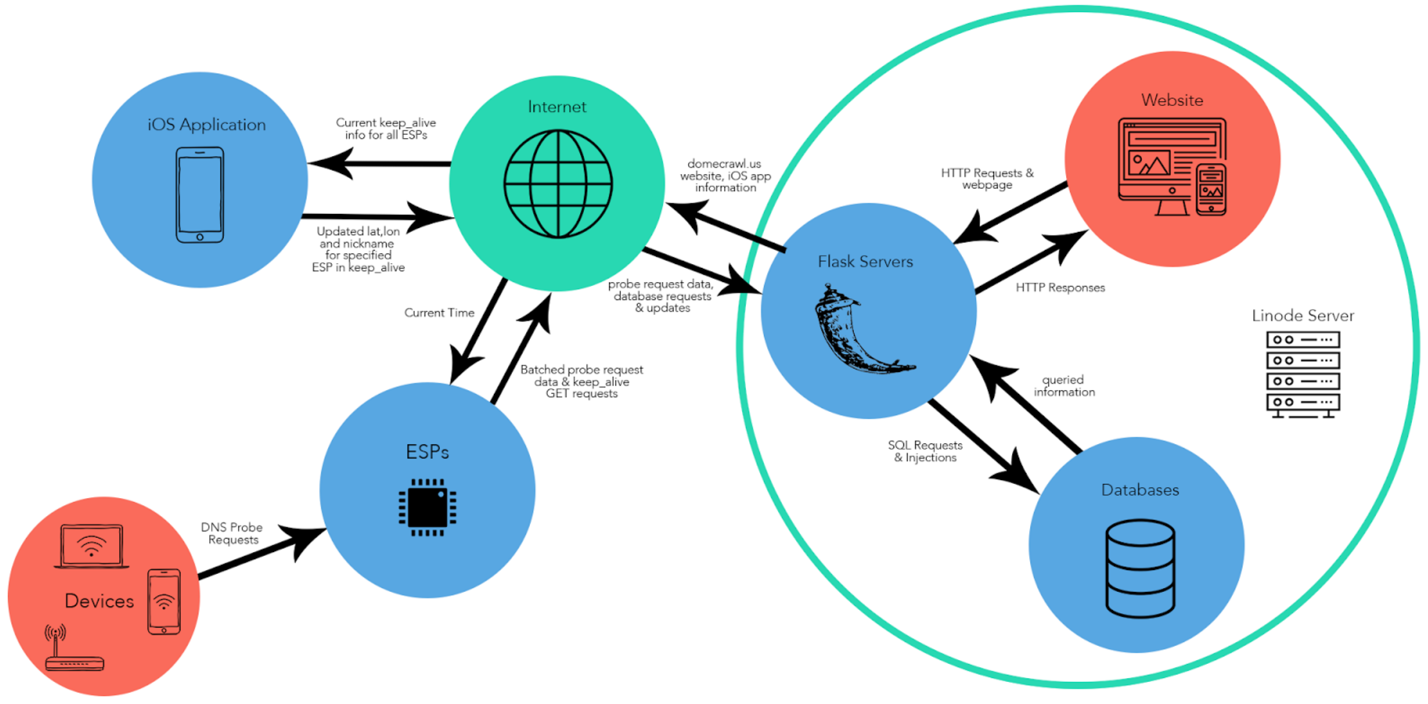
How we accomplish this task is simple: we have reconfigured WiFi chips into a network of microcontrollers that detect DNS Probe Requests. DNS probe requests are essentially signals that WiFi devices broadcast out to the routers nearby with their MAC address saying, "Hey, I'm here! Allow me to build a connection with you." We store all of these probe requests by logging MAC addresses, timestamps, signal strengths, and more. With this information, we can model the paths that people took throughout the day. We will also provide vital information to businesses and institutions about where to create new working spaces, where to perform construction, and where to relocate sites to optimal locations.

## System Infrastructure Diagram

This is a diagram depicting the major components of our system. We have a network of ESP8266 microcontrollers with power sources located within some geographical region. We call these modules ESPs. These devices connect to WiFi access points nearby and continually send the data to our server. The server then processes all of the data to create dynamic visualizations. The iOS application acts a bridge between the ESPs and the server allowing us to easily deploy the ESPs in locations and update the database with ease.



## State Machine Diagram


iOS Application
Current keep_alive info for all ESPs
Updated lat,lon and nickname for specified ESP in keep_alive
Internet
domecrawl.us website, iOS app information
probe request data, database requests & updates
Current Time
Batched probe request data & keep_alive GET requests
ESPs
DNS Probe Requests
Devices
Flask Servers
HTTP Requests & webpage
HTTP Responses
Website
Linode Server
queried information
SQL Requests & Injections
Databases

## ESPs

1. Automatically connect to WiFi.
2. Synchronize ESP8266 time with an NTP (Network Time Protocol) server.
3. Listen for all probe requests.
4. Send probe requests with time stamp in batches of 5 to the database.
5. Send "keep_alive" GET request every 30 seconds to signify the device is online.
6. Test connection and reconnect to WiFi if needed.
7. Repeat from step 2.

## Server

All of our server applications are written in Flask. Our backend is in MySQL.

1. Website
   - 1.1. Use Apache to listen for HTTP requests and serve the appropriate files.
   - 1.2. Use client-side JS to continually ping the server, checking for new data.
2. Database
   - 2.1. Obtain POST requests with new MAC addresses.
   - 2.2. Validate the data types.
     - 2.2.1. If valid, insert the data, timestamp, and signal strength in a table of mac addresses.
   - 2.3. Return valid or invalid HTTP status codes.
3. iOS Endpoint
   - 3.1. Listen for HTTP request and update "keep_alive" table according to the microcontrollers being deployed.
4. Notifications
   - 4.1. Twilio application in Flask that sends texts whenever there is a deployed device is unreachable due to some unforeseen circumstance.

## iOS Application

1. Calls a GET request to get the list of ESPs in the "keep_alive" table and display the list in the app.
2. Click on ESP of scan QR code of ESP box.
3. Waits for user to enter nickname and then the application automatically assigns GPS coordinates based off of location readings from the phone.
4. Locations are updated effectively and accurately with an HTTP request including coordinates, nickname, the MAC address, and time of movement

# Deployment

## Objectives

Deployment is one of the key concepts behind our project. The ability to deploy an additional ESP quickly, cheaply, and effectively should be an integral part of the customer experience. Therefore, as we considered how we should deploy devices, we had a few core concepts in mind.

1. Each deployment should require minimal configuration. The more the ESP could initialize *on it's own*, the better experience we could provide.
    a. This means that the ESP should connect to WiFi automatically, as the user shouldn't have to consider this.
2. Each deployment should be modular. Information systems—particularly information visualization systems—should be accommodative of them without additional effort on the part of the user.
3. Uptime is key—if one node goes down, then our data becomes exponentially less valuable. Therefore, we should be able to monitor uptime for devices and be alerted when one device goes online or offline. The device should also reconnect to WiFi in the case of a lost connection.
4. Quick and effective deployment is a necessity, so we decided to use create an iOS companion app to speed up the process. We can simply snap a picture of the QR code on each module and our database will be updated with the coordinates, time of last update, and the nickname of the location. This is then used to automatically create graphs and maps for our website.

## Steps for Deployment

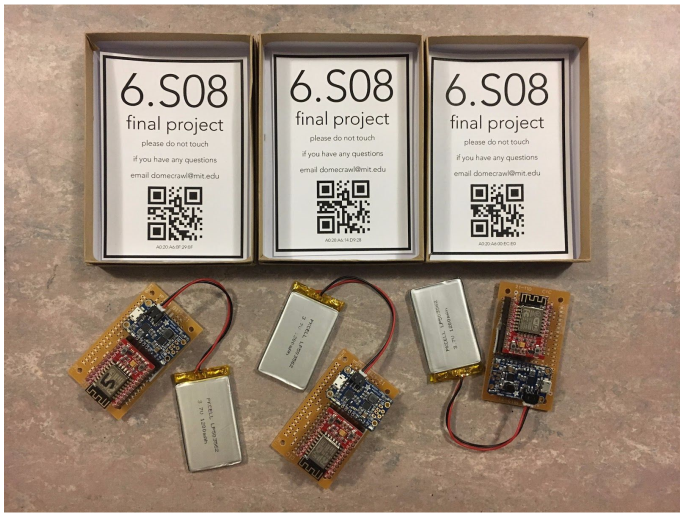
1. **Charge batteries or gather power supplies.** Our system currently needs either charged batteries or USB power supplies that plug into outlets.
2. **Install the iOS companion app.** The app is currently not on the app store, but we have plans to further develop the app for both iOS and Android and make it publically available with our product.
3. **Place the modules in desired locations and snap a picture of the QR code.** Set the GPS coordinates with either your current location or by moving the map with your fingers. Type in the nickname for each location, hit submit, and you are done.
4. Simply head to domecrawl.us to view your data and and our analysis of your location data.

# Portable Modules

## Hardware

The hardware in our system consists of an ESP8266 WiFi module microcontroller (here)[1] and source of power. We currently power our devices via a micro USB cable or a LiPo battery (here)[2]. The power sources plug conveniently into our power breakout board (here)[3], and the footprint of each unit is designed to be very compact, modular, and portable. With parts provided by the 6.S08 class, our team soldered 8 protoboards meeting these goals.

This image depicts 3 of our 8 modules. Our prototype boards have female headers so that we can easily disconnect and reconnect the ESPs and power boards. This proved very important because we can program our ESPs on a separate breadboard, move them to the protoboards, and then deploy the modules with ease. By doing it this way, we eliminated the need for extra hardware on each protoboard.



We chose to use the ESP8266 WiFi module instead of a Teensy microcontroller to save money, reduce size, and enhance performance. The ESP is cheaper, smaller, and more powerful than the Teensy. Furthermore, it can be used a soft access point, meaning it can act as both a WiFi station (router) and as a client for our database. With these capabilities, we have the ESP set to constantly receive probe requests and upload the data to the our database every 5 entries.

After completing the 8 hardware units, we created a clean-looking box to enclose each unit for deployment across MIT's campus. We labelled the boxes in a way that ensured the project would be safely distributed and not disrupted. Furthermore, we added a QR code to each box signifying the MAC address of each ESP. These QR codes are used with the iOS companion app to quickly deploy a network of modules with corresponding GPS coordinates.

[1] http://www.electrodragon.com/product/esp8266-smd-adapter-board/
[2] https://www.adafruit.com/product/258
[3] https://www.adafruit.com/product/2465

## ESP Software

The software on the ESPs works by collecting information about nearby devices (phones, laptops, etc.) and sending it to our server. The data that we are sending to the database consists of the epoch time (milliseconds since January 1, 1970), the unique MAC address of the ESP, the probe request from a device searching for WiFi, and the signal strength of that device. The data is conveniently packaged into a string on the ESP client side and sent via a POST request to the server. Furthermore, we adjusted the server side to handle batch posting, meaning we can send multiple database entries in the same string to reduce time spent sending the the server. We added this feature in hopes to reduce power consumption, but we learned that most of the power was consumed by acting as an access point regardless of the time spent on HTTP requests.

Some very important features of our ESP software include the following:

1. Automatic WiFi Connection
   a. When powered up, the ESP automatically searches for unencrypted WiFi networks nearby. With the list of networks ordered in strength, the function iterates through connecting to each network and pinging google.com. If there is a successful response from Google, the ESP will remain connected to the network. If there isn't a successful response (meaning no internet connectivity), the ESP will repeat by testing the next unencrypted network and so on.
2. SSID Includes MAC address
   a. We have programmed the ESP to set its SSID to our team name including the last two digits of the ESP's MAC address. This allows us to use our phones for quick debugging and check to see whether the network is broadcasting a WiFi signal as an access point. Furthermore, this permits us to ensure that our WiFi signals don't overlap, such that devices don't report to be in multiple places simultaneously.
3. Sending a "keep alive" GET request every 30 seconds
   a. The ESP sends a GET request to our server every 30 seconds signifying that the ESP is still online and functional. This is important because we use this information to keep up to date on which ESPs and locations are operational. When the server notices that one has gone offline, the dashboard on our website (domecrawl.us) will update and the team members should receive text notifications.
4. HTTP Post Requests with MAC address Information
   a. The ESP is constantly sending a post request with "BATCH_NUMBER" of MAC addresses per post. Furthermore, in the case where very low numbers of MAC addresses are being collected, the post request will default to posting at a minimum rate of POST_UPDATE_MINIMUM.
5. Automatic WiFi Reconnect
   a. IF the ESP goes without a successful response from the server for SUCCESSFUL_RESPONSE_INTERVAL milliseconds, the ESP will automatically reconnect to to WiFi.

6. ESP Blinking LED
    a. The ESP module will blink every time that it executes an HTTP request. This allows us to quickly know if the device is working while debugging.
7. Synchronize Time
    a. We use an NTP client library to sync the ESP clock with the real time. We then use the epoch time in seconds associated with each MAC address when uploading the database.

In the near future, we plan to implement a way to change all of the ESP parameters remotely. We will also implement a way to remotely flash (over WiFi) the ESP with firmware. This will help make the network of devices much more scalable because currently each device is programmed individually, which wouldn't be feasible for anything greater than 20 devices, for instance.

## Hardware Libraries and Dependencies

The following libraries have been very important to our software development on the hardware side:

1. ESP8266WiFi.h and ESP8266HTTPClient.h
    a. These two libraries aid in setting up the ESPs as soft access points, connecting to the internet and our server, finding nearby WiFi networks, collecting MAC addresses, and displaying each ESP's unique MAC address.
2. elapsedMillis.h
    a. This library is vital to keeping our ESP process's timely. We have many timers and intervals for each process so that we can conduct GET / POST HTTP requests at the correct times preset by constants. Eventually these constants will be adjustable over the year via our website.
3. NTPClient.h and WiFiUdp.h
    a. These libraries are used to sync the ESP clock with the time in UTC. We are then using this data to send the epoch time with each MAC address that is collected.

## Energy Management

In order to stay portable, our team worked hard to test possible ways to conserve energy in our system. One way that we tried to save energy was by using batch posting, meaning we send multiple MAC addresses clustered in an extended string in an HTTP request. We accomplished this task on both the client and server side so that we could send a group of collected MAC addresses up to some predefined limit.

With this software in place, we ran a test run with our 8 devices. The results from our test are summarized below. The MAC Address represents the ESP modules, the batch number is the number of collected MAC addresses before sending the database, and the time alive is the time in hours (rounded to the nearest hour) before the ESP died when starting with power from a fully charged 1200mAh 3.7 V battery.

| MAC Addresses | Batch Number | Time Alive (hours) |
|---|---|---|
| A0:20:A6:00:EC:E0 | 5 | 8 |
| A0:20:A6:0F:29:0F | 5 | 7 |
| A0:20:A6:04:62:4A | 5 | 8 |
| A0:20:A6:14:D9:28 | 5 | 8 |
| A0:20:A6:01:56:52 | 10 | 0 (b/c of broken ESP) |
| A0:20:A6:0F:26:15 | 20 | 7 |
| A0:20:A6:0F:2C:91 | 30 | 8 |
| A0:20:A6:01:56:F4 | 50 | 8 |

From this data, we concluded that the HTTP requests didn't take up much power compared to the other processes onboard the ESP microcontroller. For example, because 50 batch posts is 10 times fewer than 5 batch posts and yet the ESPs lived for roughly the same amount of time, it is safe to conclude that broadcasting the ESP as an access point is taking up the majority of the power. According to our calculations, the ESP appears to draw on average 150 mA (1200 mAH / 8 H). This is a constant power consumption of 0.555 W (150 mA * 3.7 V).

After conducting two tests that lasted around 8 hours each, the team decided that it would be most efficient to use a power outlet to collect data for a longer time interval. For our third and final test in this class, we have deployed 8 devices that are plugged into micro USB wall plugs. We plan to leave these running for a few weeks to collect a large dataset that we can use to perform data science and data visualization on.

Although wall outlets work without limits, the team is actively working on finding ways to improve power efficiency. We would like to have a system that has two options: use batteries and last a week in any location or use outlets to last indefinitely.

# Software Overview

## Server Software

- Database
    - Our database is using MySQL and is hosted locally on our team's Linode server instance. We currently have two main tables that we are using in our database: "keep_alive" and "mac_addresses." Our server is constantly listening for HTTP (both GET and POST) requests and updating these tables accordingly. The server is multithreaded to handle many requests at once.
    - We are using a MySQL database on our Linode box, with the following tables:
        - keep_alive
            - This table stores all of our devices, the time deployed, the last time communication occurred between our device and the server, the nickname of the deployment location, and geographic coordinates of the deployment location. Most of these are set through our iOS app!
        - mac_addresses
            - This table stores all probe requests from devices. These entries for probe requests include time captured, RSSI signal strength, and the associated ESP the device was captured from.
    - With 8 devices constantly running for 3 days, we receive on average 1,000,000 entries of MAC addresses from MIT's traffic on campus.
- Flask App
    - Endpoints
        - We have created many endpoints to hand all of our HTTP requests. We have endpoints for the database to keep it updated. We have many endpoints created for navigating our website, creating graphs behind the scenes, and keeping the website up to date. We also created an endpoint for the iOS companion app. This allows us to easily deploy our devices--giving them unique names and locations for each module.
    - Challenges
        - Multithreading
            - We found that multithreading our server was extremely important. When we have 8 devices running with over 5 requests a second from each, we must run processes in parallel. Originally we didn't have this enabled in Flask which caused our server to crash multiple times. However, our problem was fixed as soon as fixing this problem.
        - Closing MySQL connection
            - Our team ran into some problems on the server when opening too many MySQL connection in our Flask App. Our server would crash on occasion because we were not terminating our opened connection. This was fixed by closing a connection after each query that is made in Flask.

## Website Software

- Status Page
    - The status page is a convenient way to display the number of devices online and offline. It also keeps us updated with the number "unique MAC addresses" that have been detected since our last deployment.
- Graphs Page
    - On the http://domecrawl.us/graphs.html webpage we are utilizing Bokeh plots in Python to generate graphs that display the number of probe requests at a specific location over time. This page is created dynamically using Javascript. As soon as the HTML page loads, the Javascript sends a GET request to a Flask endpoint to get the list of devices online from our "keep_alive" table. This returned list is then parsed as a JSON list and used to create a box for each Bokeh plot to be displayed. After this is run on the initial page load, a user can select a time interval from the "Minute Interval" dropdown. This is then used to send a GET request to another Python endpoint, generate a Bokeh plot given the time interval (which counts the number of probe requests in each interval), and then return the HTML and Javascript text for the corresponding Bokeh plot. Finally, the Bokeh plot is inserted into the correct div tag and is displayed on the screen. As our dataset becomes larger, the graph takes more and more time to load. We will work on speeding up our graphing algorithms in the future to reduce time delays and improve large data management.
- MAC Address Search Page
    - This page is used to search for individual MAC addresses that have been saved in our database. We create and display an image for the corresponding MAC address and log the path of the device as it moved throughout the location of deployment--MIT in our case. The image is created by making use of the libraries NetworkX and Matplotlib.
- Map Page
    - Google provides a very nice map Javascript API. We are utilizing this API to add clustering visualizations and graphical animations of foot-traffic.

## iOS Software

- Intuitive Interface
    - The iOS app was created in order to help speed up the process of deploying the hardware to collect data.  In our "keep_alive" table, each ESP has a row.  The ESP has a location, nickname, as well as other relevant information about where it was placed. When probe requests hit that ESP, the keep_alive information for that ESP is copied into relevant fields in the "mac_addresses" table where all of the probe requests are stored with their locations and times.  The iOS app makes it easy to update the "keep_alive" table with the new GPS coordinates of where it was deployed as well as a nickname for the device.
    - The iOS app is very simple, but it gets the job done.  It starts with a list of all of the ESPs in the "keep_alive" table, displaying their MAC address and nickname.  In order to modify the data in the row for a ESP, you can either click on the row or use the QR

scanner button to scan the QR code conveniently placed on top of each of the ESPs. After scanning, a page for editing the information for the ESP appears. It has a map and a nickname field. In the map, the previous saved location of the ESP is marked with a pin, and the new saved location is wherever a second pin on the map is dragged to. The nickname field can be edited easily by clicking on it. After changes are made, a user can click the save button, and the new information for the ESP is sent up to the server to update the "keep_alive" table.
    - In summary all that is needed to deploy a module:
        - Scan the QR Code
        - Drag the map to the new location
        - Type in the new ESP nickname (user friendly name for location)
        - Click Save
- Server Connection
    - The server has a couple endpoints build in to make the iOS app function
    - Keep Alive Endpoint: The keep alive endpoint servers one function. To get the iOS app the list of ESPs inside the "keep_alive" table, with the relevant information to each ESP. To do this, it simply pulls the whole "keep_alive" table, packs it into a JSON with a custom JSON encoder, then sends it to the iOS application. The iOS application then displays the information in a table.
    - Set Location Endpoint: The set location is simple. It updates the location (latitude and longitude) for a given MAC address inside the "keep_alive" table. It takes in the MAC address with the latitude and longitude URL-Encoded and saves it inside the "keep_alive" table with the corresponding MAC address.
    - Set Nickname Endpoint: The set nickname is also simple. It updates the nickname for a given MAC address inside the "keep_alive" table. It takes in the MAC address with the nickname URL-Encoded and saves it inside the "keep_alive" table with the corresponding MAC address.

## Challenges and Obstacles

1. Programming the ESPs: One of our first challenges was to figure out how to program the ESPs. After doing a lot of research online, we found that people have created ways to reprogram the ESPs using Arduino and reflashing the ESP. We had a lot of difficulty comparing the ESP8266 to the tutorials online because the pins were named differently on our ESP. After reading through a lot of tutorials and trying different wirings, we were able to find a wiring that allowed us to flash the ESP. We had to do some experimentation with pin numbers to reverse engineer which pins controlled the built-in LED and which pins crashed the example code. Soon enough we were able to set the ESP up as an access point and receive all probe requests to that access point.
2. Connecting to the internet: One of the biggest challenges we had on the hardware side was disconnecting from the internet and not sending received probe requests. When a specific

ESP goes offline, it is often very difficult to figure out why and it would also not reconnect. In order to solve this problem, we wrote code for the modules that immediately detects when it is not connected to internet. After it detects a problem with internet, the ESP scans for all available wireless networks and sequentially tries to connect to an unsecured network. If the connection is successful, the internet is tested, and if it fails, it tries the next unsecured network. Using this strategy, even if a ESP loses connection to internet, it is able to reconnect quickly and keep sending data to the server.

3. Battery life: Another issue we had was with limited battery life. With the 1200 mAH lipo batteries, the ESP lasted about 8 hours. Each ESP needs a current of about 150 mA because the ESPs need to act as a wifi router as well as post probe requests when the batch number is reached (5 probe requests sent at a time). Because we wanted long continuous sets of data, we had only a couple options in order to make our ESPs last longer. We could play with the batch size (discussed in energy section), we could get bigger batteries, or we could simply plug them directly in walls. After trying the three options, we ended up deciding to plug the ESPs directly in walls to get continuous sets of data.
4. Multiple HTTP requests: A challenge that we had on the server side is that with numerous ESPs doing post requests to the server at the same time. After we increased the number of ESPs in our test set from 4 to 8, we noticed that the server gets backed up and sometimes crashes due to the number of post requests. In order to fix this issue, we made the Flask App that handles all post requests and inserts them into the SQL database multithreaded.
5. MAC Address Randomization: One issue we have that we spend significant efforts to try and solve is the randomization of MAC addresses. In iOS 7, Apple introduced software that randomizes the MAC address used to send probe requests every so often. Because of this, people with iPhones that walk by ESPs may not be picked up in multiple locations, as their MAC address could change in between those locations. In order to deal with this, we only consider the probe requests to be valid if the same MAC address appears in more than one location, signifying that the MAC address was probably not a randomized spoofed MAC address. This added another level to our SQL queries which significantly slowed some of our graphing.
6. Location Animations: With the current SQL database structure, it is very difficult to form a query to create animations. Our original goal for animations was to be able to animate a singular MAC address as it walks across campus between the different ESPs. In order to filter MAC address and their probe requests, we had to create incredibly complicated queries, sometimes filtering many tables. We ended up using queries that filter the database and requery with tables up to four times in order to get good usable data. This ended up taking a significant hit in our endpoint's response time. For some endpoints, it requires over 10 seconds to fully query the data.
7. Scalability: An issue we are currently having with our project is scalability for the future. As of now, we are storing about 1 million probe request rows stored in our database every 3 days. With such a large database, our complex queries are getting slower and slower. The queries needed to do animations on the database have started taking a long time, so we are looking for a more optimal solution.

## What’s Next?

**Goals**

Our team is hoping to bring Arealytics outside of the classroom and into the retail space. We have been developing our technology in class in such that it will be easily transferable to be built into a real product.  The market we are targeting is data analytics for things like malls and also events.

Currently there are no efficient methods of getting data on where people go when walking around through events or buildings. However, this data can be extremely valuable to event hosts and building owners. Event hosts could keep track of which areas are visited most frequently to improve the efficiency of future events. For example, conferences could spend more time preparing the events with greatest interest/attendance.

Furthermore, malls could place stores in more efficient locations depending on the trends of customer movements (Consumers after visiting store A, are likely to visit store B and C. A, B, and C should be located near each other). Malls can figure out which locations are more profitable, and charge more for lease for those locations. Malls can do a health analysis on their current numbers of customers and how that varies throughout the day.

Outside of retail space, there is also potential outside of the retail space and in the home space. Our team competed in a hackathon at the MIT Media Lab to prototype this technology catering to the elderly. We designed a system that could locate where someone is in their home. We would put our MAC address tracker devices in each room of the house and then we could monitor the path of an elderly person as they traversed through their home, for example. Using this data, is it possible to tell when someone is spending too long in a given location. For example, if an elderly person was spending an hour in the bathroom, our system could alert their children that something may be wrong. This is just another application of our technology in case we would expand outside of the retail space. A link to our project presentation is here.

In order to make our product market ready there is a lot that needs to be done.  We hope to create PCB designs for our ESPs and figure out how to produce them at a larger scale.  We want them to plug directly into an outlet as well as look nice and unsuspicious.  We envision that the ESPs could be as small as a 3 x 3 x 2 inch box that simply plugs into a wall outlet.   We also do not want to lose out on the event space, where outlets may be limited.  In order to deal with this potential issue, we would like to also build a version of the ESP that can be battery powered and distributed anywhere on the fly.

We also plan to expand tremendously with the data analytics.  Due to the scalability issues we encountered while doing the 6.S08 project (slowing over time), we hope to create an efficient way to store and access data.  We may switch our database over to a Mongo Database rather than SQL, and do strategic insertion with only probe requests that matter (potentially reducing redundancies 20x).  We also need to optimize our graphing algorithms and analytics algorithms so that we can quickly crunch new statistics for our customers.

Furthermore, we would like to add a lot to the analytics side of the probe requests.  In order to make Arealytics market viable, we need to bring data that is useful for our customers to their fingertips.  In order to do this, we may need to talk to our potential customers and figure out what types of graphs and things would be helpful to them.