Senior Capstone
Specialized Parking Oriented Technology

Ramzey Ghanaim, Jonathan Lam, Mario Cabrera, Travis Rogers

Winter and Spring 2018
Parking Management Project

Contents

1 Introduction

The purpose of this project is to design a power efficient parking management system that will make parking easier to manage for parking services as well as give users an easier time with finding and paying for parking. Users will be able to track the number of available spaces in parking lots and automatically verify their parking through their smart phones. Parking services can track the parking spot status of each space and manage user account information through a secure website. The project is named Specialized Parking Oriented Technology (SPOT).

SPOT is divided into 6 key pieces as follows:

1.1 Microcontroller and Sensor - Travis

The microcontroller and sensor is a majority of the physical hardware for this project, consisting of the sensor unit and its enclosure. The interaction between the user and the sensor unit is limited to the user blocking the sensor with their vehicle. After blocking the sensor for a short time (5s), the unit will assume that there is a car attempting to park and then send a message notifying the gateway that the spot is now busy. Upon receiving a message from the Gateway, the LEDs on the unit will all show a color corresponding parking status: open, closed, busy or illegal. When the car leaves the spot, the sensor will be cleared and send a message notifying the Gateway to the spot's new open status. The sensor unit and its components will be in an acrylic box with LEDs to communicate with the user.

1.2 Gateway - Ramzey

The Gateway will act as a server and manage communication between the units and the App Engine. It is responsible for aggregating the data from multiple microcontrollers. This helps with scalability by reducing the message sent to the App Engine. More details can be found in the "Specifications" section.

1.3 Mobile Application - Mario

The Mobile App will automatically detect the bluetooth beacon from the sensor unit and begin the verification step. From there, the user's permit on their account will be matched with the permit type of the spot. If the permits do not match, the Mobile App will either prompt the user to pay for the parking spot or indicate that the user cannot park in that spot. While the user is not actively parking, the app will display information such as the percentage of empty spaces available in each lot, their current parking location, and their account information.

1.4 Administrator Website - Ramzey, Jon

The Administrator Website is intended for administrator use only. It will have various functions, such as modifying user accounts and viewing parking statuses. The administrator website will include parking lot maps with a real-time color-coded status for each parking space, customizable permit and payment options for users, and a data log of transaction and occupation histories.

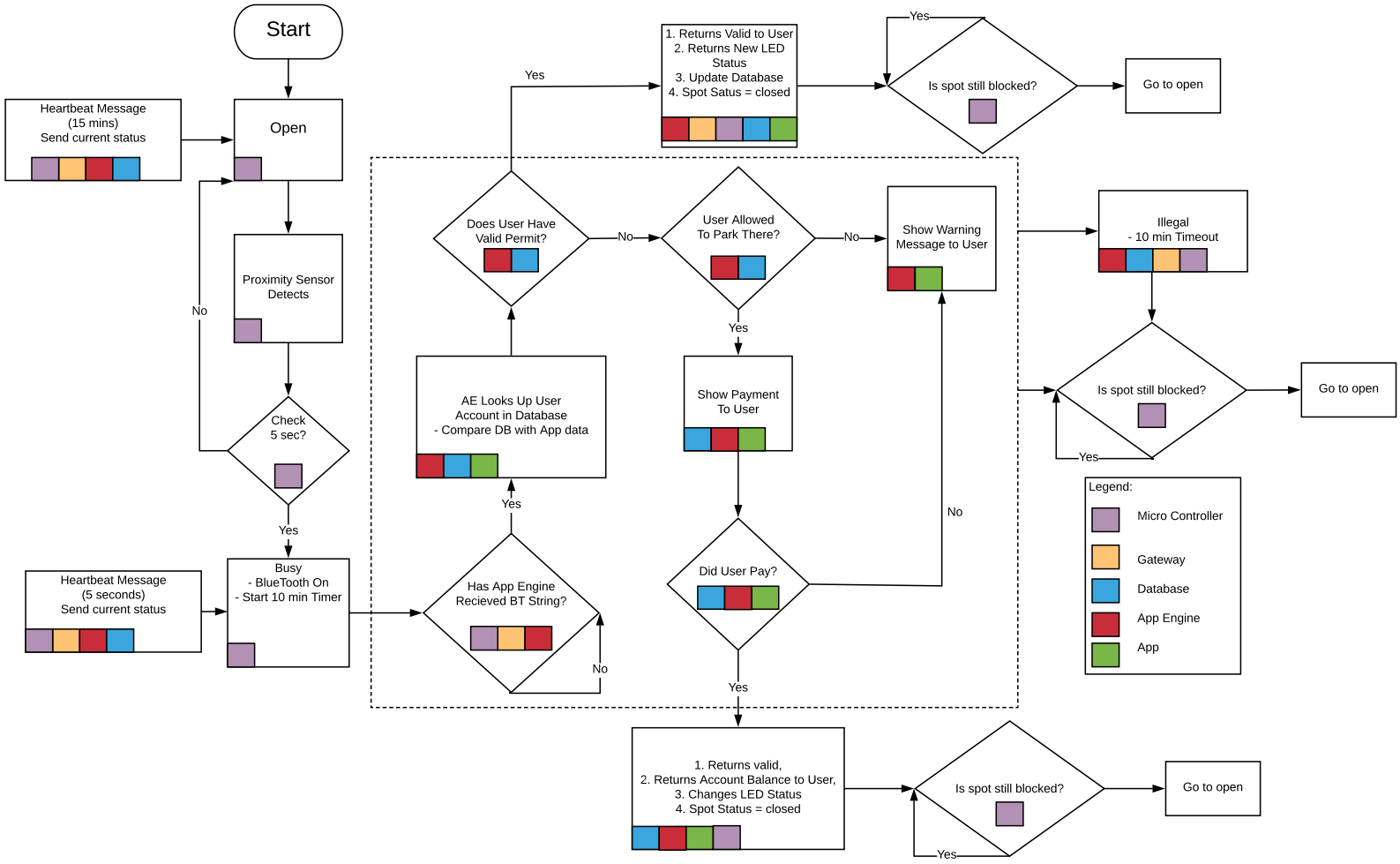
1.5 App Engine - Jon

The Google Cloud App Engine is in charge of connectivity between the Gateway, Mobile App, Website, and Database, as well as hosting the Admin Website. It is in charge of delivering and updating the new statuses from the sensors to the Database. The App Engine will handle the verification of the parking user by receiving user information from the Mobile App and checking the permit associated with the user's account from the database. It is also responsible for handling payments by deducting the user account's balance, logging transactions, recording occupation histories, and responding to requests from the Admin Website.

1.6 Database - Jon, Ramzey

The Database stores the information for the SPOT system. This includes tracking the attributes of the spot, including its current status, daily rate, Bluetooth ID, and allowed permit type. It holds all the user accounts and its corresponding balance, permit types, license plate, etc. It also contains all transaction logs and occupation histories for each the spot. Further details of the database can be found in the "Specifications" section.

An overview of our project can be seen in the next page on the following flow chart:


Start
Heartbeat Message
(15 mins)
Send current status
Open
Proximity Sensor
Detects
No
Check
5 sec?
Yes
Heartbeat Message
(5 seconds)
Send current status
Busy
- BlueTooth On
- Start 10 min Timer
Has App Engine
Recieved BT String?
No
Yes
AE Looks Up User
Account in Database
- Compare DB with App data
Does User Have
Valid Permit?
No
User Allowed
To Park There?
No
Yes
Show Payment
To User
Did User Pay?
No
Yes
Show Warning
Message to User
Yes
1. Returns Valid to User
2. Returns New LED
Status
3. Update Database
4. Spot Satus = closed
Yes
Is spot still blocked?
Yes
Go to open
Illegal
- 10 min Timeout
Is spot still blocked?
Yes
Go to open
1. Returns valid,
2. Returns Account Balance to User,
3. Changes LED Status
4. Spot Status = closed
Is spot still blocked?
Yes
Go to open
Legend:
Micro Controller
Gateway
Database
App Engine
App

2 Specifications

2.1 Sensor - Travis

The sensor we are using is the HC-SR04 Ultrasonic Module. The module uses four 5V pins. The pins are as follows:

1. **VCC** - Power is supplied through the VCC pin.
2. **TRIG** - The sensor sends out an ultrasonic wave when we send a $10\mu s$ length pulse to the trigger (TRIG) pin.
3. **ECHO** - When a wave is reflected back, a pulse is sent to the ECHO pin.
4. **GND** - The sensor is connected to ground through the GND pin.

The microcontroller will constantly and consistently send a $10\mu s$ pulse to the TRIG pin once every second. The sensor then sends out an ultrasonic pulse through the transceiver. A pulse corresponding to the distance of an object from the sensor is then sent to the receiving end of the sensor. The microcontroller will take in this signal from the ECHO pin and can then perform the necessary calculations to determine if the spot would then be considered busy.

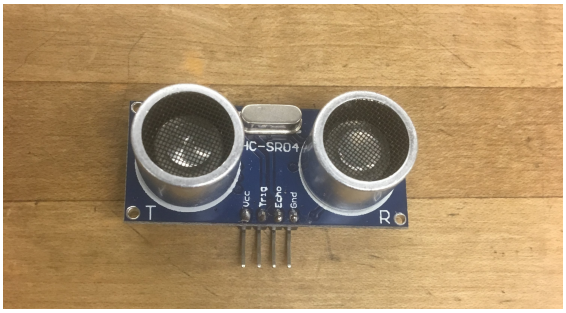


Figure 1: The HC-SR04 Sensor Unit

2.2 Microcontroller - Travis

The sensor unit will run off of the CC3220S TI Simplelink microcontroller. This chip has both 3.3V and 5V I/O power pins and 3.3V and 5V pins that can be used to supply power to the rest of the system. This chip also has Wi-Fi capabilities to connect with the gateway.

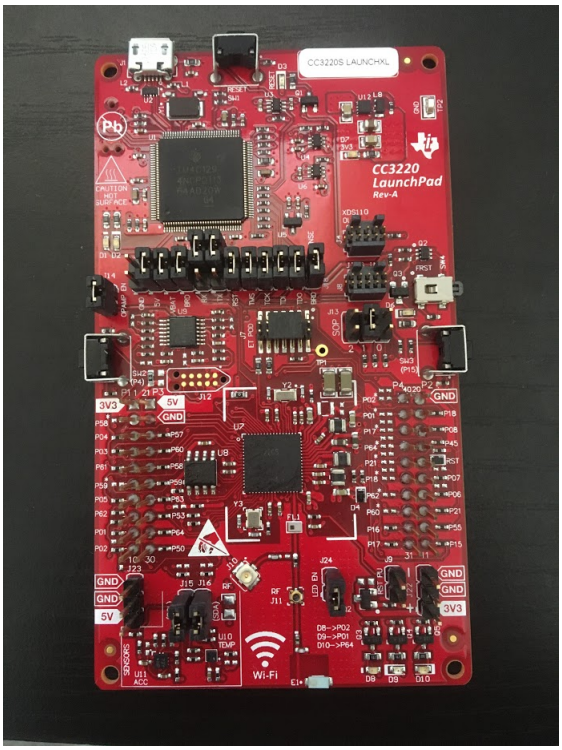


Figure 2: The TI Simplelink CC3220S Microcontroller

Each sensor unit includes the microcontroller, blue tooth beacon, ultrasonic sensor, LEDs and the body that holds the system together. Power is connected to the micro through a USB wall plug in. The micro connects to the beacon, sensor and LEDs through wired connections from the I/O and power pins. Figure 6 shows a wiring diagram of all of the connections within the unit. All of these components are soldered on to a perf board while the microcontroller is separate. The microcontroller is connected to the perf board through jumpers.

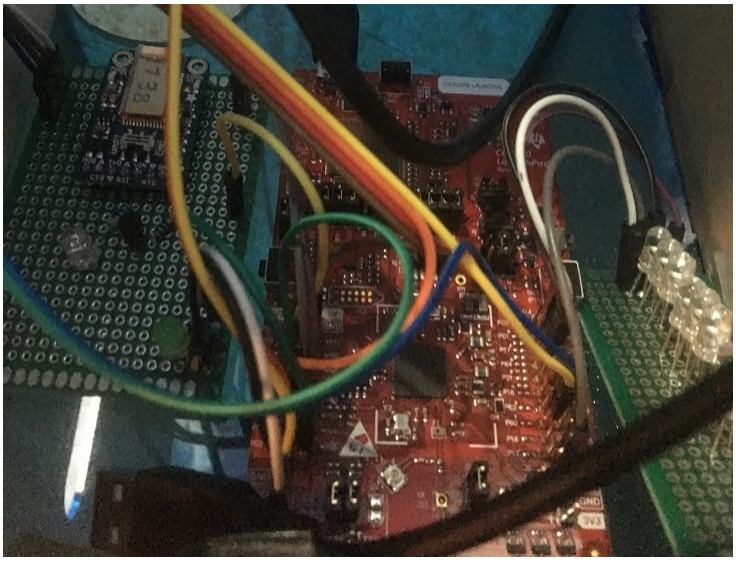


Figure 3: Soldered Board with Blue Tooth Beacon, Sensor, and LEDs

The blue tooth beacon is a Adafruit Bluefruit LE UART Friend - Blue tooth Low Energy. Each blue tooth beacon is programmed with a unique key through a phone app. When the beacon is supplied with power, this string of numbers is sent through the beacon. For this, the microcontroller sends a signal to an N-channel mosfet transistor which allows 5V power to be supplied to the blue tooth beacon. This saved some 5V I/O pins, which are much more limited than the 3.3V I/O pins.

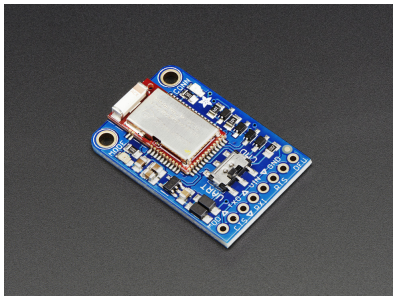


Figure 4: The Adafruit Bluefruit LE UART Friend - Blue tooth Low Energy

The sensor unit will communicate the spot status with the user through RGB LEDs. With this system the LEDs turn blue when the spot is busy, green for 20 seconds when the spot is successfully closed, red when the spot is illegal, and off during any idle time where the spot is either empty or legally closed. The LEDs are set in parallel strings of 10 LEDs each. Setting up the LEDs in parallel allows the system of lights to all use the same voltage to increase the brightness overall while not using as many pins on the microcontroller. This allows our 3.3V or 5V power rails to illuminate all of the LEDs as opposed to in series where the LEDs would need between 30-50V to power one string of ten. Two of the LED strings will be on the same level as the microcontroller approximately 32 inches from the ground and the last string will be closer to the top at 42 inches above the ground. The idea of this system is to make the top of the parking sensor as bright as possible so that the user can see the colors from their car.

Currently the sensor unit is powered through a wall outlet to the microcontroller and the rest of the components are powered through the sensor. While this will likely not get done by the time we finish this project, future work would be to incorporate other power sources such as batteries and solar panels. This will allow each unit to be powered independently and almost self completely self reliant except for damage done to the sensor. In order to save power, the micro-controller goes into a hibernation mode after every two cycles of use. During this hibernation mode, the I/O pins are disabled and the

chip is basically shutdown except for a hibernation register to store values and timer to count through the rest time. After this rest time, the microcontroller will wake up and resume normal processes.

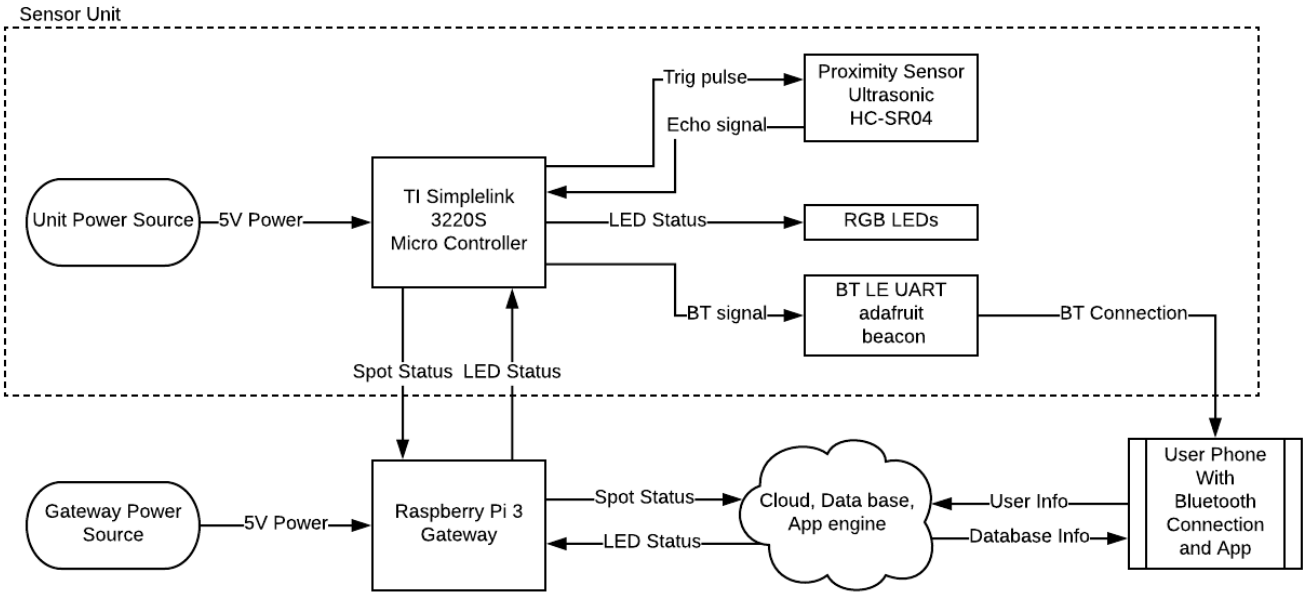


Figure 5: Hardware Block Diagram

Each sensor unit will communicate with the gateways through Wi-Fi socket connections to send status messages. To send any message, the microcontroller will first establish a socket connection to the gateway through Wi-Fi. The microcontroller also sets up the socket to have a time out of 2 minutes meaning that if the microcontroller fails to establish a connection or the gateway never sends a message back after the timer finishes then the connection will be terminated. The micro will try to reestablish a connection to the gateway to retry the process. If a user attempts to park while the sensor unit cannot connect or communicate with the gateway correctly, the spot status will become illegal after the timer finishes because there is no way to communicate that the user has the proper requirements to park legally.

Every gateway will be connected to 10 sensor units and will facilitate communication between the sensors and the cloud. The gateway will receive message from and send messages to each of the sensors as well as communicate the spot statuses with the cloud. Messages are sent for every event, like the user parking or leaving, and a standard heartbeat message is sent so the gateway can tell if the sensors are working properly.

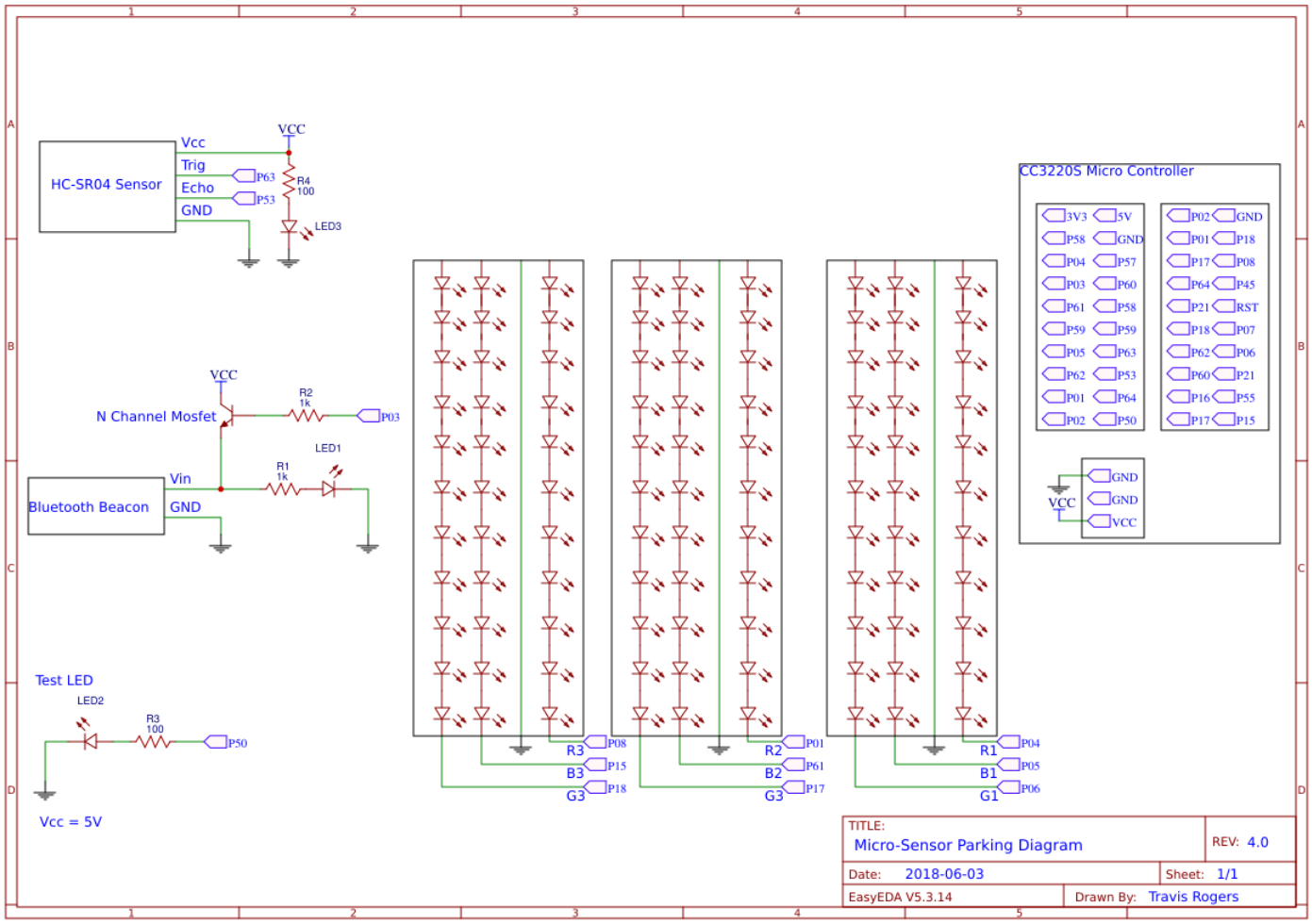


Figure 6: Wiring Diagram for micro and sensor

While idle, the microcontroller will send out the TRIG pulse and take in the ECHO pulse from the sensor. If the pulse is too large, meaning there is nothing in the desired range, then nothing happens to the LEDs and there is no connection attempt with the gateway. Around every 5 minutes, the sensor will connect to the gateway to send a heartbeat message showing that there has been no change since the last sent message. The heartbeat messages allow the gateway to determine the health of the sensor units. If there is a period of time where the gateway receives no messages from the sensors then we can assume that part of the process is broken.

When a user comes to park in front of the sensor, the car will block the sensor unit and the ECHO pulse will become short enough to fall within the desired range. This starts a 5 second counter; is the object remains within the desired range for the 5 seconds then the spot will be considered busy. The LEDs turn blue to signify to the user that the process can begin and the blue tooth beacon turns on. The sensor also sets up communication with the gateway through Wi-Fi socket relations and sends the new status message that the spot is busy. The sensor then waits for the gateway to send a message back. After the user does the necessary steps with their blue tooth device, the gateway will send a message allowing the user to park. The LEDs will turn green for 20 seconds to communicate to the user that the process was a success and this ends the process. When the user returns and leaves with their car, the sensor will be freed and the microcontroller will send a message notifying the gateway, no user input is required for this step besides leaving.

If the object leaves before the 5 second timer then we can assume that this was not a car but a person walking by or some other object passing through. This is for denouncing so that we try to only let the spot be busy when a user is parking.

If the user waits more than 2 minutes to verify then the spot status will change to illegal and the LEDs will turn red. The microcontroller will send a message to the gateway that the spot is illegal and this will update the rest of the levels of the project. This prevents users without proper certifications to illegally park in spots.

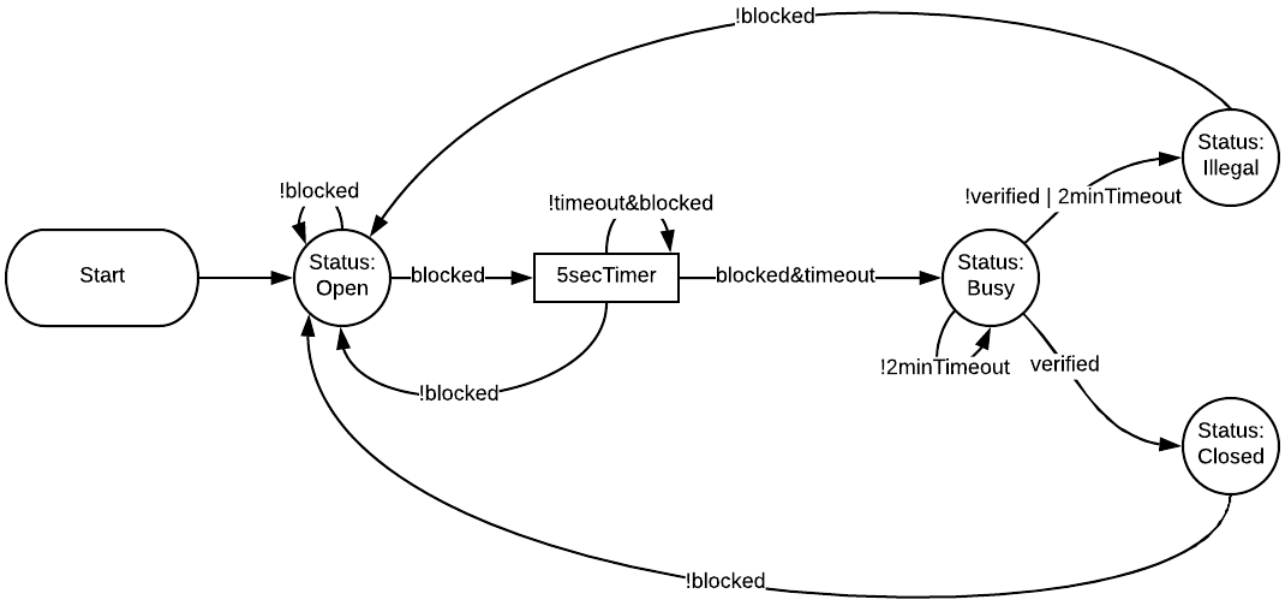


Figure 7: State Machine to Demonstrate the Logic for Spot Status

2.3 Body

The body is the physical parking sensor box that the user will park in front of to begin the verification process. The body will be made out of eighth inch thick acrylic that will allow the LED strings to illuminate the top of the structure to communicate effectively with the user. The base is a 1'x1' square with tabs in an 8"x8" square in the middle for the rest of the body which stabilizes the unit as a whole. The main body of the unit is a 8"x8"x4' rectangular prism with a floor at 30" from the ground for the microcontroller, two strings of LEDs, and the solder board to rest. The sensor will fit in to two holes on the front of the unit 32" from the ground. These two holes perfectly fit the transceiver and receiver for the sensor and will be the only parts sticking out of the sensor. For our prototype purposes, the sides will have handles 17.5" from the ground on both sides so that we can move the unit easily. Also for the purpose of prototyping, the back has a 6" door 28" from the ground to access the sensor, microcontroller, and perf board easily. The front and back only extend vertically for 42" off of the ground; the remaining 6" are used for a triangular prism designed to show the light form the LEDs better, the final string of LEDs will be at this level. The triangular faces of the prism will face the sides and while the rectangular faces will face the front and back so that the LEDs will illuminate this prism the most. The four walls that create the main body will be held together with glue while using the tab and slot method so all of the parts fit nicely.

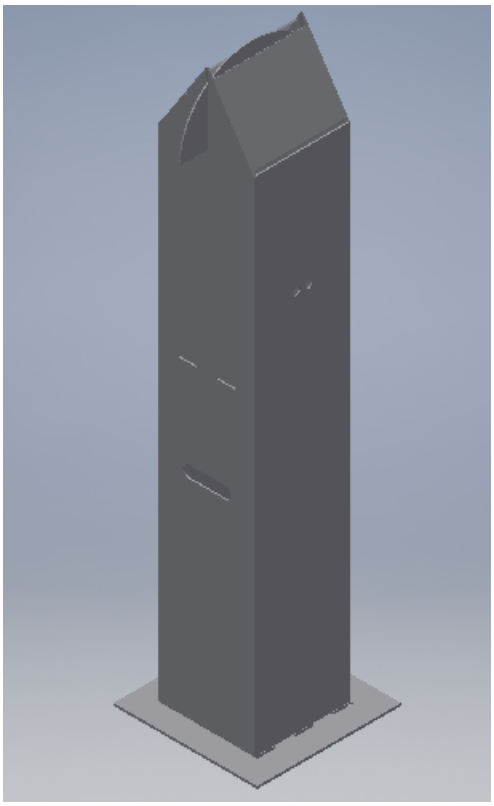


Figure 8: 3D CAD Model of the Sensor Body

Figure 9: Acrylic Body of the Sensor Body

2.4 Power Consumption

Power requirements can be seen in the following figures:
During normal (idle) operation, the whole sensor unit consumes a total of 710μA while actively on and then 4.5μA while in hibernation mode. For power management, the spot unit is on for twenty-five percent of the 13 hour day where parking is

enforced on campus, from this we can calculate the miliamp hours required to simply run the sensor through an entire day:

$$I_{idle} = 710 * 10^{-6}A \qquad I_{hib} = 4.5 * 10^{-6}A$$

$$1 \quad Day = 13 \quad Hours \qquad 25\% \quad idle$$

idle:

$$.25(13) = 3.25Hr = t$$

$$mAh = 710 * 10^{-6}A * 3.25Hr = 2.3mAh$$

hibernate:

$$.75(13) = 9.75Hr = t$$

$$mAh = 4.5 * 10^{-6}A * 9.75Hr = .044mAh$$

For sending a message, meaning that there is an event going on sending the message requires 223mA while receiving a message requires 59mA. For an event we are assuming that the event will take 20 seconds total where there is 1 second for sending the message, 18 seconds waiting for the user input in receive mode and then 1 second to actually receive the message. After this the sensor will return to its low power idle and hibernation operation.

$$I_{tx} = 223 * 10^{-3}A \qquad I_{rx} = 59 * 10^{-3}A$$

message:

$$\frac{1s}{3600s/Hr} = .000278Hr \qquad \frac{18s}{3600s/Hr} = .005$$

$$mAh_{tx} = .000278 * 223 * 10^{-3} = .062mAh$$

$$mAh_{user} = .005 * 59 * 10^{-3} = .295mAh$$

$$mAh_{rx} = .000278 * 59 * 10^{-3} = .0164mAh$$

$$mAh_{message} = .062 + .295 + .0164 = .3734mAh$$

For our power we assumed having a battery source configured to supply 6V and 3000 mAh. To find how long our system can last with this, we divide this expected mAh by the sum of the system mAh while idle all day adding in 'x' number of events.

$$Days = \frac{3000}{2.344 + .3734x}$$

This graph produced these results for one spot sensor:

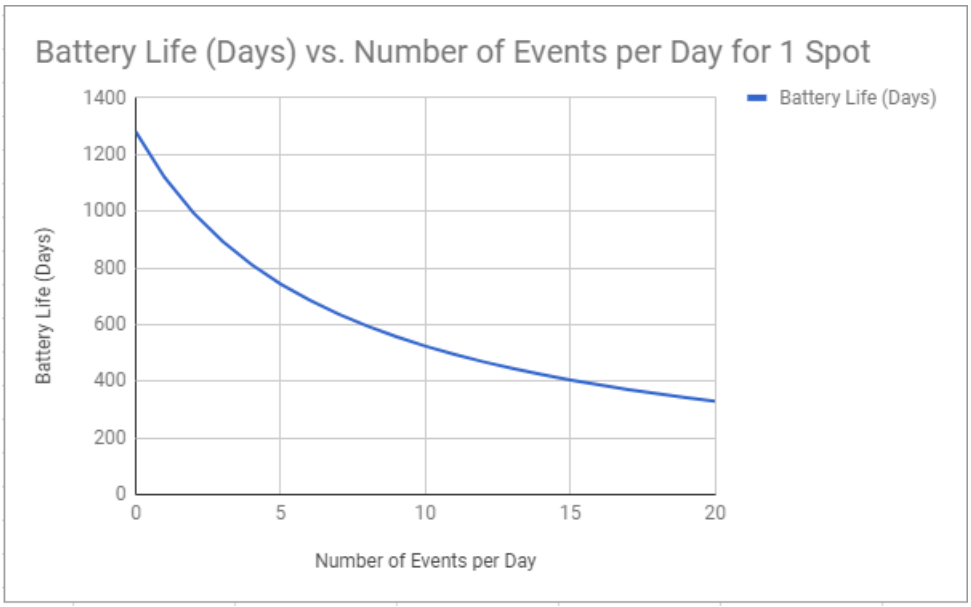


Figure 10: Power Consumption depending on events

From this we can derive that with no events, each spot sensor will last close to 4 years and that the more events per day cause a logarithmic decline in how long the sensor can last. 20 events for one day was our highest expectation for the sensor since the sensors would only be active for 13 hours of the day and since most parking on campus will be for classes and the user will be parked for more than an hour. For an average day, around 7-10 events per day, each sensor will still last more than a year with our power configuration.

2.5 Gateway - Ramzey

The Gateway is a low power device powerful enough to handle gathering, sending, and receiving messages to sensors and the App Engine. We use the Raspberry Pi 3 B's duel core CPU for its threading capabilities under the Raspian OS, which also allows us to use the network programming APIs in the C programming language.

Figure 11: Raspberry Pi 3 B

The Raspberry Pi's purpose is to be a gateway between sensors and the App Engine. This is done using Wi-Fi from the sensor to the Gateway and the Gateway to the App Engine. The Gateway can handle up to 10 threads at one time, meaning 10 users can park simultaneously within the same Gateway. Because the Wi-Fi's range is limited, mutliple Gateways may be used to cover the entire area of a parking lot or floor.

2.6 Google Cloud App Engine - Jon

To facilitate communication between the Gateways, Database, Mobile App, and Admin Webiste, we use Google's App Engine to execute application logic and provide connectivity between devices. When a microcontroller determines that a spot is occupied, the sensor status is changed and sent to the Gateway. The Gateway sends the aggregated sensor updates to the App Engine as an HTTP POST message, saving the updates to the Database and triggering a verification event for that spot. The Mobile App will read the bluetooth beacon string from the microcontroller and send it, along with its user ID, to the App Engine, which will match the Bluetooth string from the user with the blocked spot's unique string in the Database. If valid, the App Engine will send a success message back to the user app indicating a successful parking if valid, or indicate a payment option for those without the right permit. If the user does not complete a payment option by a certain period of time, in this case 10 minutes, or if the user cannot park in that spot, the App Engine will switch the parking spot status to "illegal".

The app.yaml file in the directory of the App Engine is responsible for routing requests to the correct handlers. A GET request to the website "http://car-management-66420.appspot.com" returns the administrator website. Appending the URL routes requests to different handlers. For example, appending the URL with "/gateway" routes the request to a handler that handles the gateway's HTTP POST message, while "/app" responds to the mobile app's POST message. Below is a diagram of the HTTP messages between the Gateway and Mobile App to the App Engine:

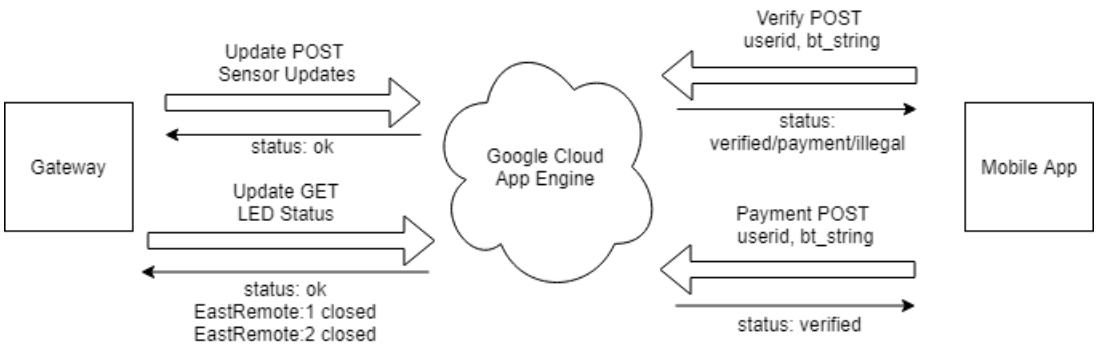


Figure 12: HTTP Messages

The App Engine uses webapp2, a Python web framework with WSGI capabilities, to handle HTTP requests. The module

MySQLdb is used to interface with the Google Cloud MySQL database. All other modules used are within the Python2.7 Standard Library.

Each interaction between the App Engine and the other devices are listed below:

1. **Gateway** - The App Engine receives aggregated sensor information from the gateway, updates the new statuses to the database, and determines if the user needs to be validated. The gateway will take sensor data and wrap it into a HTTP POST message for the App Engine to receive and parse through. The App Engine receives the HTTP POST message in the form of a dictionary containing parking lot/spot keys and status string values. It then checks these updates with the database entries to determine which spots need to be verified. The App Engine returns the statuses of parking spot to the gateway. If the spot status relies on verification from the App Engine, The gateway will send GET requests every 5 seconds as it waits for the user to verify with the Cloud App Engine through our Android App.

2. **Mobile App** - After determining that a spot needs to be verified, the App Engine waits for a HTTP POST from the mobile app of the user, which contains user identification and the bluetooth beacon string it received from the microcontroller. The POST message contains "email" and "bt_string" keys for identification of the user and the spot. The App Engine verifies that the user's bluetooth string matches the spot's bluetooth string to check if the user is in the correct spot. It then looks up the user's account from the database. If the user's database matches with the spot type, the parking is valid and the App Engine returns a string to indicate to the mobile app that parking is successful. If the permit does not match, but the user can still park there, the App Engine returns a string to the mobile app that indicates that it needs to pay the daily rate to park there. If the user cannot park there, a string is sent to indicate that the user cannot park there.

3. **Database** - Every time new updates arrive from the gateway or HTTP POST messages arrive from users, the App Engine will need to access the database to compare information and update entries. This will be done using the MySQLdb module and executing SQL commands through the cursor object. The App Engine parses the cursor field to receive the return information from the database. It also needs to update new entries from the gateway and account balances for payment to the database, which is done by executing the proper SQL commands in the function cursor.execute() function.

4. **Website** - The App Engine hosts the administrator website's page by serving requests with the proper HTML, CSS, and JavaScript files. It will respond to the website's requests for parking statuses, user account searches, modifications, additions, and deletions, and return transaction and occupation history logs.

The API section of this document includes more detail about the types of HTTP messages.

2.7 Database - Ramzey

The SQL database is responsible for holding data for a variety of information. The database for our project contains multiple tables for different categories we want to track. A description of each table is as follows:

1. **Spaces** - The Spaces table keeps track of all parking spaces, the occupation status of each space, the legality status of each space, the Bluetooth string associated with each space, the corresponding lot for each space, the daily cost and associated permits for each space, and the current user parked in each space. The list of fields can be seen in Table 13

Field	Type	Null	Key	Default	Extra
Spots	int(11)	YES		NULL	
Status	varchar(20)	YES		NULL	
BT_String	varchar(88)	YES		NULL	
Spot_Type	varchar(20)	YES		NULL	
Daily_Rate	decimal(12,2)	YES		NULL	
Parking_Lot	varchar(25)	YES		NULL	
User_ID	int(11)	YES		NULL	
Complete	int(11)	YES		NULL	
Gateway	int(11)	YES		NULL	

Figure 13: All entries in the Spaces Table

2. **Users** - The Users table keeps track of all users along with their corresponding permit types and account balances. Figure 14 describes all fields in the Users table

Field	Type	Null	Key	Default	Extra
Last_Name	varchar(255)	YES		NULL	
First_Name	varchar(255)	YES		NULL	
id	int(11)	YES		NULL	
Permit_Type	varchar(10)	YES		NULL	
E_mail	varchar(50)	YES		NULL	
Account_Balance	decimal(12,2)	YES		NULL	
Password	varchar(128)	YES		NULL	
Log_In_Status	int(11)	YES		0	
License_Plate	varchar(10)	YES		NULL	

Figure 14: All entries in the Users Table

3. **Transaction History** - The Transaction History table keeps track of all transactions made. Each transaction records the user ID, time stamp, payment amount, transaction type, parking spot and parking lot. Every time a user leaves a space or enters a space, the data is logged in this table. The list of fields can be seen in table 15

Field	Type	Null	Key	Default	Extra
User_ID	int(11)	YES		NULL	
Time_Stamp	varchar(25)	YES		NULL	
Payment_Ammount	int(11)	YES		NULL	
Transaction_Type	varchar(20)	YES		NULL	
Spaces	int(11)	YES		NULL	
Parking_Lot	varchar(25)	YES		NULL	

Figure 15: All entries in the Transaction History Table

4. **Occupation History** - The Occupation History table keeps track of the user history and corresponding time stamps for each parking space. It records who and when parking spaces are being occupied along with which specific spots are being used the most. The table entries can be seen below in table 16

Field	Type	Null	Key	Default	Extra
Spots	int(11)	YES		NULL	
Status	varchar(20)	YES		NULL	
Time_Stamp	varchar(50)	YES		NULL	
Parking_Lot	varchar(25)	YES		NULL	
id	int(11)	YES		NULL	

Figure 16: All entries in the Occupation History Table

2.8 Mobile App - Mario

The Mobile Application will have two main objectives. The first objective requires the app to pull information from the database such as number of empty spots per lot, current lot/space number the user is parked in, user account balance, current

user permit, and so on. This is to add usability of the app while the user is not actively trying to park and view current account information or parking information. The second objective requires the app to automate the user verification process when the user is actively parking in front of a parking unit. This requires the app to automatically read the bluetooth beacon and send HTTP POST requests to the App Engine for gathering proper payment/verification information.

The Mobile App layout will rely on a simple navigation drawer as seen in figure 17. Normally, each page on a mobile app uses what are called activities. However, for a navigation drawer, each page option the user can choose from in the drawer is composed of what are called fragments. In short, a fragment is a layer that sits on top of the activity that called it. As for the app it will consist of a single background activity with one or more fragments being layered on top of that single activity. The navigation drawer options will consist of fragments that will either display information for a specific parking lot and parking space, or display general user account information such as available permits and current account balance.

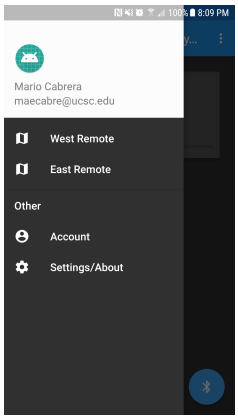


Figure 17: Navigation Drawer

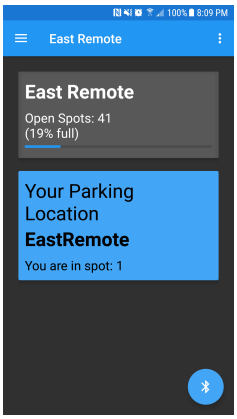


Figure 18: East Remote Fragment

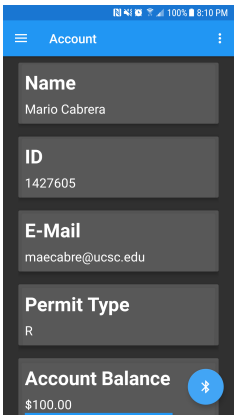


Figure 19: Account Fragment

For the automated verification process, the Mobile App is designed to immediately read the bluetooth beacon and send a JSON object containing the last 12 digits of the Bluetooth UUID and the user ID to the App Engine for verification. This process happens during app start up all within the background main activity. In the event where the post request failed, the bluetooth beacon was not read, or any other error occurred, the user can reread the bluetooth beacon and resend another post request using the action button. This is the button in the bottom right corners in figures 17, 18, and 19. Ideally, in the case where multiple bluetooth beacons are being detected at the same time by the Mobile App, a pop up window will appear that will ask the user which specific parking unit (or blueTooth beacon) they are trying to park in, and upon decision another post request is sent as well.

In terms of the verification process as a whole, considering three cases, there are a maximum of two post requests needed to be sent to the App Engine for parking verification.

1. **Permit Verification** After the initial post request is sent to the App Engine, the App Engine will then compare the user's permit with the permit required for the specific parking space they are parked in. If both permits match, then the App Engine will respond with a "verified" status to the mobile app. From the there the Mobile App will prompt the user that their parking has been verified.

2. **Payment Verification** If the user permit and the parking space permit don't match, then the App Engine will return with a "payment" status along with the user's current balance and daily rate. This will then trigger the Mobile App to prompt the user that a payment option is available for their current spot displaying the daily parking rate and the user's current balance. If the user decides "yes" and they would like to pay, then a second post request is sent to the App Engine indicating a payment has triggered. The App Engine will then deduct the daily rate from the user's account balance and respond with a "verified" status along with the user's new balance. Otherwise, if the user says no, then the mobile app will simply advise the user to move from the parking space because their place at the spot is considered illegal.

3. **Illegal Parking** The last case is where the user does not have the proper permit for a specific parking space and a payment option is not available for that specific spot such as with the case of a handicapped spot. In this specific case, the initial post request will return a status "illegal" to the Mobile App. From there, the app will prompt the user to leave the spot because they are illegally parked.

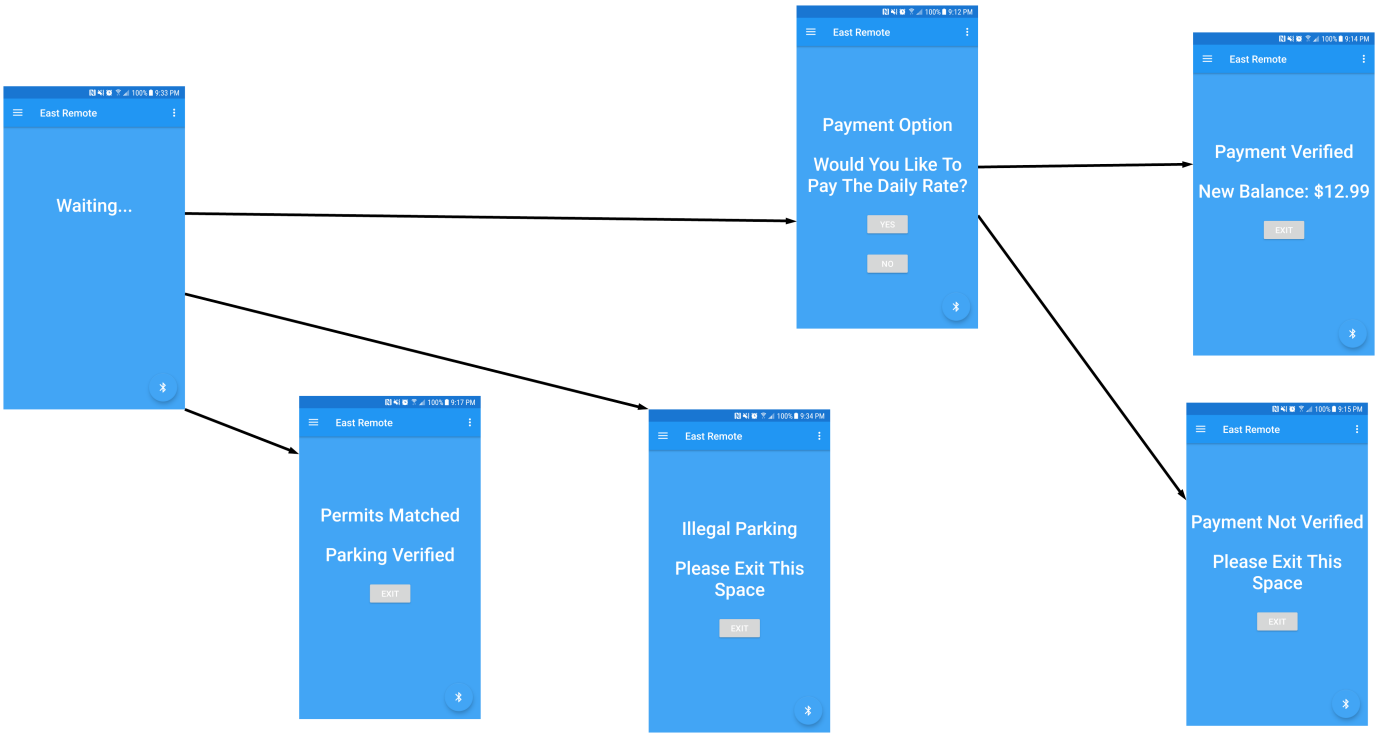


Figure 20: App Payment Process

2.9 Administrator Website - Ramzey, Jon

The Admin Website allows parking services to track the status of parking spaces in parking lots through a map. The admin can search, list, create, and edit user accounts from the database and track transaction and occupation history of the parking spots. The website will achieve these functions in multiple tabs:

1. **Parking Lot Map Tabs** - The map tab shows the status of parking spots in their corresponding parking lots. There will be tabs for each of the parking lots in the database. The color codes of the statuses are listed below:
   (a) **Green** - The spot is open and working.
   (b) **Blue** - The spot is verified and is currently being used.
   (c) **Yellow** - The spot is busy and in the process of being verified.
   (d) **Red** - The spot is being illegally used.
   (e) **Orange** - The gateway is not receiving sensor information from this spot, indicating that the sensor is broken.
   (f) **Grey** - The database is off.
2. **User Account Tab** - The account tab allows the admin to access user accounts and change fields in the database. Like the spot tab, admin can search for specific user or select from a list of users and edit the fields of the account, such as adding to the balance or changing a name. The page will have a "Create an account" button that shows a form to create and save a new account to the database.
3. **Logs Tab** - The logs tab allows the admin to view transaction and occupation entries in the Database. The tab provides a form that will search for specific days, IDs, permit types, etc.

3 APIs

The following section contains APIs for our Gateway, Database, App Engine, Mobile App, and microcontroller programming

3.1 Microcontroller

The microcontroller will send updates on the spot status to the gateway and interact with the sensor, bluetooth beacon, and LEDs. The microcontroller and gateway set up a network connection over Wi-Fi to send messages to each other to

properly update the spot status. The microcontroller interacts with the other sensor unit components through I/O pins in order to communicate the status with the user, turn on the bluetooth beacon for the app and detect when a user is attempting to park. A main Work() loop will control sending messages, and power management while events with the sensor cause interrupts so that the status can be real time.

3.1.1 The "Work" Loop

The main loop runs indefinitely while the the device is on and not in low power mode. Within this main loop, the function checks to see if a message needs to be sent by looking at the sendFlag. If this value is high then the program begins connecting to the gateway through the network through Wi-Fi. The main function has a counter for sending heartbeat messages, when the program enter hibernation mode, all previous values are reset except for a hibernation register which stores memory in low power (hibernation) mode. After checking to see if there is a message that needs to be sent, the program extracts the hibernation register value, increments it, and then stores this new value back into the register. If this counter has reached a certain point then the program sets the sendFlag high so that the next loop will send the heartbeat status message. If there is no event currently happening, such as the spot being busy and awaiting response from the user, the microcontroller will then enter hibernation mode with the following function:

```
powerShutdown(NOT_ACTIVE_DURATION_MSEC);
```

This puts the micrcontroller into hibernation mode for however long the duration is set to. For this project the microcontroller is on for one loop (approximately one second) and then off for the next three seconds. This greatly helps with power management since the sensor and microcontroller are only out of lower power mode for twenty five percent of the time. After the three seconds, them the microcontroller enters its normal power mode and repeats the loop again.

3.1.2 Network Connection to the Gateway

In order to connect to the gateway, the microcontroller must set up a Wi-Fi connection over the same network and then connect to the socket. The microcontroller will send out a message in the following format:

```
ChangedBit ParkingLot:ParkingSpace Status
```

For this message format:

1. ChangedBit: This is either 1 or 0 where 1 means that an event has happened and that there is a change in the spot status and 0 means that there has been no change. The changed bit will only be 0 for heartbeat messages that assure the gateway that the sensor is still working. If the gateway does not receive any messages after a certain period of time, then we can assume that something is wrong with either the gateway or sensor.
2. ParkingLot: This is just the name of the parking lot that the sensor is in. An example of this would be EastRemote for the east remote parking lot on the UCSC campus.
3. ParkingSpace: This is the parking spot number, each spot sensor will have their own number, this allows the gateway to distinguish between spots.
4. Status: This is the spots current status, either open, closed, busy or illegal.

The microcontroller will send messages for every event that occurs with the spot and every 10 minutes for heartbeat messages.

After sending the message to the gateway the microcontroller will wait to receive a message back from the gateway. These messages can be any of the following:

1. closed: This means that the user parked successfully. The status will be set to closed and the LEDs will turn green communicating to the user that they parked successfully.
2. illegal: This means that the user did not park successfully. Either the gateway timed out waiting for the user to respond or the user did not have the proper credentials to park. The status will be set to illegal and the LEDs will turn red to communicate the spot status with the user and any parking officials.
3. ACK: This is a response to a heartbeat message. This changes nothing for the status but just lets the sensor unit know that the heartbeat message was sent successfully.

If the return message is not any of these then there is an error and the LEDs turn purple and the process needs to be re-attempted.

In order to properly do these tasks the microcontroller uses these functions:

1. `void Wifi_Connect(void)`

   This function initializes the microcontroller connecting to Wi-Fi. It uses the saved SSID and password to start the connection. The connection itself is used with an event handler which uses WLANN_CONNECT to signal that the connection was successful.

   In order to signal when the microcontroller needs to send a message, there is a sendFlag variable that is set high whenever an event occurs for the spot or when it is time to send a heartbeat message. In the main Work() function, when the sendFlag is high, the microcontroller starts connecting to the socket:

2. `void Socket_Connect(void)`

   This function sets up the socket connection to the gateway using the sl_socket library from TI. This function also sets up a two minute connection timer, if the connection is made and either sending or receiving takes longer than two minutes, the program will terminate the connection and then try to reconnect to the socket.
   After establishing the connection, the next step is to call the command:

3. `void SendStatusMessage(void)`

   This function formats and concatenates all of the strings, in their current state, to create the status message. By having different strings for each part of the message format, the program can easily update parts like the ChangedBit and the Status to any of the available values. After making the status message the function calls:

4. `void TX_Send(void)`

   This is the socket transfer function. With the socket connection established, the function sends out the status message in the form of a string buffer. Within this function there is a counter that increments on attempts to transfer the string, if the microcontroller is unable to send the data after five attempts then the program terminates the connection and tries to establish a new one and send the message again. Immediately after successfully sending, the program executes this function:

5. `void RX_Recieve(void)`

   This is the socket receive function. With the socket connection established and after sending the status message, this function waits to receive data from the gateway within the two minute timer. This message is stored into a received message string buffer where it gets interpreted in this function:

6. `void Check_Response(char *message)`

   This function takes in the message from the gateway and interprets it by changing the color of the LEDs or attempting to reestablish the connection and resend the status message if the response was something different from what was previously described.

3.1.3 Sensor and ISRs

While the main loop runs constantly as long as the device is on and not in low power mode there is an interrupt service routine (ISR) running waiting for the sensor. Since the ECHO pin of the sensor uses pulses to determine distance there is an interrupt for both the rising and falling edge of the ECHO pulse:

`void Echo_NoTimer_Callback(uint_least8_t index)`

This function starts by reading in the ECHO pin to see if this event is a rising or falling edge. If it is a rising edge then the interrupt starts a timer and when the interrupt is for a falling edge then the timer is stopped. That time is then sent through the function:

```
int GetDistance(uint32_t distime)
```

Which compares this pulse width to the threshold set as a `#define` value. If the value is less than the program increments a counter. If this counter reaches five, meaning that the sensor has been blocked for five seconds, then the program will assume that there is a user trying to park and the interrupt sets the sendFlag high. If this is an event, meaning that the spot status just changed, this this also sets the ChangedBit string to 1 and changes the Status accordingly. If at any point during this process, the pulse time becomes higher than the threshold then the process stops and the spot is still in the open state. When the user leaves, then the pulse time will be larger than the threshold and this event will also trigger the sendFlag to become high.

When the spot suddenly becomes busy, then the micrcontroller will turn on the bluetooth beacon and change the LEDs color to blue signaling to the user that the spot is now busy and they can continue the parking process. Since the beacon and LEDs are connected to the microcontroller through I/O pins, turning these on and off simply uses the TI `GPIO_Write(I/O pin, value)` function. To change the color of the LED strings, the program calls this function:

```
void RGB_Color(int color)
```

Which takes in a defined color and uses a switch statement to change what pins for the LEDs become high or low.

3.2 Gateway - Ramzey

This part of the project involves a gateway to take in data from many microcontrollers and send the data together to the App Engine so the app engine could do computations based on this data. There are two parts to the program. Each side has its own message format. The server side acts as a server to all the sensors and gathers data in a specific message format while the client side formats the message into standard JSON key value pairs. Detailed information an be found bellow.

1. **Server** - The server portion of the Gateway uses a TCP connection that takes in the status of a particular parking spot from all the microcontrollers and reads it into a file. The format of both the data arriving and the format of the file is:

   `ChangedBit ParkingLot:ParkingSpace Status`

   (a) `ChangedBit` - This bit indicates whether or not the microcontroller has changed its state. (Sensors can send keep alive messages to indicate they are still functioning properly)

   (b) `ParkingLot` - This is the parking lot ID the microcontroller is in.

   (c) `ParkingSpace` - the ID of the parking space the microcontroller is in.

   (d) `Status` - The current status of the spot. This can be busy, closed or open.

   i. open - The spot is open and waiting for someone to park.

   ii. busy - Someone has parked and the verification process of a user or the awaiting of a purchase is happening.

   iii. closed - Authentication has been successful and the spot is taken

   iv. broken - If the microcontroller cannot be contacted, this state

   All parking spaces and statuses under the gateway's control will be in this file. It will be the App Engine's job to access the database and determine if the status of a spot has changed.

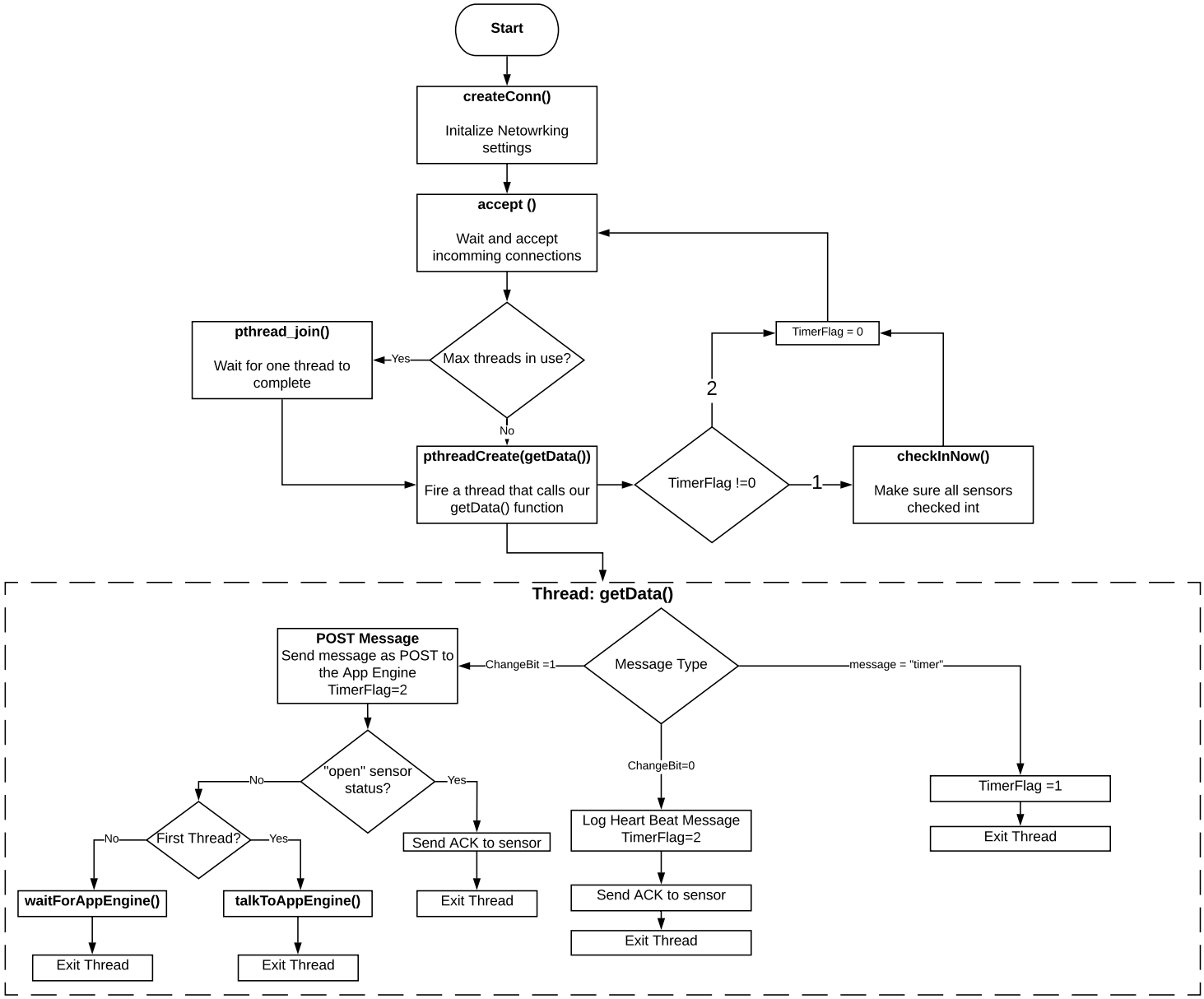
2. **Client** - The gateway client will format the data and send it to the App engine though HTTP POST messages. Once the data is received and the message is not a heart beat message, standard HTTP POST headers are added to a buffer, followed by the message received from the sensor in JSON form. The JSON format we are using involves the Parking lot and spot number as the key and the status of the spot as the value. This gives us a key value pair.

   `{"ParkingLot:ParkingSpace" : "Status"}`

   This message is then sent from a thread. After this, a HTTP GET request is sent periodically to request a verification response from the App Engine. Even if multiple threads are sending POST sensor messages to the App Engine, only one GET request is needed to get verification for all sensors. Because the App Engine cannot hold or make a connection, a periodic GET request is necessary to ask the App Engine for the verification data.

To keep track of broken sensors, every sensor sends a heart beat message every 15 minutes. If this is just a heart beat message, then the changed bit will be 0 since the spot status did not change. The check-ins with the gateway will be saved to memory where the program will then iterate through memory to ensure all sensors have checked in. If a sensor has not checked in more than 2 times in a row, the program will send the JSON POST request for that spot with a Status of “broken”.

The Diagram on the next page describes the process for the gateway’s operation.


Start
createConn()
Initalize Netowrking settings
accept ()
Wait and accept incomming connections
Max threads in use?
Yes
pthread_join()
Wait for one thread to complete
No
pthreadCreate(getData())
Fire a thread that calls our getData() function
TimerFlag !=0
2
1
TimerFlag = 0
checkInNow()
Make sure all sensors checked int
Thread: getData()
Message Type
ChangeBit =1
POST Message
Send message as POST to the App Engine
TimerFlag=2
ChangeBit=0
message = "timer"
"open" sensor status?
No
Yes
First Thread?
No
Yes
waitForAppEngine()
talkToAppEngine()
Exit Thread
Exit Thread
Send ACK to sensor
Exit Thread
Log Heart Beat Message
TimerFlag=2
Send ACK to sensor
Exit Thread
TimerFlag =1
Exit Thread

A detailed description of each block is described here:

1. **void createConn(int port)** - This wrapper function is responsible for creating TCP connections based of a provided port number that defaults to 1111.The function establishes a connection with the following UNIX/Linux library networking functions:

   (a) `int socket(AF_INET, SOCK_STREAM, 0)` - This function creates a socket on a provided port. The first parameter specifies the socket as an IPv4 socket, and the second argument specifies a TCP connection. There are no special flags bing used, so the third parameter is 0. The file descriptor of this port is returned so one can receive from or send to it in the future

   (b) `setsockopt()` - This function is used to allow the program to rebind to the same port number once the program exits. This makes it so we don't have to wait for the TCP connection to end after we exit the program when debugging.

   (c) `bind(socketFD, (struct sockaddr *) &server, sizeof(server))` - This function binds the server's `sockaddr_in` properties (IP address, connection type, and port number) with to socket file that was created with the `socket()` function. First, the file descriptor of the socket we are using is passed. Next, we cast our `sockaddr_in` to a sockaddr as the second parameter. Lastly, we provide the size of the `sockaddr_in` structure.

   (d) `listen(socketFD,501)` - This function tells the program to start listening for incoming connection on the socket file descriptor we created (first argument) and allow up to 501 connections (second argument) at one time.

2. **accept()** After a networking initializations are set up by the `createConn()` function, the main program waits to accept incoming connections Once an incoming connection is accepted with the UNIX/LINUX standard library's `accept()` function, a thread is fired.

3. **void* getData(void* tData)** - When a thread is fired, a function must be called to run in the new thread. This function is our `getData()`. Because of threading API limitations, only one parameter can be passed into this thread function. As a result, all thread data will be sent in a structure called `tdata`. This function will receive the status of the microcontroller and save the information to a file, while sending an ACK. The parameter of this function is a structure of type `tdata`. This structure has all variables necessary to get data from a particular microcontroller. A breakdown of the tdata structure can be seen here:

   `struct tdata`

   (a) `char *buffer` - This is the main buffer where the data is received in JSON format.

   (b) `int socketFD` - The socket that is being used to communicate between the client and gateway.

4. **void talkToAppEngine(tdata *threadData, int socketAE, char *array[2], char newbuff[MAX])** - The goal is to only have 1 thread communicate to the App Engine to save costs on communication and message data.Thread 1 will always talk to the App Engine by setting up an GET request, and continously pinging the App Engine until a response is received or a timeout of 10 minutes occurs. All other threads will be waiting in a loop in the `waitForAppEngine()` function. When Data is recieved from the App Engine, the JSON message is parsed and saved into shared memory. Each thread looks through the data and looks for their particular spot information. If the spot is found, the data (LED status) is sent to that particular SPOT unit. Otherwise the threads continue to wait and `talkToAppEngine()` continues to talk to the App Engine. Only when all SPOT information is recieved (or a timeout occurs) does the first thread exit.

5. **void waitForAppEngine(tdata *threadData, char *array[2])**- As mentioned previosuly, All threads except the first go to this function and wait for LED status messages to arrive for that thread's particular SPOT unit. After the JSON data is parsed by Thread 1, Thread 1 signals all waiting threads to check shared memory for data on their particular spot. If the thread's LED SPOT status is found, the data is forwarded to the proper SPOT unit. The SPOT unit then closes the connection to the gateway.

6. **void handleAppEngineTimeOut(tdata *threadData, int socketAE, char *array[2])** - This function's job is to send "illegal" messages to all sensors when no response is recieved from the App Engine when a timeout occurs. Usually we have the timeout set to 10 minutes. If a timeout occurs then the user never verified their parking permit with our system, and are declared illegal. This function tells the App Engine the spot is illegal as well so it can be logged and viewed on the Administrator Website.

7. **void checkInNow()** - This function's job is to make sure all sensors have sent a heartbeat or keep alive message. On the Gateway, a second process is running that simply sends a "message: "timer" and immediately sleeps for 15 minutes. When the main server process gets this message, checkInNow() is called. By this time, all parking spots have messged in.

3.3 App Engine

The App Engine consists of a WSGI (Web Sever Gateway Interface) application that handles HTTP GET and POST requests from the Gateaway, Mobile App, and admin website, as well as access the SQL Database using MySQLdb. All HTTP data sent and received are in the form of JSON objects.

3.3.1 Message Types

HTTP GET and POST messages are defined by a header field called "Message_Type", which indicates the type of message the App Engine is receiving. There are 4 types:

1. **Update** - Send and receive updated sensor status information
2. **Verify** - Verify a user's parking spot
3. **Payment** - Complete payment of parking spot
4. **Spots** - Returns the number of empty spots in a parking lot

The Update messages are for the Gateway to send updates to the App Engine while the Verify and Payment messages are used for the user verification step in the Mobile App.

3.3.2 Gateway Protocol

The Gateway Protocol deals with the sending and receiving data from Update HTTP GET and POST messages that facillitate the updating of sensor information and receiving LED status information for the sensor units. The Gateway sends POST messages to send updated sensor statuses to the App Engine. The Gateway also sends GET messages periodically whenever one of its connected microcontrollers is "busy" and needs to be verified.

```
POST car-management-66420.appspot.com/gateway HTTP/1.1
                    Message_Type: Update
   "EastRemote:1" : "busy", "EastRemote:2" : "busy"
```

The POST message is sent to the App Engine URL address with path /gateway, which routes the message to the gateway handler. There, the App Engine checks the message type for "Update" and converts the JSON object into a dictionary. It then iterates through the entries and updates the status of those spots in the database.

```
POST car-management-66420.appspot.com/gateway 200 OK
                    Message_Type: Update
                     "status" : "ok" or
                     "status" : "error"
```

The method returns a JSON object with "status" set to "ok" to notify a successful database update. If the `Message_Type` header is not "Update", the method returns status "error".

```
GET car-management-66420.appspot.com/gateway HTTP/1.1
          Message_Type: Update, Gateway: 1
```

The GET message from the Gateway to the App Engine requests a list of verified parking spaces corresponding to the Gateway header that have not been sent to the Gateway and sensor units. The Gateway, when receiving and sending a spot that is waiting to be verified, will periodically poll the App Engine for verified spots. The Gateway will continue this until all of its spots are no longer in the "busy" state.

```
GET car-management-66420.appspot.com/gateway 200 OK
                    Message_Type: Update
"status" : "ok", "EastRemote:1" : "closed", "EastRemote:2" : "closed" or
                     "status" : "none"
```

It will return a status key set to "ok" and a list of verified parking spots/status entries. If the status value is "none", the App Engine is indicating that no spots have been recently verified.

3.3.3 Mobile App Protocol

The Mobile App Protocol consists of HTTP POST messages and replies between the Mobile App and the App Engine. The Verify POST messages are used by the Mobile App to send user and spot information to the App Engine to verify the user's permit. The Payment POST message complete payments for those without the proper permits in the spot. The Spots GET message returns the number of empty spots in a specified parking lot.

```
POST car-management-66420.appspot.com/mobileapp HTTP/1.1
                Message_Type: Verify
   "userid" : "1429392", "bt_string" : "DEFJKL5564"
```

The Mobile App sends a POST message to path "/mobileapp" and a message type header set to "Verify", with its user ID and Bluetooth string it obtained from the sensor unit. The App Engine receives this and checks the database for the user account and spot ID. It will return a string that indicates the result of the verification:

```
POST car-management-66420.appspot.com/mobileapp 200 OK
                Message_Type: Verify
              "status" : "verified" or
               "status" : "illegal" or
"status" : "payment", "current_balance" : "85.25", "daily_rate" : "3.75"
```

The response tells the Mobile App to display a certain prompt. "verified" indicates that the user's parking spot is verified while "illegal" prompts the user to choose another parking spot. The "payment" response includes the current balance of the user and the daily rate of the user's spot. This triggers a payment prompt on the Mobile App that displays the balance and rate and indicates if the user wants to pay for parking. If not, the Mobile App urges the user to vacate the spot. If the user wants to pay, a Payment POST message is sent:

```
POST car-management-66420.appspot.com/payment HTTP/1.1
                Message_Type: Payment
  "userid" : "1429392", "bt_string", : "DEFJKL5564"
```

The POST message is sent to the "/payment" handler and "Payment" message type. The message includes the user ID and bluetooth string to indicate which user account is being deducted. The App Engine will use the userid and bt_string to subtract the daily_rate amount from the account_balance of the user. The response is:

```
POST car-management-66420.appspot.com/payment 200 OK
                Message_Type: Payment
   "status" : "verified", "current_balance" : "96.25"
```

The response are keys indicating a successful payment and its current balance to the user.

```
GET car-management-66420.appspot.com/spots?Parking_Lot=EastRemote HTTP/1.1
                Message_Type: Spots
```

To retrieve the number of empty spots in a parking lot, the Mobile App sends a GET message to the spots "path" and includes the parking lot query.

```
GET car-management-66420.appspot.com/spots?Parking_Lot=EastRemote 200 OK
                Message_Type: Spots
            "status" : "ok", "Spots" : "55"
```

The response are keys indicating a successful message and Spots that gives the number of empty spots in the specified parking lot.

```
POST car-management-66420.appspot.com/login HTTP/1.1
"email" : "jokwlam@ucsc.edu", "password" : "example"
```

The login message sends an email and password string to the App Engine.

```
POST car-management-66420.appspot.com/login 200 OK
     "status" : "ok", "login" : "verified" or
                "login" : "failed"
```

The POST message response is a status key and a login key that indicates if the login information is successful

```
POST car-management-66420.appspot.com/mobileinfo?email=jokwlam@ucsc.edu&password=example HTTP/1.1
```

The Mobile App sends a POST message with a email and password.

```
POST car-management-66420.appspot.com/mobileinfo 200 OK
"status" : "ok", "First_Name" : "Jonathan", "Last_Name" : "Lam"
"email" : "jokwlam@ucsc.edu", "id" : "1429392"
"Permit_Type" : "R", "Account_Balance" : "100.00", "Log_In_Status" : "0", "License_Plate" : "6EDT973"
```

The POST message responds with the user's account information, such as first and last name, email, id, permit type, account balance, login status, and license plate. The status key returns "error" if the email and password combination fails.

3.3.4 Admin Website Protocol

The Admin Website Protocol deals with handling website requests and manipulating Database data from the website.

```
GET car-management-66420.appspot.com/get_spotinfo HTTP/1.1?Parking_Lot=EastRemote&Spots=1
```

The GET message for get_spotinfo contains a query string for Parking_Lot and Spots to get information about the spot. This used for the admin website map.

```
GET car-management-66420.appspot.com/get_spotinfo?Parking_Lot=EastRemote 200 OK
"status" : "ok", "Spot_Type" : "R", "Status" : "busy"
"User_ID" : "1393696", "Daily_Rate" : "3.75", "Permit_Type" : "R"
```

The App Engine responds with a status of the message and spot information about the specific spot in the parking lot. This includes the spot type, status of the spot, user ID of the currently parked user, daily rate, and permit type of the user.

```
GET car-management-66420.appspot.com/get_parkinglot?Parking_Lot=EastRemote HTTP/1.1
```

The website sends a GET request to get_parkinglot with a query line that specifies as Parking_Lot.

```
GET car-management-66420.appspot.com/get_parkinglot?Parking_Lot=EastRemote 200 OK
"1" : "open", "2" : "busy", "3" : "closed", "4" : "open"
```

The response is a JSON dictionary of spot numbers and their statuses. This is used for the parking map.

```
POST car-management-66420.appspot.com/account HTTP/1.1
"First_Name", "Last_Name", "id", "Permit_Type", "email", "License_Plate"
```

The website sends a POST message to the account handler that includes the search fields from the account search form, which includes the first and last name, id, permit type, email, and/or license plate.

```
POST car-management-66420.appspot.com/account 200 OK
forms.html template
```

The response is a web page containing the list of users matching the sent parameters. The App Engine will return all the data from each user, including first and last name, id, permit type, email, account balance, log in status, password, and license plate.

3.3.5 Database Protocol

The Database Protocol uses the "MySQLdb" module to connect to the database, execute SQL queries, and receive query outputs. SQL queries are called in wrapper functions that catch possible errors from execution.

1. `update_spaces(cursor, parkinglot, spot, status)`
   Updates an entry in the spaces table of the database. Using the Cursor object from the connected database, it uses a SQL UPDATE command to change the Status field of the entry with *status* at spot where field Parking_Lot is *parkinglot* and field Spot is *spot*. If the spot is transitioning to "open", the function selects the User_ID of the spot (if there is one) and resets the Log_In_Status of the user to 0. It then sets the User_ID field to NULL. This function is used for Updated POST replies from the Gateway.

2. `get_verified(cursor)`
   Returns a dictonary containing a status field and list of verified parking spots for the LED statuses of the sensor units. Using the Cursor object, it checks the spaces entry for "closed" parking spots with the Complete field set to 0, indicating that the spot is verfied but the LED has yet to indicate this change. The function gets the entries, formats the data into the "ParkingLot:ParkingSpace" as the key and "Status" as the value, sets status key to "ok", and returns the dictionary. If no entries are found, the status key is set to "none" and the dictionary is returned. This will be used for Update GET replies from the Gateway.

3. `verify_user(cursor, userid, bt_string)`
   Returns a dictionary containing a status field and other fields depending on the status. The function searches for the Spot_Type of the spot with *bt_string* and the permit of *userid*. It also checks if the spot is busy to see if the received Bluetooth string is waiting to verify. If the user permit matches one of the spot permit, the user is verified and the function updates various fields in the database, such as the spot status to "closed", the user ID of the spot to the user's ID, the Log_In_Status of the user to 1, and the Complete field set to 0. The dictionary's status field is set to "verified" and the dictionary is returned.
   If the permits do not match and the spot type is Handicapped, the App Engine will set the dictionary status field to "illegal" and return the dictionary.
   If the permits do not match but the spot type is not Handicapped, the user can choose to pay for daily parking. The dictionary status field is set to "payment", current_balance is set to the user's current balance, daily_rate is set to the daily rate of the parking spot, and the dictionary is returned.

4. `complete_payment(cursor, userid, bt_string)`
   Returns a dictionary containing a status and current balance field. The function uses the userid and bt_string to obtain the daily rate of the spot and the account balance of the user. It subtracts the daily rate from the account balance and begins to to validate the user through updating the fields in the database, such as updating the spot status to "close", Complete to 0, user ID of the spot to the user's ID, and Log_In_Status to 1. The dictionary's status field is set to "verified" and the current balance field is set to the user's updated account balance.

5. `fill_transaction(cursor, userid, bt_string)`
   Takes the user ID and bt_string to populate the transaction table with a payment. Uses various fields from the given user account and parking spot to log details of the payment.

6. `fill_occupation(cursor, userid, bt_string)`
   Takes the user ID and bt_string to populate the occupation table with an entry of a spot's activity. It uses the arguments to search for user data and spot information to create an activity log for the specific spot.

3.4 Mobile App

The Mobile App consists of two major functionalities: Bluetooth scanning and HTTP POST/GET requests.

3.4.1 BluetoothAdapter.LeScanCallback (Bluetooth)

The `LeScanCallback` API allows the mobile device to scan for other LE devices within the area and return the appropriate callback results. In order to configure the mobile app to only scan for beacons with iBeacon formatting, the following additions were made:

1. **Androidmanifest.xml** - To suppoort Bluetooth Low Energy scans, the following permissions were added. The first permission (`BLUETOOTH`) is required to perform any type of Bluetooth communication such as requesting a connection, accepting a connection, or transferring data. The second permission (`BLUETOOTH_ADMIN`) enables device discovery and is required for discovering other Bluetooth devices within the immediate area. This permission also enables the app to manipulate Bluetooth settings. More information on these permissions can be found in the Android Developer Documentation.

```
<manifest ... >
  <uses-permission android:name="android.permission.BLUETOOTH" />
  <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
  ...
</manifest>
```

2. **private BluetoothAdapter.LeScanCallback leScanCallback** - The purpose of the callback function is to recognize other Bluetooth devices in the area and interpret that information. The function is set up so that it only recognizes and reads data from iBeacon BLE devices. The function handles the recorded data (saved into `scanRecord`) and converts the raw byte information into hexadecimal format for better understandability. The UUID, major, and minor values are then parsed out the final hexadecimal value and are then checked into a string array for further Bluetooth differentiation.

3. **private Runnable scanRunnable** - This function invokes the above callback function on a 20 milisecond interval. This is so that the Bluetooth scanning is not continuous, and does not consume too much power when the app is in use. This function is initialized in the `onCreate()` method of the `MainActivity.java` file.

3.4.2 Volley (HTTPS POST/GET Requests)

`Volley` is an HTTP library that allows for networking between the mobile app and web servers as opposed to the the standard network connection APIs within Android. Utilizing the Volley library allows for easier and faster HTTPS POST/GET requests and the use of a request queue when multiple requests are called at once. This is necessary when multiple requests may be called within the same function. The setup and use of volley is as follows:

1. **Gradle** - Adding the Volley library requires the following dependency within the `build.gradle` file.

```
dependencies {
    ...
    compile 'com.android.volley:volley:1.1.0'
}
```

2. **public class MySingleton** - The `MySingleton` class initiates a request queue that lasts the entire lifetime of the app. With the request queue implemented using a singleton pattern the request queue does not need to be closed or stopped when requests are not being called. This is so that the request queue does not have to be reinitialized each time a request is made, expecially since the app may make a request multiple times within the same function.

4 Project Simulation

To test how the project works in different situations, we developed a simulation program to verify our design through a wide range of conditions. The goal of the simulation is to mimic an average workday and observe how our SPOT system responds to the events. We want to test a random range of users entering and leaving the spots at different times.

4.1 Simulating Gateway

We spawn multiple Gateways with the a Python script using the `pexpect` library. From here, processes are spawned with the `pexpect.spawn(<programName>)` command. Our gateway program begins with the following paramters:

```
./server <Port>
```

where `<port>` is the port the gateway is using to talk to sensors on.

4.2 Simulating Sensors

Using a sensor program written to test the Gateway, we use a Python script to spawn multiple sensor programs. The sensor program, called `client.c` takes in the arguments:

```
./client <GatewayPort> <ChangedBit ParkingLot:PrkingSpace> <Spot Status>
```

Starting a sensor program with the spot status field as `busy` will simulate a user entering a spot.

Starting a sensor program with `open` for the spot status indicates that there is no user in the spot

4.3 Mobile App Simulation

To simulate the Mobile App, we use a Mobile App test program that is run with he following paramters:

```
python app.py <USER ID> <Blue Tooth String>
```

The app program will simulate the Mobile App by sending a HTTP POST message and waiting for verification of the spot.

5 Conclusion

While the entire project is functional, a few complications hindered our along the way. Getting over the learning curve for Google Cloud took a good portion of our time. Reading in bluetooth IDs from an Android app proved to be much more difficult than anticipated and set us back a few weeks before overcoming the problem and understanding how to use the bluetooth APIs. Lastly, figuring out how to use Wi-Fi on the microcontroller, and connecting to our gateway was challenging since the documentation on Wi-Fi left a lot to be desired. However, the team was able to accomplish their tasks in their respective roles that where crucial to the success of the SPOT project.

If given more time, improvements could be made to add additional functionality and reliability to the SPOT system. First, adding solar panels to our sensor units would effectively make the sensors self sustaining when combined with rechargeable batteries. We would also like to add a camera for license plate recognition for users who do no have our app or a mobile device and include a kiosk for user payment. Lastly, while the Gateway is robust and handles sensors breaking and threading, the whole process could be simplified and streamlined to improve system performance.