Kerby: Your Curbside Parking Buddy

Neville Chima (Author), Mrinmayee Mandal (Author), Kanvi Shah (Author)

Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A system capable of reflecting street parking availability, giving drivers a time-sensitive insight on parking locations. It alleviates the frustration of having to circle around blocks multiple times only to never find a parking spot. Our solution aims to expand upon current garage and lot parking management systems and include street parking as another application area. It also provides a cheaper alternative to current state of the art systems since street parking is oftentimes close to free.**

***Index Terms*— ESP8266, IoT, MQTT, NodeMCU, Parking, Sensor, Smart Cities, WiFi**

I. Introduction

PARKING is one of the most painful parts of driving. In addition to the issues drivers face with inadequate parking spaces and parking maneuvers, they may simply not find parking spots, especially on the street. Finding an available spot is often a cycle of inconvenient turns and circling. Our proposal is to make a smart city solution that reduces the amount of time wasted looking for street side parking by providing users with the location of the closest open parking spot.

Commercial technologies include systems such as ParkPGH, SpotHero, and BestParking; however, these systems usually always work within garages and enclosed parking lots, due to the convenience of implementation and hence, economic viability. The target users for these apps are people who occasionally visit some city for an event. City locals use street parking more often because (i) it is way cheaper and (ii) it might be closer to their destination. Street side parking is here to stay for the foreseeable future and there exists a need to create a centralized parking system that allows users to access both garage and public street parking.

Kerby, our curbside parking management system, aims to extend the application of parking technology while remaining cost effective.

II. Use-Case Requirements

Our proposed system has two kinds of users: drivers and operators. In this section, we have included requirements necessary from each user's perspective. For both, the car must be less than 16 feet in length, in order to limit the scope of our system for a semester-long project. The average car length is about 14.7 feet long, so our system will be able to accommodate a large proportion of cars.

A. Driver Requirements

Kerby should be able to direct a driver to the closest open street parking spot from the driver's inputted destination, thus requiring user-friendly web interfaces. We also require that when Kerby shows a parking location, the driver can expect the spot will be open, have enough space to park, and be within 30 feet from where Kerby's route ends. This quantitative metric is used because 30 feet is approximately the length of two cars. Additionally, we require that Kerby provide accurate results about availability over time. The sensors will wake up every two minutes to sense if their state should change (spot gets taken or not). Street parking can be used both by Kerby drivers and local drivers. Many Kerby drivers will also be requesting spots ahead of time before they are anywhere near their destination. We also found that most people park their cars in two minutes. So taking all of the above into account, we set Kerby's real-time information requirement to one with two minute periods.

B. Operator Requirements

Kerby should help operators manage their street parking areas. We require that the spot modules have modular design for ease in scaling up or down. We also require that the spot modules are cheap enough (i.e. < $50) to encourage scalability and citywide adoption. These two requirements will sustain the benefit of street parking over building more garages. The spot modules should have a power source that lasts for a reasonably long amount of time without replacement, which we determined as at least 6 months. This is another factor in determining the two minute period where the sensor wakes up and sleeps. The spot modules should be weatherproof and compatible with roadside infrastructure, with easily replaceable batteries. Lastly, Kerby should convey overall parking usage information to operators through an interpretable graphical interface.

III. Architecture and/or Principle of Operation

Kerby's system design is aimed to be simple and digestible by any driver or operator. The principle of operation is easy communication between the state of the physical world (i.e. the parking spot's availability) and consumers (i.e. the drivers). The overall system has three key components: sensing IoT hardware installed on the curbside, an information warehouse stored on the cloud, and the user interface web application.

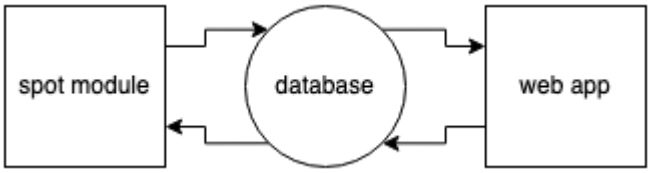


Fig. 1. Overall system block diagram

The essential functions are accurate sensing, timely communication, accessible information, and an intuitive user interface. Each of these functions are supported by subsystems of our design. Accurate sensing will be completed by an ultrasonic sensor and transmitted through an IoT device. In Fig. 2, a closer look at the "spot module" hardware is described. The details of each component are shared in section VI, but the overall concept of component interactions is displayed in these diagrams.

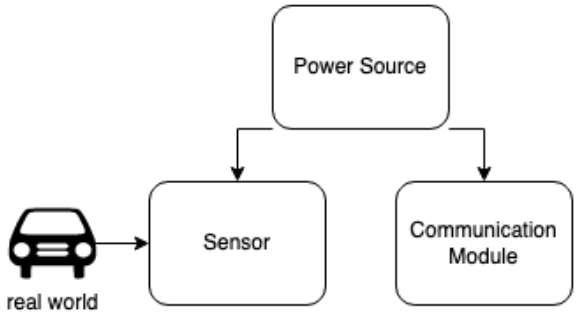


Fig. 2. Zoom-in of spot module system

Similarly, Fig. 3 shows a closer look at the database/cloud section. Our hardware spot module will take care of collecting the information of parking spot availability, transferring it over to this next section that processes and stores it in our cloud server (e.g. AWS). These server-side processes were where we improved our timing-based performance the most, so it was key our system implementation was as clean as possible.

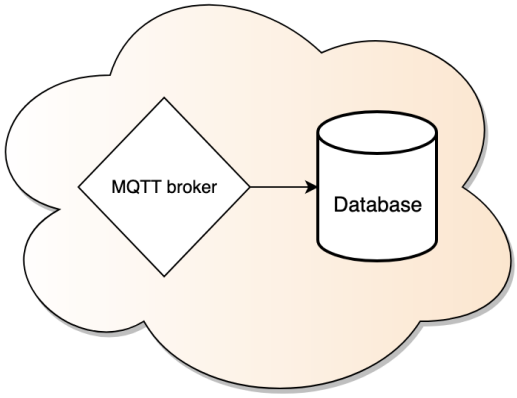


Fig. 3. Simplification of cloud-side communication and information storage

The block diagram of the entire system architecture is included in Fig. 4 in the Appendix. One major change is choosing to use Flask instead of Django.

IV. Design Requirements

Kerby's principles of operation paired with the user requirements has provided us with a roadmap for the design requirements your curbside parking buddy will have to fulfill.

The spot module subsystem must be able to detect cars parked in front of the sensor, calling for a distance sensor with a range of up to at least 1 foot. Next, the spots are to be detected for cars under 16 feet, which requires the modules to be configured for optimal results with this constraint. This requirement is fulfilled by positioning sensors every 8 feet on the curbside.

Additionally, we are looking for our system's power to last a reasonable amount of time before needing to change the batteries. The design will call for sensor wake ups every two minutes and a battery that can work with this timing to last around 6 months.

Requirements for communication come from the need to transfer data from a spot module to a central database and into the user's hands. Modules require a small, low-power consumption, low-cost communication device, with a reliable protocol like Wi-Fi. The Wi-Fi range also must be large enough to be able to hop on to the closest network and communicate with our central database. Since our MVP plan is to implement this on CMU campus, we have defined the range requirement only to be at least 100 meters. For future development, this would be extended to at least a few kilometers (may require the addition of an antenna).

To maintain the scalability and accessibility of Kerby, we defined the user requirements of modules to stay under $50 each. This has worked well with our principle of operation, to stay simple and functional, and driven our design requirements to be reasonable for cheaper IoT to fulfill.

Since we would like to provide instant data to our users, our database needs to be able to answer real-time requests from our users but also track all the sensor data that has been collected. Cloud services, like AWS, will be able to provide this specification for Kerby.

The user requirement of an easy-to-use graphical interface will be fulfilled by a web application that is compatible with the most common browsers and accessible to users. This means we will design the web app according to ADA standards. Our plan is to make the application very simple and clean, without too many options and extremely clear about where to input information and how to reserve a parking space.

Providing an accurate available parking location within 30 feet will require Kerby's modules to have their geolocations (longitude, latitude) be encoded with each of their unique IDs. This will allow us to use an API (such as Google Maps) to calculate driving distance between destination and parking spots, fulfilling another user requirement of being easy-to-use.

V. Design Trade Studies

We continue with a trade-off analysis on our chosen design specifications regarding how they satisfy our design requirements. Our system requirements were primarily

concerned with 1) detection and location accuracy, 2) transfer latency, and 3) project scalability.

A. Sensors vs. Cameras

Brainstorming in the initial phases of our project had led us down the path of possibly using cameras with computer vision to fulfill the requirement of "seeing" parked cars. Solving the same problem, these cameras would be installed at city intersections and look down the road and "see" if there were open spots or not.

However, after some driving around on the curvy Pittsburgh roads, a realization of how expensive and complex computer vision would become to get any accurate readings turned us away from this route.

Additionally, processing images using CV has an exponentially higher power consumption rate than a sensor has to take in a reading.

Instead, to fulfill the requirement of scalability and accuracy, we decided that a curbside module would be more feasible. This way location accuracy and detection accuracy could be fulfilled, at a lower cost than high functioning cameras paired with complex computer vision computations.

Table 1. Cost Comparisons of Different Sensing Options

Computing Device with Sensing Device	**Cost**
Jetson Nano with Camera	$120 + $26 = $146
Raspberry Pi Zero W Camera Pack	$45
NodeMCU ESP8266 with HC-SR04 Ultrasonic sensor	$6 + $2 = $8

B. IR Sensors vs. Ultrasonic Sensors

The design specification of needing a sensor that can sense if a car as far away as 1 foot from the curb could be fulfilled by different distance sensors. We researched both IR and ultrasonic sensors. The most common IR distance sensor used with Arduino projects has the range of 10 cm - 80 cm but is priced at $10 a piece. Another common IR distance sensor was priced at around $3.33 per piece but the range was only 10cm-30cm.

The most common ultrasonic sensor used in Arduino and IoT products, the HC-SR04, has a range of 2 cm - 400 cm and was priced at $2.40 per piece. This additional range and lower price point made it seem like the perfect choice. Additionally, we were able to find more documentation for projects with this sensor.

C. Microcontroller and Communication Board

Initially, the hardware portion of our project fits the description of a common household Arduino project. However, with the addition of needing to communicate between modules and with a central database, we introduced the possibility of needing WiFi.

Most Arduino boards are not WiFi capable and would require an external module for enabling this functionality. The Arduino UNO WiFi R2 does meet this specification; however, its price point at almost $45 essentially put it out of the running.

With our design specs of WiFi range, power supply needs, and price, we settled upon the ESP8266 NodeMCU. The low sleeping current consumption of 0.5 uA and active current consumption as low as 0.15 uA were huge factors, as well as the 3.3 V operating voltage. Additionally, there are more than enough digital I/O pins (13) to attach the ultrasonic sensor as well as any other debugging/future work extensions. There also exists plentiful documentation for creating IoT devices with the ESP8266.

VI. Power Source

Our original idea (and still hope for the future) was that we were able to run our system on solar power. However, the tradeoff in time spent making an inexpensive solar-powered system vs. time spent on making a more accurately functioning system made us lean towards focusing on the other sections of our project first. For power, we settled on rechargeable LiFePo4 batteries, as they have been researched to work well with the ESP8266 NodeMCU boards for long-lasting power. The batteries that we tested our system with had a range of 3.3V to 4.7V with 1200mAH current output. In future development, before approaching reusable power sources, we would want to update our battery to something with higher voltage output to make sure our HC-SR04 sensors work.

VII. System Implementation

This section describes our system's implementation in detail. End-to-End, Kerby features an on-site hardware subsystem (spot modules) connected to a remote software subsystem via an ioT communication pathway. We now describe each subsystem's internal components along with their design justifications.

A. Spot Modules

Kerby's spot modules are small packaged hardware circuits placed along the curbside of street parking spots. It features (i) an ultrasonic sensor to provide object detection readings (ii) an ESP8266 microcontroller chip equipped with Wi-Fi capability to process and transfer sensor readings and (iii) a battery source to reliably power the system-on-chip. A diagram of one spot module is depicted in figure 1.

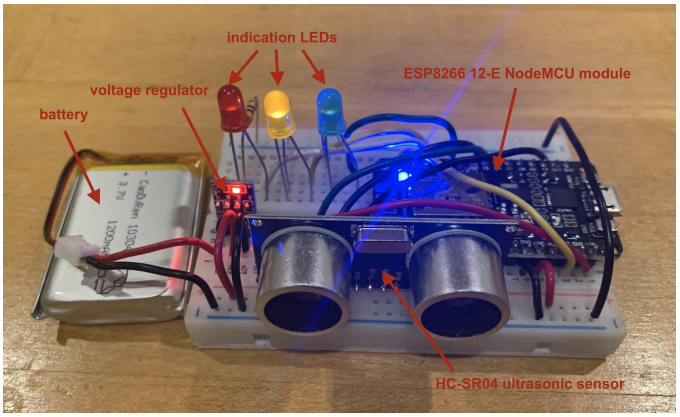


Fig. 4. Image of the hardware of a spot module. The HC-SR04 sensor (middle), LifePO4 battery (left) and ESP8266 NodeMCU (right) are electrically connected.

Our choice of ultrasonic sensor is the HC-SR04. This sensor's capabilities are described as follows. The HC-SR04 creates readings by measuring the time taken for a sound wave traveling through air to reflect off of a surface. Using internal ultrasonic transducers, the sensor converts electronic signals to sound pulses whose width is measured to determine the time the sensor traveled. The distance is easily calculated using the distance-speed-time equation. The HC-SR04 detects distances in the range of 2 - 250 cm with an effective angle of 15° according to its specification sheet (see Table 2) and our functionality tests. When no object is detected within its range, it emits a 250cm reading. Because Kerby's sensors are placed right on the curbside, centimeters from a car's expected parking position, we safely assume that such maximum reading is not a false negative. The HC-SR04 features *Trig* (output) and *Echo* (input) pins which can be connected to any of the microcontroller's GPIO pins. The *Trig* pin sends short LOW pulses to stabilize the sensor followed by longer HIGH pulses to trigger ultrasonic sound pulses. A 66% duty cycle over a period of 15 µs was sufficient to receive accurate readings. On the other hand, the *Echo* pin produces a pulse when a reflected signal whose length is proportional to the signal transmission time is received. Once connected to the ESP8266's GPIO pins, its values are read over a programming interface.

TABLE 2. HC-SR04 SPECIFICATION SHEET

Specification Name	**Specification Value**
Operating Voltage	DC 5V
Operating Current	15mA
Operating Frequency	40KHz
Range	2cm - 2.5m
Measuring Angle	15°
Trigger Input Signal	10µS TTL pulse

Open-source MicroPython firmware enables Kerby to run lightweight programs on the ESP8266 microcontroller. Apart from providing the flexibility to write programs at a higher level of abstraction (optimized python) than most IoT platforms, MicroPython comes with a rich set of in-built libraries for data processing. For example, the GPIO interfaces and data uploads are accessed via its *Machine.Pin* and *umqtt* classes respectively. Kerby uses *Esptool* to serially communicate with the chip's rom bootloader in order to install Micropython. Once installed, *Pymakr* - a compiler plugin - transmits the micropython code onto the controller's 2MB flash memory where it is converted to bytecode.

Algorithm Justification

In line with Kerby's design requirements, a main concern is the accurate representation of the real-world parking scene. Hence, Kerby requires an algorithm to convert analog distance readings to categorical availability readings i.e whether an object occupies the spot or not. We utilize the idea of multi-sampling over short time frames in making the conversions. Simply put, we take X sensor readings in Y second intervals to form one categorical value. Our working version uses 6 & 5 for X & Y respectively (see *testing latency* subsection for justification). If the multiple readings provide values below a threshold and within a range, the spot is considered occupied, otherwise it is available. In our working version, we adopted a threshold of 60cm because Pennsylvania law requires cars to park no more than 12 inches (~30.5cm) from the curb and a range of 5 cm from tape measure observations. Once the categorical value is determined for the current epoch, the sensor ID (SID) and value are published to an MQTT broker over the earlier established wi-fi connection. More details regarding the MQTT protocol can be found in the *IoT communication* subsection. And Figure 2. summarizes the spot module workflow.

Spot Module Workflow

1. Power on hardware & connect to Pymakr via serial port

2. Flash python & AWS credential files to controller root directory

3. Find MAC address and register ESP8266 on wi-fi network e.g CMU-DEVICE

4. Connect to wi-fi, periodically poll and pause CPU clock till status is connected. Backoff on connection failure

While True:

5a. HC-SR04.send_pulse_and_wait()
Get 6 readings every 5s

5b. If 5/6 readings < 60 cm && range(5/6) < 5 cm: occupied
Else: available

5c. umqtt.publish({sensor ID, occupied status})

5d. Connect to wi-fi, periodically poll and pause CPU clock till status is connected. Backoff on connection failure

5d. time.lightsleep(30 s) # Low power consumption

Fig. 1. Psuedocode describing the spot module software workflow.

Circuit Components Justification

While the main two components (ESP8266-12E and HC-SR04 ultrasonic sensor) ran perfectly well during our programming phase, we realized that once the board was unplugged from the USB port and only running on battery power, the readings being sent from the module were no longer accurate. After testing and researching other projects with these two components, we realized that the ESP8266 required a 3.3V source to avoid damage while the HC-SR04 was a 5V sensor. To remedy the first part, we inserted AMS1117-3.3 voltage regulators along with 100uF electrolytics capacitors to bring down the voltage being plugged directly into the board. The 5V ultrasonic sensors were not agreeing with this drop in voltage, so the latest design has the VCC pin of the sensor plugged directly into the battery Vout. To avoid damage of this higher voltage going into the ESP8266 from this route, a voltage divider circuit of two 10K Ohm resistors was placed on the Echo pin of the sensor.

Lastly, the indicator LEDs were added on as a way for us to know what phase of code was running (connecting to wifi, connecting to MQTT, taking a reading) as well as indicating if the sensor was detecting itself as "occupied or not."

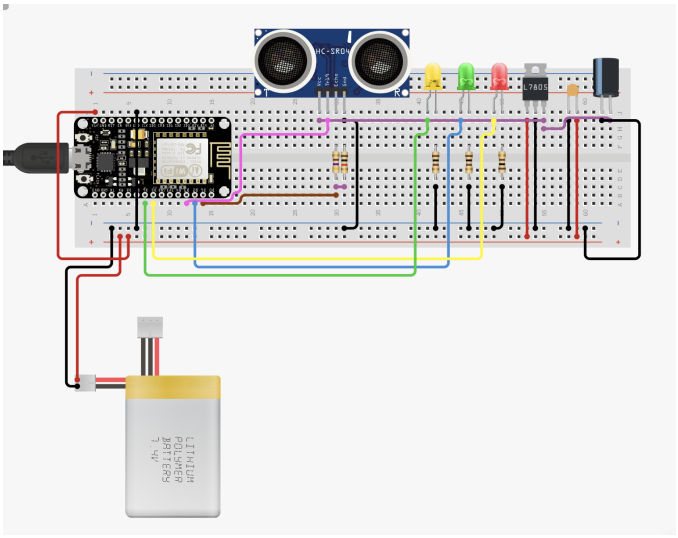


Fig. 2. Circuit visualization of all components.

B. *IoT communication medium*

This subsystem is concerned with the transfer of our sensor readings from the spot source to a cloud sink. Kerby achieves this using a popular lightweight IoT publish-subscribe network protocol, MQTT, provisioned through AWS's IoT core cloud service. At its core, IoT core provides an MQTT broker that connects to multiple client connections via TCP/IP over an exposed endpoint. These client connections take two forms: (i) a *Publisher* which uploads payload data to an intermediate cloud storage queue (ii) a *Subscriber* which connects to the MQTT broker and inserts the payload into the database on receipt. AWS IoT core provides extensive startup documentation and cost-tier levels to facilitate our implementation. At a high-level, we configure AWS IAM (Identity Access Management) profiles and create abstract IoT *things* with security policies attached to them for authN & authZ respectively. The security policies generate certificates containing ECDSA public-private encryption keys which allow a client connection access to publish/subscribe on a topic. *AWSIoTPythonSDK* is the API we use to call the client connection's access methods. Moreover, we monitor device connections and IoT telemetry via the AWS console.

Architecture Justification

Reliable data transmission and project scalability are use-case requirements outlined earlier that drive our IoT design decisions. We entrust AWS to guarantee the reliability of their MQTT protocol implementation as a leading cloud provider while we tune other parameters within our control. For example, the *AWSIoTPythonSDK* provides parameters for handling connection interrupts, resumptions and socket persistence. Nevertheless, MQTT offers varying levels of Quality of Service (QoS) that indicate whether a message was successfully delivered or perhaps not. Kerby uses the exactly once semantic (QoS level 1) which guarantees each published message will arrive exactly once.

A scalability issue Kerby faces is that connection queues may become congested as spot modules are exponentially added to the network. As a solution, our design doc proposed to assign connections to handle incoming sensor payloads belonging to the same geolocation group. i.e all spot modules in the same physical location publish to a topic designated for that physical location - say "readings/maggie_mo." This effectively allows us to move away from a many:1 to many:many pub-sub architecture and parallelize uploading/downloading our sensor readings from the cloud via multithreading. This architecture allows for quick synchronized database inserts and can linearly grow to be grouped via geolocation sub-hierarchies in a larger-scale future version. However, after implementing the many:many architecture, we realized the parallel processing benefit was minimal on our small-scale campus setup. Kerby witnessed <1s speedup periodically publishing data from 9 sensors evenly spread across three locations on CMU's campus - Margaret Morrison, Tech & Frew Streets - to a single sub client versus sending to three clients, one per location. Though we remain positive parallel processing remains important in a large-scale deployment of Kerby e.g a local municipality, its importance was nevertheless realized during redeployments of our software system. Basically, because the spot module and backend process code may be redeployed at different times during development and testing, published messages may be queued up over the network while the subscribe script is not running. Once the script is once again running, many messages are instantly dequeued and a single subscriber processes the batch slowly. This was evident in an experiment where we witnessed a ~4x speedup (214s to 623s) on inserting backlogged data into dynamoDB, between both architectures when deploying the subscribe script thirty minutes after the spot modules began publishing.

C. Server-Side Processes

This section discusses the server-side processes in Kerby's workflow. i.e the software modules invoked once data from the sensors is available on a subscribe client connection in order to serve a client's parking request. The major components involved are (i) a NoSQL database - DynamoDB, (ii) Google Geolocation APIs (iii) a Backend Engine and (iv) a Flask-based web-app.

DynamoDB

AWS's DynamoDB is Kerby's schemaless data warehouse. The schema for our database table is described in figure 3. The important details of our working implementation and potential improvements in a scaled-out version are outlined as follows: (i) We store all published sensor readings in a singular table. In the future, it may be more practical to store readings from co-located groups of sensors in the same table.This will help with faster table scans and data overflow in a larger sensor network (ii) We configure a data throughput on our tables of 5 *ReadCapacityUnits* & 5 *WriteCapacityUnits* which is basically DynamoDB terminology for a throughput of at most 5MB/s worth of data. Our payload size during development was small enough to work with these parameters but they may need to be adjusted in a data-intensive live system. (iii) Table records have a composite partitioning key consisting of the spot module's SID and insert datetime. The composite key is necessary in order to distinguish stale readings from a sensor when the BackEnd engine makes queries. (iv) Dynamo's schemaless nature provides the added advantage of flexibly extending our data structure. As seen in figure 3, *Occupied* is the only additional field stored in the database. Nevertheless, during range tuning and testing, we were able to easily include the taken sensor readings as part of the payload without any additional costs. This could be useful for communicating training data between remote ML models and the spot modules in a future iteration of Kerby (see *Future Work* section).

Table: Sensor-Data

```
{
 "SID": "String" # Primary Key
"Datetime": "String" # Partitioning Key
"Occupied": "String"
}
```

Fig. 3. DynamoDB schema for sensor readings

BackEnd Engine

Kerby's backend engine parses sensor data and runs algorithms in order to perform two main tasks. (i) update sensor history/state and (ii) indicate spot availability.

The former task is motivated by our use-case requirement to accurately reflect the field parking situation in real-time. Initially, Kerby queried DynamoDB for readings from the most recent publishing period - 60s. This data was then used to make decisions regarding the available sensors via quick web-app request-response calls. This implementation was unreliable (see *testing accuracy* subsection) and suffered from various issues. For one, since a user was provided with parking based on a single timeframe worth of data, oscillations in the data received due to unexpected foreign objects could soon after make the readings incorrect. Secondly, simultaneously providing parking to multiple users was a conflicting process because once one user A had reserved a spot, Kerby kept no history regarding existing clients and so user B was also provided parking with the latest database readings as the source of truth. This was quite problematic given that user A needed some time to arrive at their parking spot, before or after B arrived at theirs. Thirdly, the sensor sometimes did not publish readings in the last period due to clock drift on the microcontroller or may go completely offline if power is disconnected. In general, the scenarios described above create various distributed system fault tolerance and inconsistency issues Kerby needed to handle.

Our solution was to devise sensor state logic inspired by hardware finite state machines (FSMs). Kerby's sensor readings and client statuses formed various states, between which, each sensor transitions with time. New sensor readings from dynamoDB and time thresholds formed next state parameters to govern which new state a sensor transitions to. And based on a sensor's current state, rules determine whether to mark them available for parking, occupied by users, or uncertain for use and hence, hidden from users. A detailed diagram of Kerby's time-regulated FSM can be found in Appendix D. In general, its logic is as follows.

- A spot is characterized by two states. It's field state which can be one of (i) *Available* or (ii) *Occupied*. And its client state which can be one of (i) *Available* - implying not reserved for any client (ii) *Transient* - reserved for a client in transit to park (iii) *Occupied* - occupied by a parked client.
- A field and client state of *Occupied*/*Occupied* or *Available/Available* respectively is considered a consistent state. i.e the field readings match expectations from Kerby clients. All other state permutations are considered inconsistent i.e carry a certain degree of uncertainty
- A client can only reserve *Available/Available* spots which then moves them to the *Available/Transient* state. Once an active client has noted they no longer need parking, the spot immediately moves from the *Occupied/Occupied* state back to the availability pool.
- Other movement (or not) between states is possible via one of two types of conditions. (i) A new

database reading immediately triggers action e.g When a sensor is in an *Available/Available* state and an *Available* reading is received, it remains *in Available/Available.* (ii) A new database reading triggers action after continuously emitting the same categorical value over a period of time that falls below, above, or within a threshold range. For example, when a sensor's next reading is *Available* in the *Occupied/Available* state, it must have emit a streak of *Available* readings over a period of time greater than a set threshold for Kerby to decide, with high probability, that it should return to the *Available/Available* state.

- The FSM diagram in Appendix D notes two time thresholds. (i) *Conflict Thresh* is a parameter that determines how long a spot's readings must be consistent before Kerby decides it is safe to leave an inconsistent state (ii) *Arrival Time* refers to how long the user is expected to arrive at the parking location based upon Google Distance Matrix API estimates (discussed below).

The other major backend task involves determining spot availability. A read-write lock is used to control critical sections in accessing spot module states while the backend engine updates states in the background. The algorithm relies on a predefined orientation of the sensor network stored in-memory. The sensor topology schema is shown in figure 4. The topology assumes that spot modules are placed in a row on the curbside 8 feet apart from one another (see *testing cost* section for cost-distance tradeoff). Three adjacent *Available/Available* sensors are needed to dynamically mark an open parking area which a car of average length should fit in. Out of those three, either the left or right spot module could be used together with two adjacent free spots to form another spot, provided they are not at the edge of the parking perimeter. This is because each spot has eight feet in front and behind it. Hence, to allow cars to park bumper to bumper (as done on the street), each set of three occupied spot modules can provide overlapping use to multiple clients at their boundaries. A visualization of space allocation nuances can be found in the Testing Subsection B.

```
Configuration Schema
{
  "GroupID": "String -> PK",
    [
      {
          "SID": "String",
            "Left SID": "String",
            "Right SID": "String",
            "Geolocation":
              {
                  "Longtitude":"String",
                  "Latitude": "String"
              }
      }
    ]
}
```

Fig. 4. Schema indicating sensor row topology.

GoogleMaps APIs

The other major backend component involves using Google Map geolocation APIs. Specifically, we access gcloud accounts and obtain APIs keys which allow Kerby to use the *Geolocation & Distance Matrix* APIs. The *Geolocation* API is used to convert the user's destination from a well-formatted text address (taken from a web app form) to longitude and latitude coordinates. These coordinates, along with spot module coordinates are passed into the *Distance Matrix* API to show the shortest distance & shortest travel time sensor locations.

Web-App

The web-app renders html page templates and displays results from the Backend engine via flask. It features two main views. (i) a request view that prompts the user for a destination address and shows a google map route from their current location to the parking spot (ii) a viewspots view to show the current status of spot modules. Images of the web app are shown below.

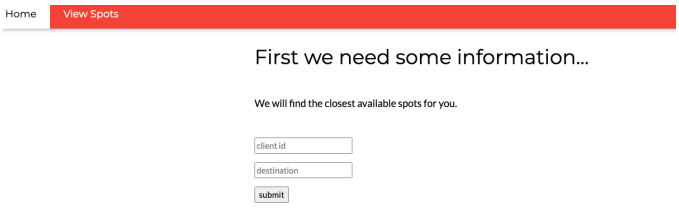


Fig. 5. Request parking webpage

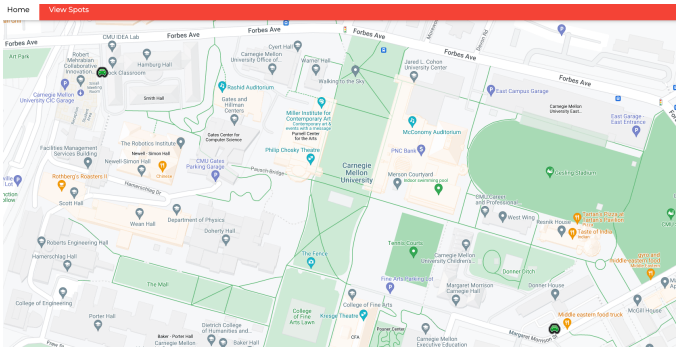


Fig. 6. Viewspot webpage. (Green car icons on map indicate available spots)

VIII. Test, Verification and Validation

A. Test Results for Ultrasonic Sensor Performance

The sensors came with a spec of a 15 degree measuring angle. However, there was not much research out there about how this affects readings in terms of objects in vertical and horizontal view range of the sensors.

In order to make sure our sensors would be able to meet the requirements of detecting the average vehicle, we performed curbside testing for many cars along Margaret Morrison St. We were able to verify that the sensors detected any car that came within the 400 cm and 15 degree range, adding a stipulation to the users that could use our product as is - their cars needed to be at a max of 18 inches above the ground.

B. Test Results for Placement of Spot Modules

In order to fulfill our easy-to-install requirement, we built Kerby to be situated on the curbside. After testing and researching the range of the ultrasonic sensor, we found that one sensor would not be able to detect an open spot for an entire car from head to trunk. Thus, we decided to "sample" the curbside by placing a module every few feet to detect at least 16 feet of open space for a user. There were three measurements we looked out for: distance between sensors, cost associated with one open spot, amount of unutilized space on street parking area. In the case of 6 feet spacing, we found that to guarantee a spot of at least 16 feet, we would need 4 sensors, thus translating to an open spot of 18 feet. If the average car came to park in this spot, we would end up wasting at least 4 feet.

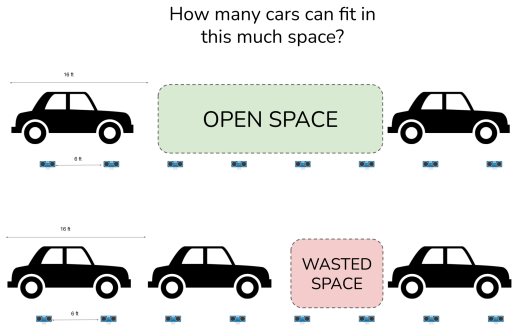


Fig. 7. Vehicle allocation visualization

Extending this analytical process to different distance values, we found that 8 feet would be the most optimal choice for our project, as it wastes the least amount of space and is a good middle ground in terms of cost. This way, when three sensors indicate that they are not blocked, we can guarantee that our average user will have a spot to park in. Overall, this testing helped us verify our design requirement of accurate request location, meaning when the user gets to their requested spot they will have the appropriate amount of space to park.

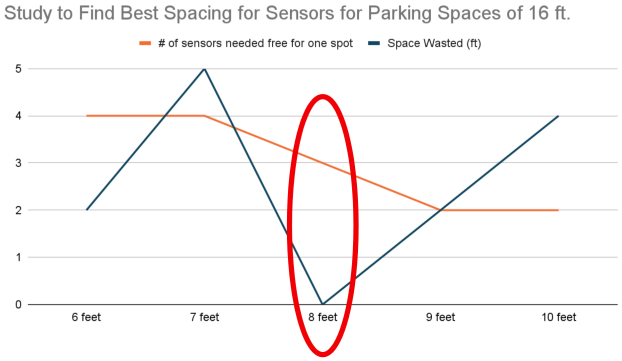


Fig. 8. Graph showing testing results of tradeoff between frequency of sensor placement, costs and unutilized space on street parking area

C. Usability Test Results for Web Application

After building the first functional prototype of the web-app, we asked 5 volunteers to use and navigate it without any prior knowledge. We recorded some common questions asked, such as "do I have to type a full address when requesting?". Then, we explained the purpose of Kerby and recorded any additional feedback they had for us. Four out of five volunteers did not like typing in the full address when requesting a spot. We changed our design to incorporate a drop-down menu that suggests or autocompletes as the user types. Two out of five participants thought that the route should be more interactive and informative. In response, we added a blue dot that follows the user's location as they move. We also added an information window that pops up when a marker is clicked.

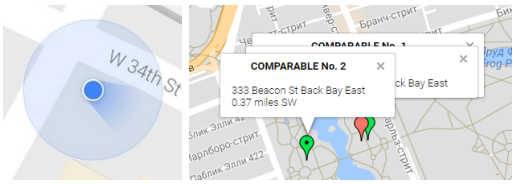


Fig. 9. Examples of suggestions made by volunteers during testing

After these changes were made, we did another round of the same testing and found that there is a 80% likelihood that people would use Kerby again in the future. Thus, this testing process achieves the user requirements of an easy-to-use web application.

D. Test Results for Power Consumption of Spot Modules

From running our batteries for long periods of time, we were able to test their lifetime. We found that the batteries drain to lower voltage values when run for long periods of time without any deep sleep periods. Results showed that after 2 hours of continuous running, the batteries were only giving

around 3.3V. This downward trend of voltage draining continued as the batteries ran longer. This showed us that we would need to figure out a different system for battery charging or replacement - or especially look into how other low-power IoT systems in the current civil infrastructure work.

E. Test Results for Latency

We measured the time taken for various processes in Kerby's communication pipeline to complete (see Table 3) over ten minutes of simulating E2E requests. Generally, each task was completed within a few seconds and the only noticeable latency bottleneck was the RW lock used in the backend engine. This was expected as the rate of user requests may variably align with the frequency at which the spot states are updated. Hence, the request may have to wait a few seconds before the update is finished before proceeding. Nevertheless, the total communication latency time for all tasks sums up to under less than a minute. This plus the two minute conflict threshold used in the time-regulated FSM totals approximately a 3 minute wait time for Kerby requests. This is quite less than our 5 minute parking use-case requirement.

TABLE 3. LATENCY RESULTS OF KERBY SOFTWARE TASKS

Process	**Average Time Taken**
mqtt publish	0.87s
mqtt subscribe	2.1s
DB inserts	< 1s
DB queries	< 1s
Availability check	2.48 s
RW Lock delay	6.24 s

IX. PROJECT MANAGEMENT

A. Schedule

Refer to the attached schedule at the end of this report in Appendix B. We have modified the schedule along the way several times, but have reached its end with a functioning demo by the due date.

Major changes from the design document to the final project report document were primarily due to making additional time for integration of subsystems.

B. Team Member Responsibilities

Originally, we distributed responsibilities among team members according to each person's strengths and interests. Kanvi would handle the hardware setup for the sensor modules and the frontend graphics for the web app design. Mrinmayee would be in charge of researching and testing sensor detection algorithms to parse data from the spot module. Neville would implement the communication between module and central database and the central hub maintenance software. However, getting the hardware spot modules functioning took longer than planned, and duties had to be rearranged. Neville finished setting up the IoT communication medium rather early. Thus, Mrinmayee passed the development of sensor detection algorithms and the overall software backend engine using database info to Neville in exchange for Kanvi's secondary role in designing the frontend of the web app.

As for presentations and paperwork, we all worked on different sections to draft up content and edit together. Lastly, we all worked together for integration and field testing, and we also helped each other out as needed throughout the semester.

C. Bill of Materials and Budget

Refer to Table I, Bill of Materials in Appendix C. As mentioned earlier, we prioritize being cost effective in our choice of materials.

D. AWS Usage [if credits requested/used]

Our database workload was small enough to stay within the cost-free AWS tier. No credits were requested.

E. Risk Management

One of the possible risks for Kerby included failure of communication from module to database. In the case one module starts malfunctioning, either because of low power, WiFi interference, or another reason, we made sure to mark its place in our database with "unknown status." This is also what was displayed to our users on the graphical interface, as to make sure no one is led to a "possibly open" spot.

Another possible risk for Kerby is being able to detect correctly that the requesting user parked into their requested spot and not someone else. We try to manage this by keeping track of the duration provided by the Google Maps API when the user requests a route. For the further improvement, we would have incorporated user feedback (simple as buttons with "yes, I found and used the spot!" or "no, the spot was not empty"). With our model of two types of users, this would be information given to the operators so they could check the modules as needed.

In order to mitigate risk of hardware damage in the field, we

planned to encase our modules with weatherproof materials. These materials were envisioned to consist of an airtight plastic box with holes cut for the sensor. However, due to the semester time constraints, we were not able to mitigate this risk entirely.

X. Ethical Issues

Streetside parking is technically available for all and not restricted to only users of our system, Kerby. The ideal user would be a driver who has access to a device that can open our web application. The product would be most appealing to drivers who are intimidated by street parking in the city, which would mean they are likely not city locals. Thus, the biggest ethical issue stemming from our system is regarding these regular non-users of Kerby who had their "usual spot", everyday for work for example, and now will have to contest with Kerby users. Additionally, since the Viewspots feature on our web-app distinguishes Kerby vs non-Kerby users, operators may "punish" non-Kerby users. Lastly, if implemented on a larger scale across cities, Kerby exposes information about traffic and congestion of the cities to the Internet, which may be concerning for security.

XI. Related Work

Currently there are no commercially available widely used products that work similar to what Kerby proposes to execute, as per its final design. However, there is work being done with similar goals. INRIX is tapping into the IoT of the automotive industry and introducing ultrasonic technology to be installed on the cars itself. If all cars use this technology, then cars parked alongside a curb would be able to provide information about the availability of spots in front and behind them.

Spot is another company that is developing smart city solutions, with the goal of making all parking information available at the tip of your fingertips. Work they have completed has made for an almost complete database of parking information for a portion of Sydney, Australia - displayed through a graphical web user interface that can tell you as you mouse over areas: if there is free or paid parking, if there is space available, if there is a temporary construction site blocking the spots.

XII. Summary

Overall, Kerby is a system that aims to reflect street parking availability, acting as a street parking management system to give drivers insight into where to go ahead of time. The system has three main component groups: spot modules, central data hub, and the web application. The stakeholders are users of street parking in big cities. In the end, the system did meet most of the design specifications.

A. Future work

Kerby has a long way to go from where we ended up this semester. One day, we intend to explore the possibilities of reusable power and maybe Kerby will be able to be solar-powered! When made for installation, it will be encased in a weatherproof and car-weight bearing material. The web application will be more robust and users will be able to learn more about the locations they are choosing to park in, like the price per hour. Kerby will require a scaled-up version of stress testing, where multi-client scenarios will be critical. For anyone who approaches the Kerby system in the future, we suggest starting with power testing almost immediately as well exploring different communication protocols like MQTT. We are excited for Kerby in the days that come.

Glossary of Acronyms

MQTT – Message Queuing Telemetry Transport

References

[1] *Arduino vs ESP8266 vs ESP32 Comparison*, Accessed on Feb 15, 2022, [Online]. Available: https://diyi0t.com/technical-datasheet-microcontroller-comparison/

[2] *Best Battery for ESP8266 microcontroller*, Accessed on Feb 20, 2022, [Online]. Available: https://diyi0t.com/best-battery-for-esp8266/

[3] *ESP8266 NodeMCU with HC-SR04 Ultrasonic Sensor with Arduino IDE*, Accessed on Feb 28, 2022, [Online]. Available: https://randomnerdtutorials.com/esp8266-nodemcu-hc-sr04-ultrasonic-arduino/

[4] *Ultrasonic Sensor Parking Availability Technology*, Accessed on March 1, 2022, [Online]. Available: https://inrix.com/blog/ultrasonic-sensor-parking-availability-technology/

[5] *Spot: Smart city parking and mobility solutions.* Accessed on Feb 2, 2022, [Online]. Available: https://www.spotparking.us/spot-cities

[6] balimaco00balimaco00, et al. "Call Function Initmap with Parameters in Gmaps API." *Stack Overflow*, 1 July 1965, https://stackoverflow.com/questions/47104164/call-function-initmap-with-parameters-in-gmaps-api.

[7] *Google Earth/Maps Icons*, http://kml4earth.appspot.com/icons.html.

[8] *Google*, Google, https://developers.google.com/maps/documentation/javascript/geocoding.

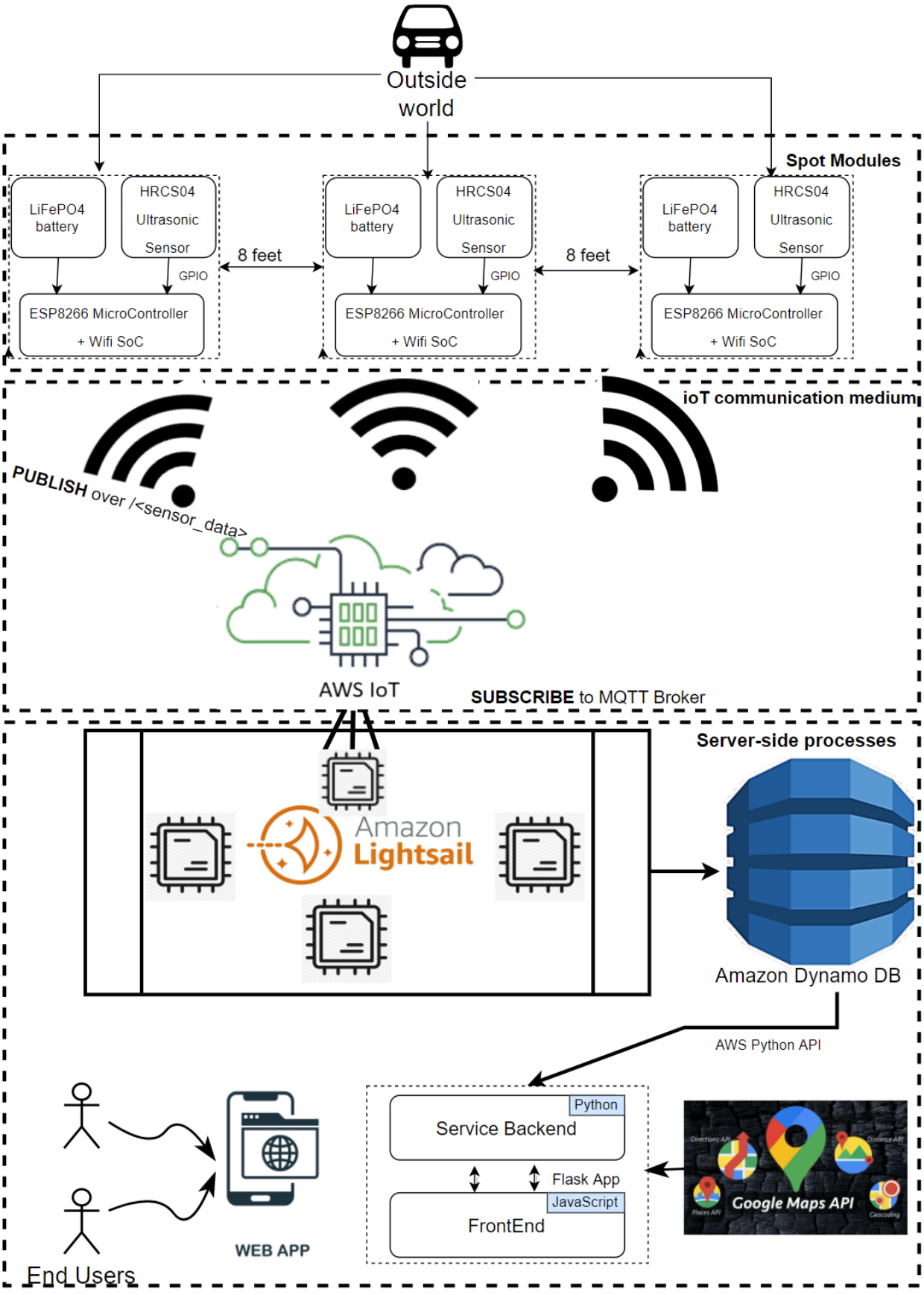
[9] *Google*, Google, https://developers.google.com/maps/documentation/javascript/markers#maps_marker_symbol_custom-javascript.

[10] "How to Add HTML5 Geolocation to Your Web App?" *Gearheart Web Development Team - Django, Angular.js, Outsourcing*, https://gearheart.io/articles/ow-to-add-html5-geolocation-to-your-web-app/.

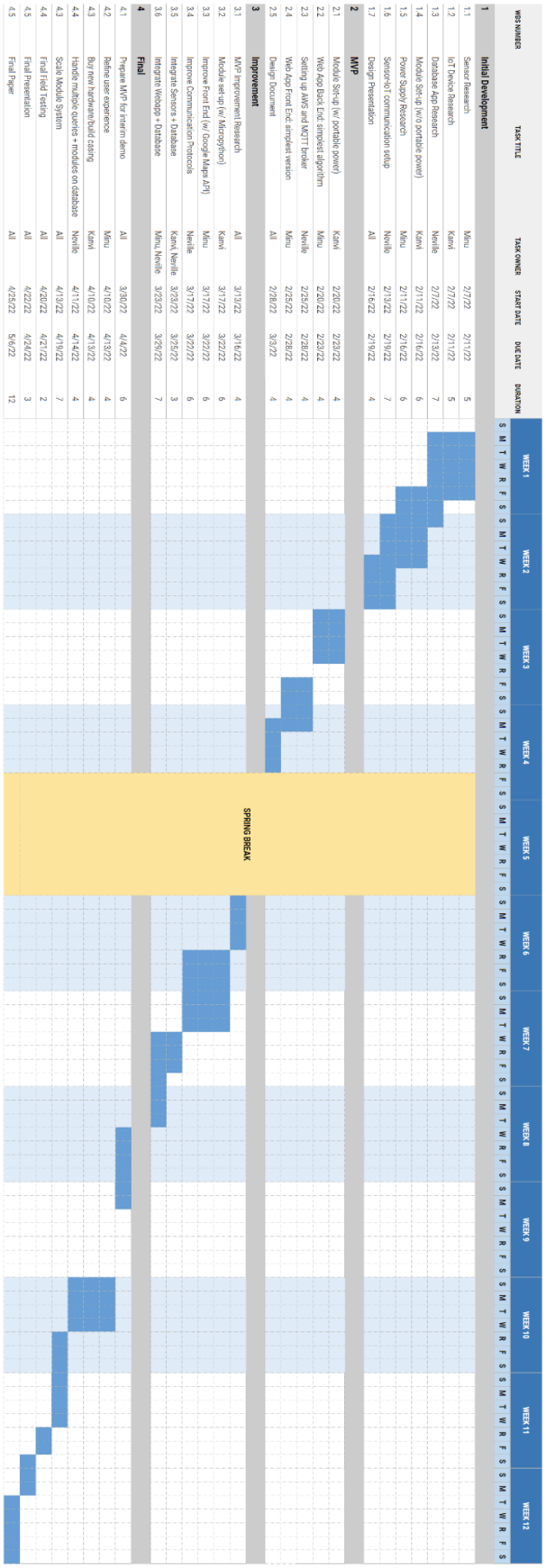
[11] James, et al. “Google Maps API - Center Map on Client's Current Location.” *Stack Overflow*, 1 Mar. 1961, https://stackoverflow.com/questions/17382128/google-maps-api-center-map-on-clients-current-location.

[12] “Python Tutorial - Google Geocoding API // Blog // Coding for Entrepreneurs.” *Coding for Entrepreneurs*, https://www.codingforentrepreneurs.com/blog/python-tutorial-google-geocoding-api.

APPENDIX A: System Description Diagrams



Appendix B: Project Management



Appendix C: BOM

Table I. Bill of Materials (to make six spot modules)

Description	Model #	Manufacturer	Quantity	Cost per unit	Total	Used or New to Design?
ESP8266 NodeMCU Dev Board	CP2102 ESP-12E	HiLetgo	6	$5.46	$32.78	Used
Ultrasonic Sensors	HC-SR04	Tangyy	6	$1.84	$11.04	Used
Batteries	14430 Rechargeable LiFePo4	JESSPOW	8	$2.13	$17	Not used
Batteries	103040 1200mAH	CaoDuRen Store	12	$9.99	$119.88	Used
Voltage Regulators	AMS1117-3.3	Frienda	6	$0.45	$2.70	New
Green, Yellow, and Red LEDs	-	-	18	-	-	New
10K Resistors	-	-	12	-	-	New
100 uF Capacitor	-	-	6	-	-	New
Mini Breadboard	400 Points	Ambreddr	12	$2.02	$24.38	New

Appendix D: FSM State Transition Diagram