UNIVERSITY OF CALIFORNIA, SANTA CRUZ

# Campus Parking Infrastructure: Team 3

Scott Birss, David Caplin, Troy Fine,
Chenxing Ji (Gabriel), Aditya Sriwasth, Nelson Yeap

June 9, 2018

# Contents

# 1 Project Motivation

The emergence of the concept of "Internet of Things" has revolutionized our understanding of networking and has paved the way for several new applications and methods of infrastructure in the 21st century. By creating a network connecting physical devices, vehicles, home appliances and other items embedded with electronics, the exchange of data between these devices and the collection of data on a virtual cloud has enabled us to use this information to analyze new data and make it an easy platform for the users to utilize and manage. For instance, a modern household can have its security devices, light switches and other utilities around the house interconnected utilizing a mesh network or a cloud and a user could use an application on their phone to control all of these devices. Our group gained the inspiration to bring about the same concept on parking lot infrastructures that exist today.

The goal of this project is to improve the existing parking lot systems in place by using the concept of IOT (Internet of things). By utilizing several hardware and software elements and tools, our group seeks to design and implement an affordable, scalable and robust solution to bring about a revolutionary new concept to park and manage parking in these giant infrastructures.

# 2 Executive Summary

The Campus Parking infrastructure project seeks to improve the existing models of parking lot management and introduce elements that can improve the user experience by utilizing concepts from IOT (Internet of things).

A parking lot infrastructure consists of three major elements: the client that is parking his/her vehicle, an administrator that manages and monitors a parking lot, and the parking lot itself. Our design seeks to add elements to each part of this process and ultimately improve the user capabilities for both the client and administrator. By utilizing a sensor, we can monitor whether a parking spot is vacant or occupied. By designing a mobile app, we can let a user look at how many spots are open in a particular area of the parking lot. By designing a website application, we can let the administrators monitor the same data and give them the privilege to reserve certain spots or carry out other tasks. Finally, by setting up a backend server on a cloud, we can process and handle all of the data coming from all these components and establish communication protocols to make all of these devices work with each other. By designing all of these individual elements and setting up a method for them to communicate with each other,our group will be able to deploy the working version of this infrastructure.

# 3 Project Description

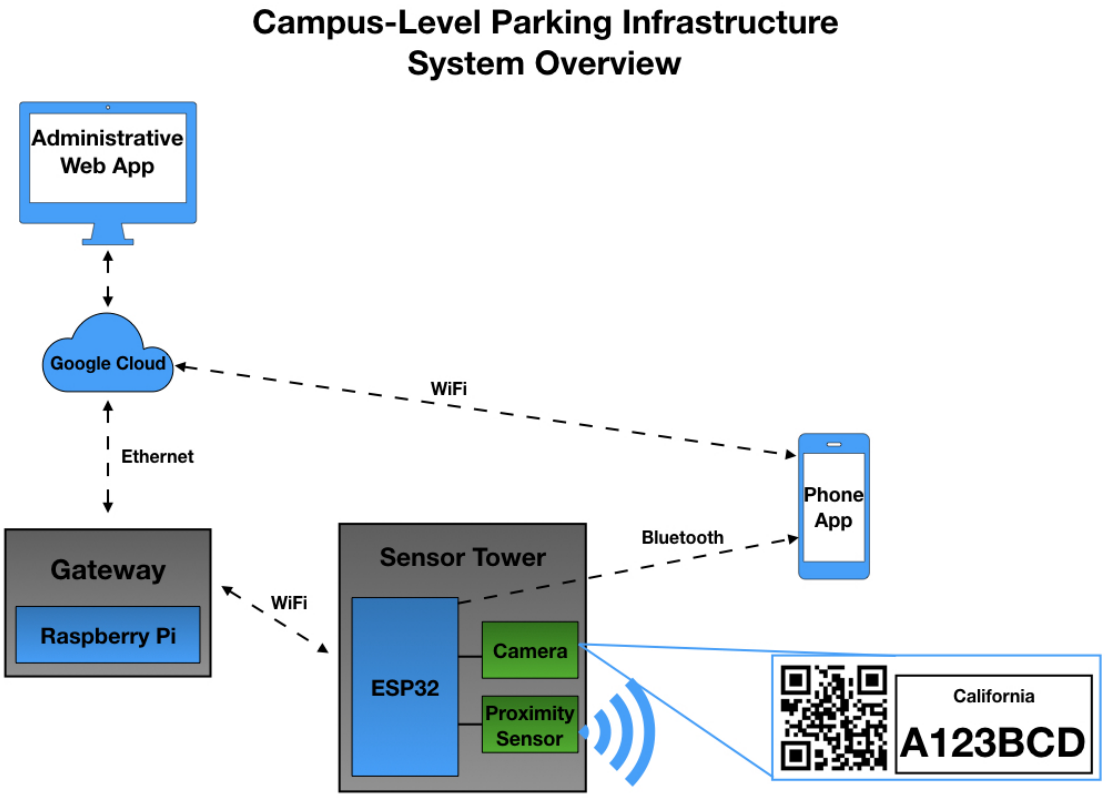



Figure 3.1: Top Level Block Diagram

In order to describe our project with a high level of abstraction, the project can be broken into several subcategories that form the pieces of the puzzle. Here are the several parts that will make up our design:

- Parking Lot Sensor: In order to determine if a spot is vacant or occupied, our group will use a sensor placed at the spot to detect whether an object is currently positioned at that particular spot. In order to scale this project, every spot in a parking lot will have a sensor placed to perform the same operation.
- Camera: A small camera is placed on the parking lot to take an image of the car's license plate in order to send that number to the cloud for further processing.
- Microcontroller: The microcontroller will be the basis for processing and sending data to the cloud from the sensor and camera in the parking lot. Once a user pulls over a vacant parking spot, the sensor will send a message to the microcontroller indicating that the spot is taken. This message is send to the cloud from the microcontroller since it has wifi and cellular capabilities. The microcontroller also has bluetooth, and

therefore the user can use the bluetooth functionality on their phone to establish a connection with the microcontroller to authenticate themselves and signal that they intend to park at that spot.

- Cloud Database: The cloud acts as the major storage center for data and is the central node of access for all of the other elements in this project. The cloud collects data from each piece in the project and also sends back the appropriate data when requested from these devices.
- Mobile Application: The mobile application acts as the user interface for a client who wants to check for parking spots in an area or authenticate themselves when parking in a spot. The bluetooth beacon on the microcontroller at a parking spot will send a unique URL to the phone to confirm which client is parking at a given spot. The phone directly communicates with the cloud in all other instances. If the user is currently enrolled at UCSC, they can simply enter their Student ID to check if they are allowed to park at the spot, and proceed to leave the car once they get a notification on the phone indicating they are good to go.
- Website Application: The parking administration would also have a website application to manage parking spots, set up reservations for special events, or perform other tasks remotely, thus making it an easier experience for not just the user but also the administrators of a parking infrastructure.

# 4 Sensor Tower Prototype Design

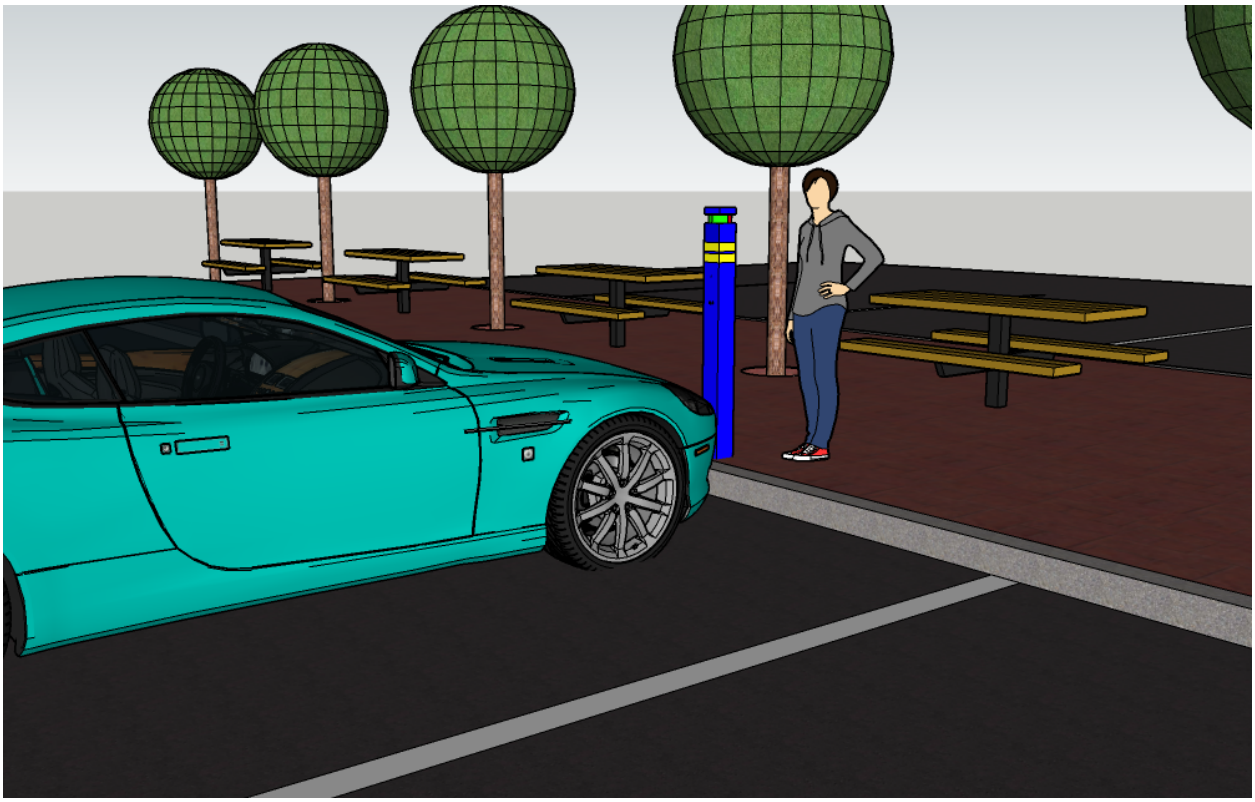


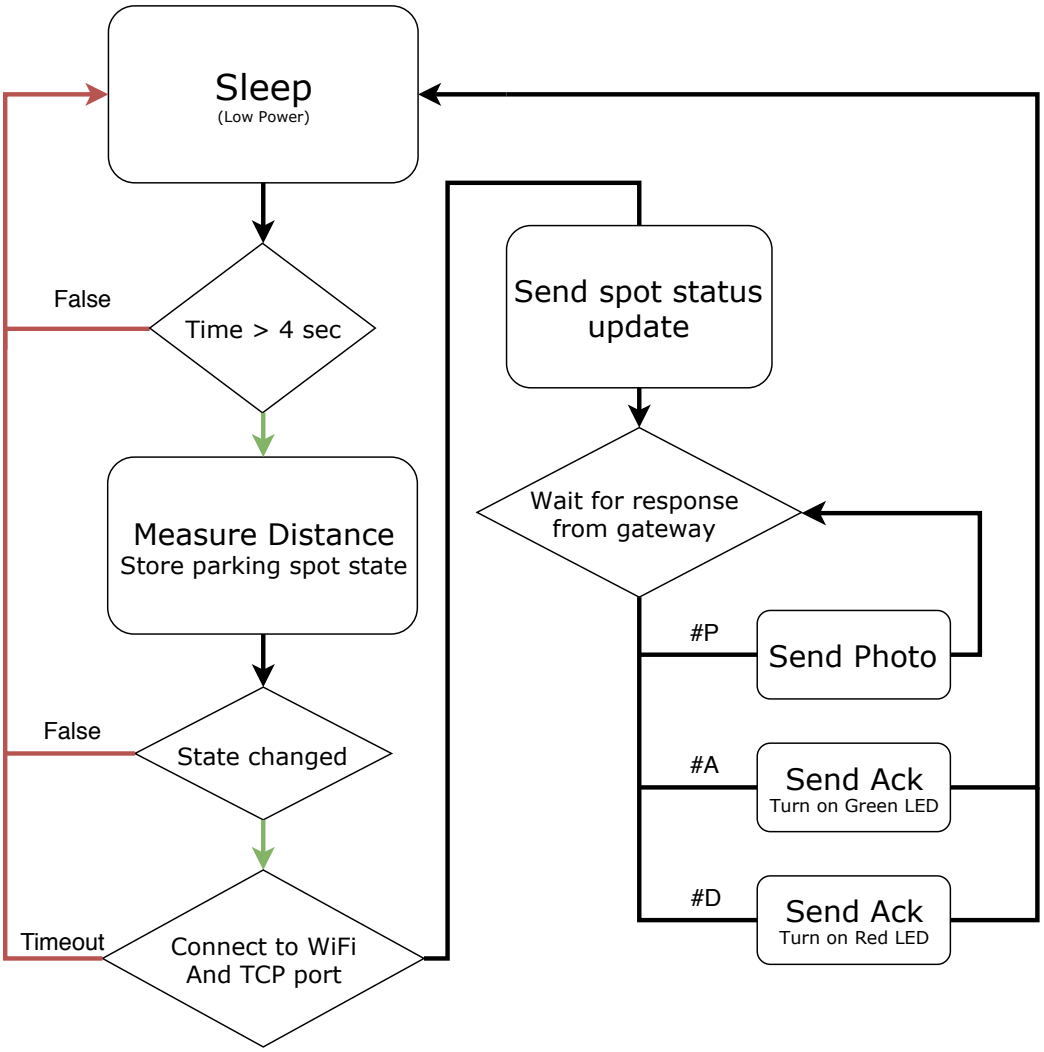
Figure 4.1: Parking Space Sensor Tower Prototype Design

# 5 Design

The following section will describe in detail the several components that will make up the design for the project and will describe the proposed approach and design for each component in the project.

## 5.1 Sensor Box

The Sensor Box will be placed at each parking spot. It is responsible for sensing when a car has either parked or has left the parking spot. The Sensor Box is controlled by a ESP32 which has built in WiFi and Bluetooth. The ESP32 can detect if a car is in the spot via SR04 distance sensor. Communication between the Sensor Box and the gateway is handle via TCP over WiFi. To save power, the ESP32 goes to sleep for 2 seconds after checking the status of the spot or finishing gateway communication. In addition, it has an OV7670 camera for sending picture to the gateway to process QR codes.




(a) ESP32

Figure 5.1: State Diagrams

Figure 5.1 covesr how the sleep function, car detection and gateway messaging works.

## 5.2 Car Detection

The Sensor Box uses a SR04 distance sensor to detect if the parking space is occupied. In addition, the Sensor Box is equipped with a camera to identify the user via QR code and licence plate. To reduce noise, each distance measurement is taken twice and an average is taken. Distance measurements under 2 meters are considered signify that a car is parked. In addition, the gateway can request additional picture to be taken in the case of a sub-optimal photo.

### 5.2.1 Camera

We are using a FIFO based camera (ArduCam with OV2640) that can take JPEG images up to 2 megapixels. There's two reasons why selected a FIFO based camera. First the ESP32 low RAM size can only support images resolutions up to 160x120, which is insufficient for our purposes. Second, it can take images in JPEG while most non-FIFO cameras take images in BMP, which has a larger file size. We used the libraries and sample code that can be found here.

### 5.2.2 TCP Messages

The Sensor Box will communicate with the gateway via TCP. Only the Sensor Box can start communication with Gateway and the Gateway sends responses to each message. The Sensor Box will notify when the status of the spot has changed such as a car pulling into the spot. In addition, every 10 wake up cycles, the Sensor Box will ask the Gateway if there any task it need to service such as send a picture of the car. Here is an example of the message's format:

```
SPOT_ID,MESSAGE_ID,SPOT_STATUS
```

Each message starts with the Sensor Box's ID followed by the message type and lastly the current status of the spot. Here is more message examples with a spot ID of 2:

```
2,StatusChanged,taken
2,StatusChanged,empty
2,AnyTasks,empty
```

For every message the Sensor Box send it requires an acknowledgement. This is to be certain the cloud and gateway correctly received the message. In addition, the gateway may tell the Sensor Box to do a task such as take a picture. Acknowledgement messages always start with a  followed by a character. For example if the Sensor Box notifies the Gateway that the spot status has changed to taken, it can response with A to tell the Sensor Box that the car is authorized to park or D to deny the parking space. The Sensor Box will then change the LED to either green or red to notify the user if they are authorized to park.

Table 5.1: Sensor Box to Gateway Messages

| Spot ID | Message ID | Spot Status |
|---|---|---|
| 2 | StatusChanged | taken |
| 2 | StatusChanged | empty |
| 2 | AnyTasks | taken |
| 2 | AnyTasks | empty |
| 2 | Picture | taken |
| 2 | Picture | empty |

Table 5.2: Gateway to Sensor Box Messages

| Ack Type | Description |
|---|---|
| A | Authorized to park. |
| D | Deny space. |
| P | Take picture. |
| N | No tasks. |

### 5.2.3 Pin configuration

The ESP32 is connect to the camera, SR04 and LEDs in the following way:

Table 5.3: SR04

| SR04 | ESP32s |
|---|---|
| VDD | Vin |
| ECHO | 27 |
| TRIG | 26 |

### 5.2.4 Bluetooth Beacon

The Sensor Box is equipped with a Bluetooth beacon which assist the mobile app in authenticating users. It is based on Google's Eddystone URL beacon. When the user parks into a spot the Bluetooth beacon will turn on and transmit the beacon URL with the following format:

```
#AF:14:A9:01:23:45
```

The URL can then be used by the app to identify what spot the user is trying to park at. See section 5.7.3 for more details on how the app uses the URL.

Table 5.4: Camera with FIFO

| ArduCAM | ESP32s | GPIO | DevKit |
|---|---|---|---|
| VDD | 3.3v | 3V3 | I19 |
| CS | TX2 | 17 | B9 |
| MOSI | D23 | 23 | B18 |
| MISO | D19 | 19 | B12 |
| SCK | D18 | 18 | B11 |
| SDA | D21 | 21 | B14 |
| SCL | D22 | 22 | B17 |

Table 5.5: RGB LED Strip

| LEDs | ESP32s |
|---|---|
| Digital Pin | 14 |

## 5.3 Gateway Hardware

The gateway (Raspberry Pi 3) is setup as a WiFi hotspot using RaspAP. This allows the Sensor Box to easily connect to the gateway via WiFi. In addition, it allows full control over IP address allocation. The gateway will always have the IP address 10.3.141.1 while the Sensor Box can have a range of IP from 10.3.141.50 to 10.3.141.255. The gateway's SSID is pi_gateway followed by a number such as pi_gateway_1. The password is a hash of the SSID to provide a small amount of security. In addition, only the gateway itself can connect to the Internet. A Sensor Box connected to the gateway can not access the internet directly.

## 5.4 Gateway

The gateway which runs on a Raspberry Pi 3, serves as a intermediary step between multiple sensor towers and the cloud, multiplexing data to and from each sensor tower and the google cloud datastore (the database). The purpose of this step is to increase scalability of the parking infrastructure by buffering data and messages before sending them off to and from the database. For smaller parking lots, a single gateway may prove to be sufficient, but any larger and several gateways will be important to increase WiFi coverage of the parking lot.

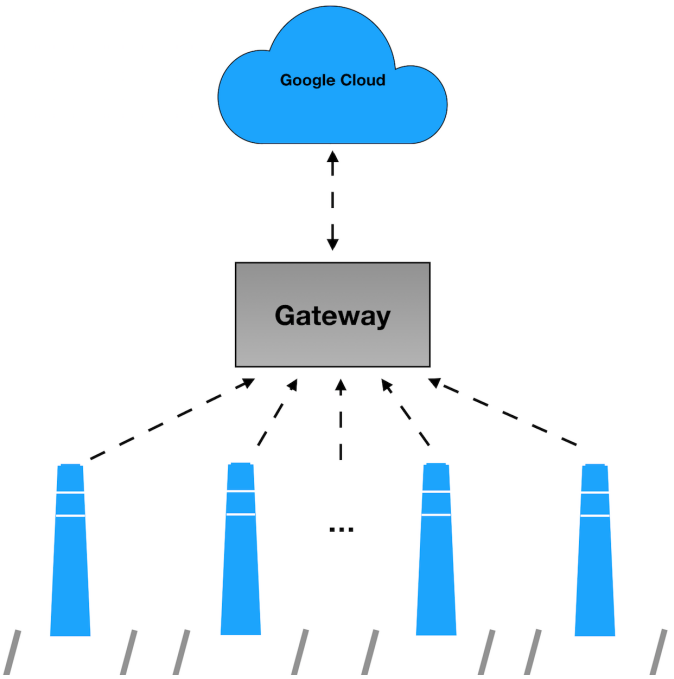



Figure 5.2: Gateway Scaling Parking Infrastructure System

Additionally, the gateway also receives and converts photos of license plates and QR codes into strings with the help of Google cloud and computer vision (more discussed in the Computer Vision section below). It also logs the photo onto the Google clouds data storage platform, the blob. Using API's from Google cloud, it is able to communicate to the database and log events, receiving responses, and send status.

### 5.4.1 High Level Overview

The gateway can broken down by it's core functionalities. Firstly, it hosts a WiFi LAN hotspot for the sensor boxes to connect and disconnect from (discussed in Gatway Hardware section above). Secondly it acts as the intermediary communicator between its LAN of sensor boxes of sensor boxes and the Google cloud datastore, providing scalability. Thirdly, it is responsible for taking the sensor box's photo of the car with the QR Code and License plate in the windshield and translating that into strings which are forwarded to the cloud.

The basis for the interconnection between the each of the gateway and surrounding sensor boxes is a server/client architecture using a TCP socket protocol, where the clients are the sensor boxes and the server is the gateway. Additionally, the gateway also functions as a client in the gateway to database relationship.

In the figure below, the basic flow is outlined into two primary components. It explains the main sections that make up the gateway and its ability to handle multiple sensor boxes and there various tasks.

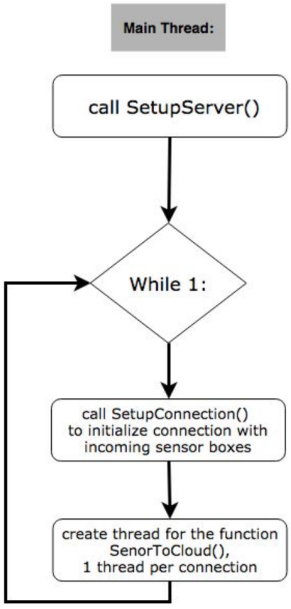



Figure 5.3: Main Thread Flowchart

### 5.4.2 Initialization and Main Thread Spawning Flow

The primary code begins with some constants and variables such as the maximum amount of clients the gateway intends to listen to at any given moment and several semaphores for protecting multi-threaded code in critical sections (more on this later). Then the SetupServer( ) function is called, declaring a socket of type IPv4, TCP, which then binds to its existing static IP address and defined static port number, listening for the defined amount of connections.

After the declaration of some constants and the server setup, the gateway then begins to wait for new connections, spawning a new thread for each incoming connection. The primary purpose of multithreading in the gateway is to create a system that allows for each connection to invoke blocking calls such as send() and recv() without blocking another sensor tower from running. Calling SensorToCloud() for each incoming connection as a new thread does just that. Once the gateway creates the thread, it waits for more incoming connections and spawns new threads for those while processing its current connections.

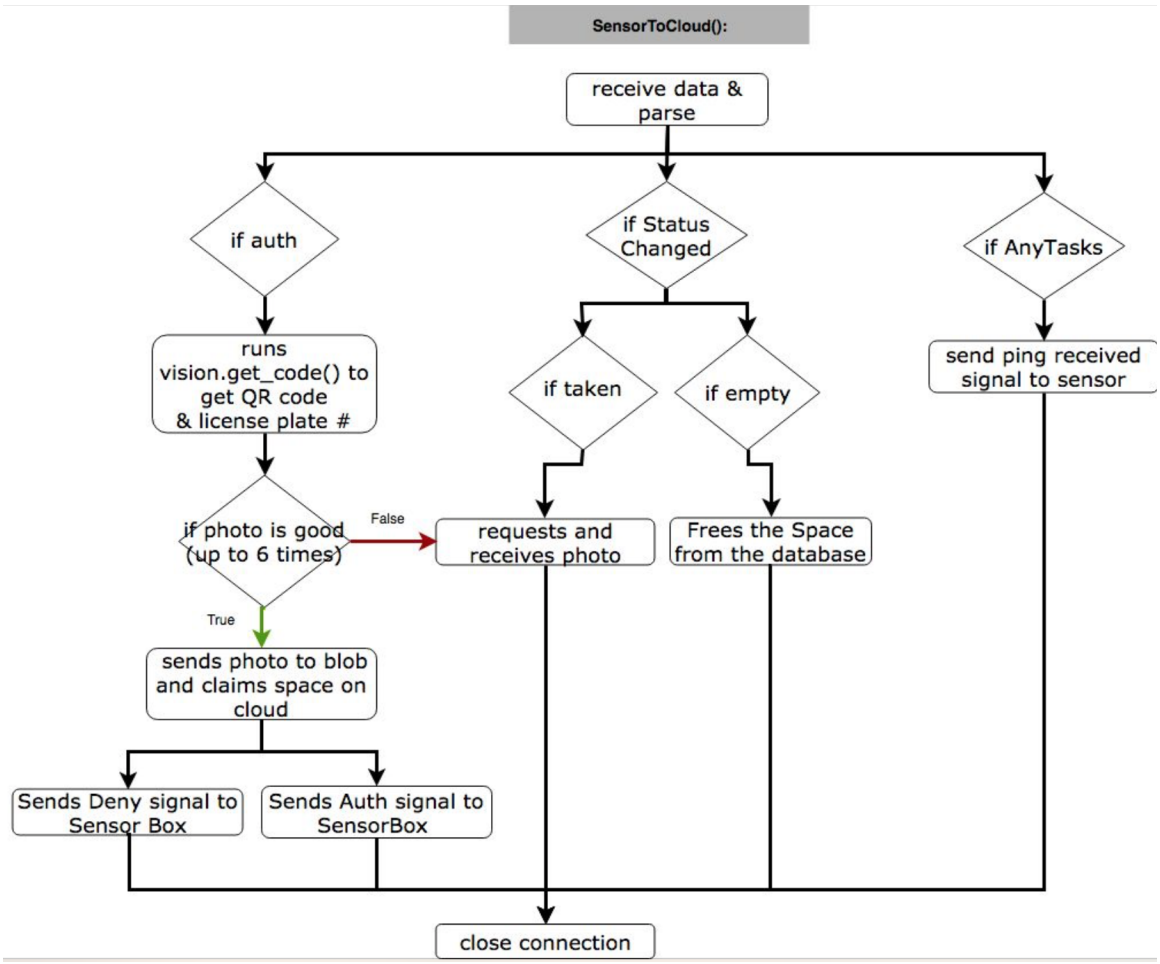



Figure 5.4: Main Thread Flowchart

### 5.4.3 Sensor tower to Cloud Communication

Above we have a flow chart to help explain the various states of the gateway. Once a new thread SensorToCloud() is spawned, the function receives data from the sensor box and parses it. If the sensor box was triggered by a car pulling into the spot, then the sensor would deliver a message like this: "2,StatusChanged,taken" Where 2 is the spot ID, StatusChanged indicates a status change, and taken indicates the spot is now taken. It then requests a picture from the sensor box. Soon after the gateway begins to receive the photo, and the sensor box initializes the closing of the connection to indicate end of file (EOF). It then opens a connection and receives an "Auth" message, indicating that it has re established a connection and can go ahead and decode the picture for a QR code and license plate number (discussed in more detail in the computer vision section). If the photo is bad, as in the license plate was not recognized or something went wrong, then it requests a new photo from the sensor box. This can happen up to six times before it gives up and determines the user as unauthenticated. Once a good photo is received or the 6 cycles are up, it waits for the database to be unoccupied by the gateway by using a semaphore. The gateway then

sends it's data to the database. If it matches whats in the database then the database will return a authorized signal which will then traverse to the corresponding sensor box and it will light up green to let the user know they are good to go. Otherwise, if it didn't match, it will return a denial signal, in which case the user either not not supposed to be parked there, or there was an error in the plate/QR code detection. In the latter case, the user can use the Bluetooth beacon emitting from the sensor box in order to authenticate themselves on there smart phones using the mobile app. (Discussed in more detail in the mobile app)
In addition, every 10 wake up cycles, the Sensor Box will ping the gateway with the message "AnyTasks" to ensure that it is still active.
Lastly, when a car leaves the parking spot, it would send a "2,StatusChanged,empty" to indicate that spot 2 has changed to status empty. This message then waits in the buffer (just as the "taken" message had) to access the database and then updates the corresponding field in the database.

## 5.5 Cloud

### 5.5.1 Cloud Summary

The Cloud will serve as the central node for sending and receiving data to all of the components and establish an important communication link between all of the devices. The Cloud will conceptually have a front-end and back-end section, with the front-end designated for communicating with the user interface of the website and mobile application while the back-end designed to collect and process data coming from the Gateway(including the image captured from the camera) and the two applications. For this project, we are using the Google Cloud Datastore and the App Engine platforms. Datastore will be used to maintain the databases in figures 6.4, 6.5, 6.6, and 6.7 which are accessed by The Gateway, the Administrators Web Application and the Mobile Application. A python program will be deployed using App Engine which will be accessed by the user mobile application and administrator web application. As an extra safety precaution, the Mobile Mobile User Database in Figure 6.7 can not be accessed by the Gateway since it has users passwords. Thus, it is only used to authenticate the users via mobile app and the administrators.

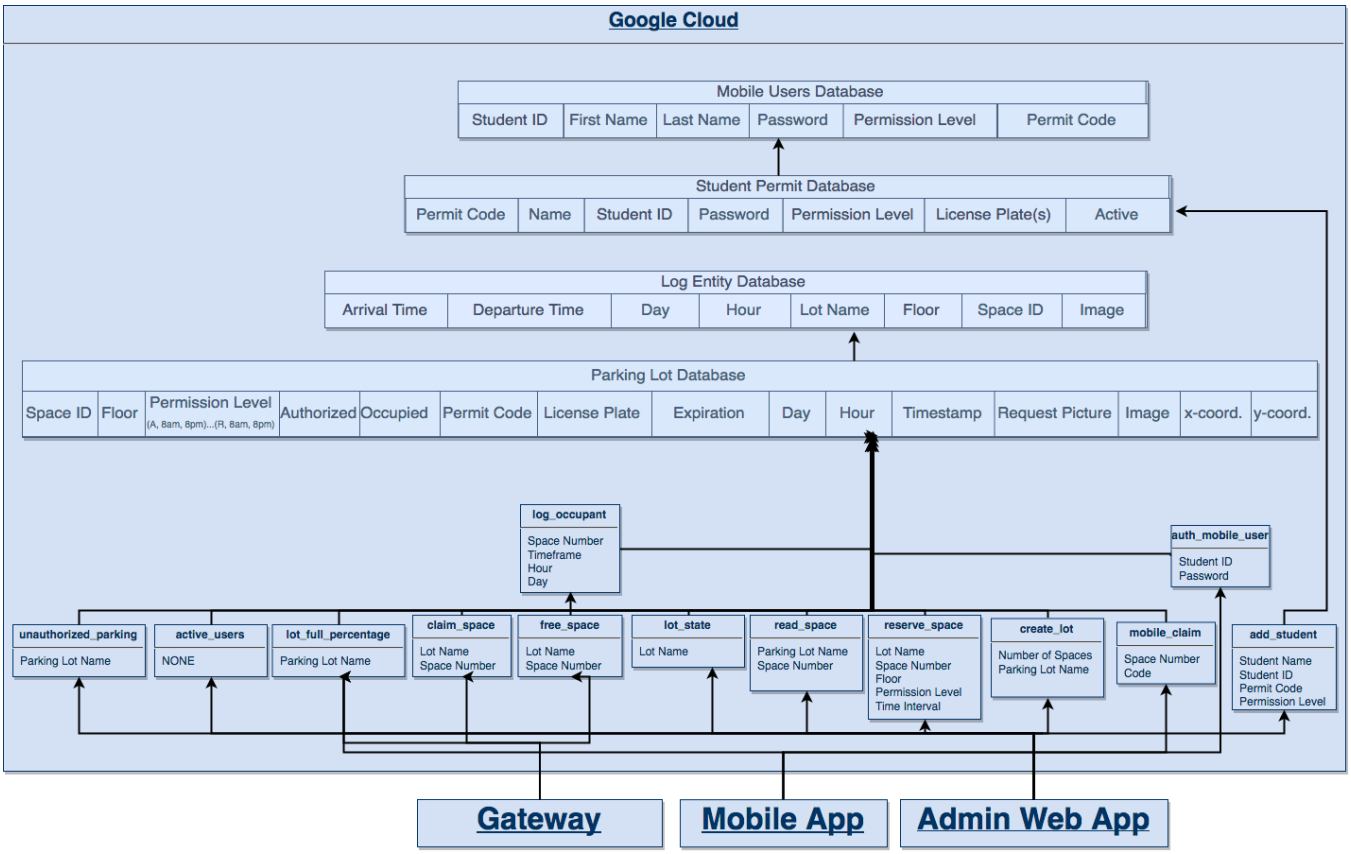



Figure 5.5: Cloud Data/Flow Block Diagram

| Parking Lot Database | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Space ID | Floor | Permission Level (A, 8am, 8pm)...(R, 8am, 8pm) | Authorized | Occupied | Permit Code | License Plate | Expiration | Day | Hour | Timestamp | Request Picture | Image | x-coord. | y-coord. |

Figure 5.6: Parking Lot Database

| Student Permit Database | | | | | | |
|---|---|---|---|---|---|---|
| Permit Code | Name | Student ID | Password | Permission Level | License Plate(s) | Active |

Figure 5.7: Student Permit Database

| Log Entity Database | | | | | | | |
|---|---|---|---|---|---|---|---|
| Arrival Time | Departure Time | Day | Hour | Lot Name | Floor | Space ID | Image |

Figure 5.8: Log Entity Database

| Mobile Users Database | | | | | |
|---|---|---|---|---|---|
| Student ID | First Name | Last Name | Password | Permission Level | Permit Code |

Figure 5.9: Mobile App Users Database

### 5.5.2 Gateway to Cloud APIs

The Gateways gain access to a database corresponding to the parking lot they are in by generating a key that corresponds to the lot by and a specific space within the lot by executing the following commands:

```
ID = 'Space-'.format(space)
  with datastore_client.transaction():
    key = datastore_client.key(lot_entity_name, ID)
    space_ = datastore_client.get(key)
```

When a car has pulled into a parking space, the sensor box will send an image of the car and its space ID to the Gateway. The gateway will then use computer vision to decipher the permit code and forward it along with the corresponding space ID to the Google Cloud Datastore to be added into the structure database which will also store the original image taken at the space for storing if the administrator wishes to view it later. Since this function will be executed by the Gateway, it needs to be able to work with however many spaces the gateway communicates with.

- claim_space : space_num | code

After the key is pointing to particular space, by being given the ID from the sensor box at the space, you can then manipulate the parameters of the database. In this case, the "claim_space( )" function will set the "Occupied" field to "true", "Code" to the permit code that is deciphered, "Authorized" to true or false depending if the permit code has the level of permission equal or greater to that of the space, and "Timeframe" to the time that the space was initially taken. As an additional measure to guard against fraudulent permit codes, this function also checks if the permit code is currently parked in a lot on campus by checking a field in the Student Permit Database when it checks the students parking permissions. If the permit code is currently in use in a parking space, the function will return a value, '3', indicating so to the Gateway which will cause the LED to turn red, signifying that the student is not authorized. Those changes are then "put" to Google Cloud Datastore, as follows:

```
space_['Code'] = code                    string
space_['Occupied'] = True                boolean
space_['Authorized'] = True              boolean
space_['Timeframe'] = datetime.now()     date and time
datastore_client.put(space_)
```

When a car has left a parking space, the sensor box will send its space number to the Gateway, which will then forward all the spaces that have become open to the structure database to be updated.

- free_space : space_num

This function is similar to claim space, however, it basically resets everything changed by claim space. This can be accomplished by executing the following commands:

```
space_['Code'] = 0                          string
space_['Occupied'] = False                  boolean
space_['Authorized'] = False                boolean
space_['Timeframe'] = datetime.now()        date and time
datastore_client.put(space_)
```

### 5.5.3 Administrator Web App to Cloud APIs

The Administrator can create a parking lot simply by giving it a name, number of spaces, and the desired permission level of the parking lot.

- create_lot : space_num | lot_name | permission_level

The Administrator has the ability to select any space to reserve if, for instance, there were a guest speaker coming to a conference. Given a lot name, space number, permission level, and span of time that the space will be reserved for. After the time period expires, the space will then return to its default attributes.

- reserve_space : lot_name | space_num | permission_level

Administrator wants to add a student to the database of students along with their corresponding parking permission level

- add_student : code | first name | last name | student ID | permission level | license plate | expiration date

Every time the dashboard of our website is loaded, a percentage indicating the amount of cars are currently in each parking lot will be displayed. This is achieved using the following API

- lot_full_percentage : lot_name

At any time the Administrator can see the status of any space in any lot. This information returned by this function includes the name, permission level, permit code, and the license plate number of the car currently parked in the space.

- read_space : lot_name | space_num

This function will be executed if the administrator wishes to see any changing trends in parking over a specified timeframe. The administrator will be able to view parking statistics at the hour scale over the quarter or academic year.

- get_parking_stats : lot_name | start_date | end_date

Another capability of the administrator is to view all of the cars that are illegally parked in any of the campus parking lots. This can be achieved theu the following API:

- unauthorized_parking: lot_name

### 5.5.4 Mobile App to Cloud APIs

The Client using the mobile application has limited access from the cloud to perform tasks like view the status of a parking lot and can send information to check against what is registered in the cloud. Though we have a camera on the front of our Sensor Tower that is designed to decode our parking permit codes, it is not always 100% accurate. Thus, the mobile application acts as a second form of authentication. When the user pulls into a parking space, if the code is not recognized, the user will receive a bluetooth beacon from the sensor tower indicating which space they are in and at that point the app will authenticate them and their permit will be placed in the parking lot database as occupying the space they are currently in. Here are the API's that will be used for this interface:

User is a guest that wants to register themselves on the cloud:

- add_guest : name | code

This API will be called so that when they are logging in for the first time the user will have to enter their student ID and password in order to receive their permit code to use for spot authentication:

- student_login : student_ID | password

Every time the mobile app visits the page where they can select a parking lot, a percentage indicating the amount of cars are currently in each parking lot will be displayed. This is achieved using the following API:

- lot_full_percentage : lot_name

When a student buys a new permit, they are given a temporary password that is an encrypted string based on their first and last name, so the student should reset their password when they use the mobile application for the first time, which can be done using the following API:

- reset_password : old password | new password

## 5.6 Computer Vision

Integrated into the Gateway, the Computer Vision part of this project aims to convert the input image from a connected camera into a character string which is the stored QR code in the Google Cloud Datastore. Additionally, the computer vision module also leverage Google Cloud Vision service for License Plate recognition along with the QR code.

### 5.6.1 Computer Vision Goal

The main goal of computer vision system is to provide automation in this Parking System. In order to do so, QR code detection are used for the permit code of the user, where the gateway can decode the QR code to get the permit code of that parked in user without human intervention.
However, the QR code does have a security flaw that anybody can take an image of other people's code and use that. To insure authentication of the module, License Plate Recognition are introduced. Additionally, it is assumed that the each vehicle's License Plate are unique and if anybody is impersonating another vehicle would be detected by the Police on the road.

### 5.6.2 Computer Vision detailed description

This module now consists of three different sub-modules and three available APIs.
The first part is a QR code generator that would takes in the input and generate the QR code based on the input. The second part of the Computer Vision module is a QR code decoder for the image taken by the esp32.The third sub-module is the License Plate Recognition system for integrity goal of the QR code
The QR code generator would run on the administrator app that allows the admin to generate the QR code based on the database and gives the permit to the students. This would allow students to have automated parking ability when they comes into any on-campus parking lots deployed with our system.
The QR code decoder, on the other hand, would run on the Gateway for authentication such that user can claim the usage of the spot in the Google Cloud Datastore. The decoder would read in an image and scan to get the permit id of the user. If there is no QR code detected, the sub-module would return "ERROR" indicating either the image quality is low or the user doesn't have a qr code

### 5.6.3 System Design

The Computer Vision part two consist of two python programs:
The first part of the program is a QR code generator embedded in the Administrative App allowing admins to generate permits for the users to tag on their cars so that the QR decoder has something to work with. A function `generate_qrcode(input_string, output_filename)` is used in this module so that the input string would be contained in the generated QR code

file.
The second part of this Computer Vision module is the QR code decoder embedded in the Gateway to process the images came from the Sensor box. A function `get_code(image_filename)` is used to decode. If decoded successfully, the function would return the code representation as a string in order for the authentication on the Google Cloud side.
The third sub-module leverages Google Cloud Vision service. It uploads the Image to the Google Cloud Vision and they would return all the text with in the image. If the license plate of the user is detected, it then would return the decoded user permit code. Otherwise, it would return message indicating the license plate does not match with the user that such that the QR code is used by a third party.
This Computer Vision part would also handle uploading images to the Google Cloud and then use these images for the Administrative app so that the admins can view the pictures taken later.

### 5.6.4 Flow Chart

The following graph is a flow chart for a QR generator module which would be embedded in a page in the administrative app to allow admins print out permits for users.

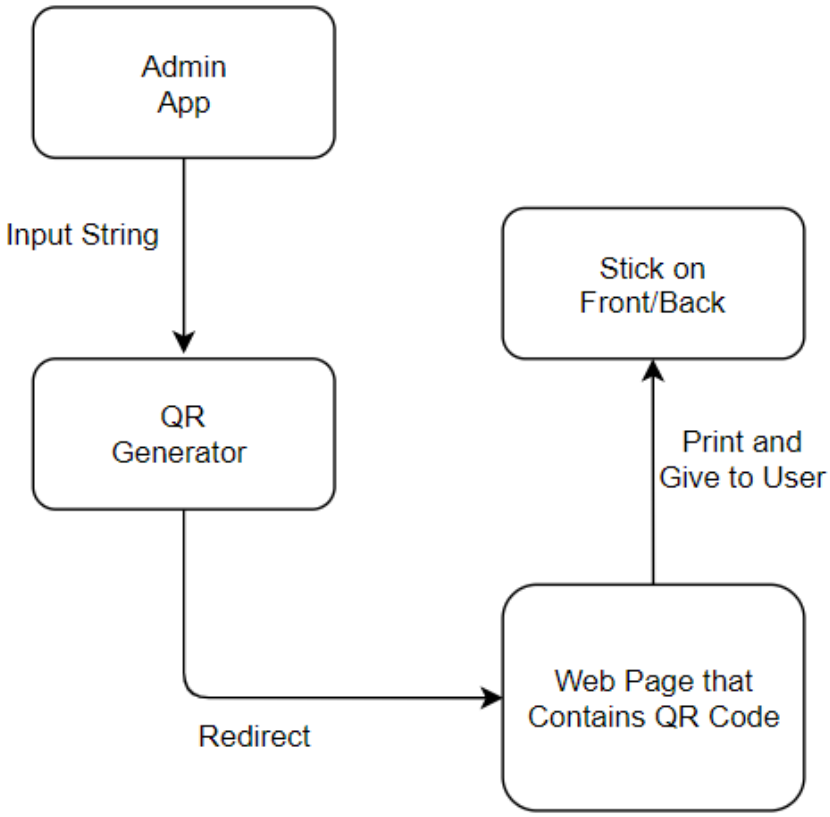



Figure 5.10: QR generator Flowchart

The following is the flowchart for QR decoder module which would be embedded in the Gateway for automatic authentication for the users.

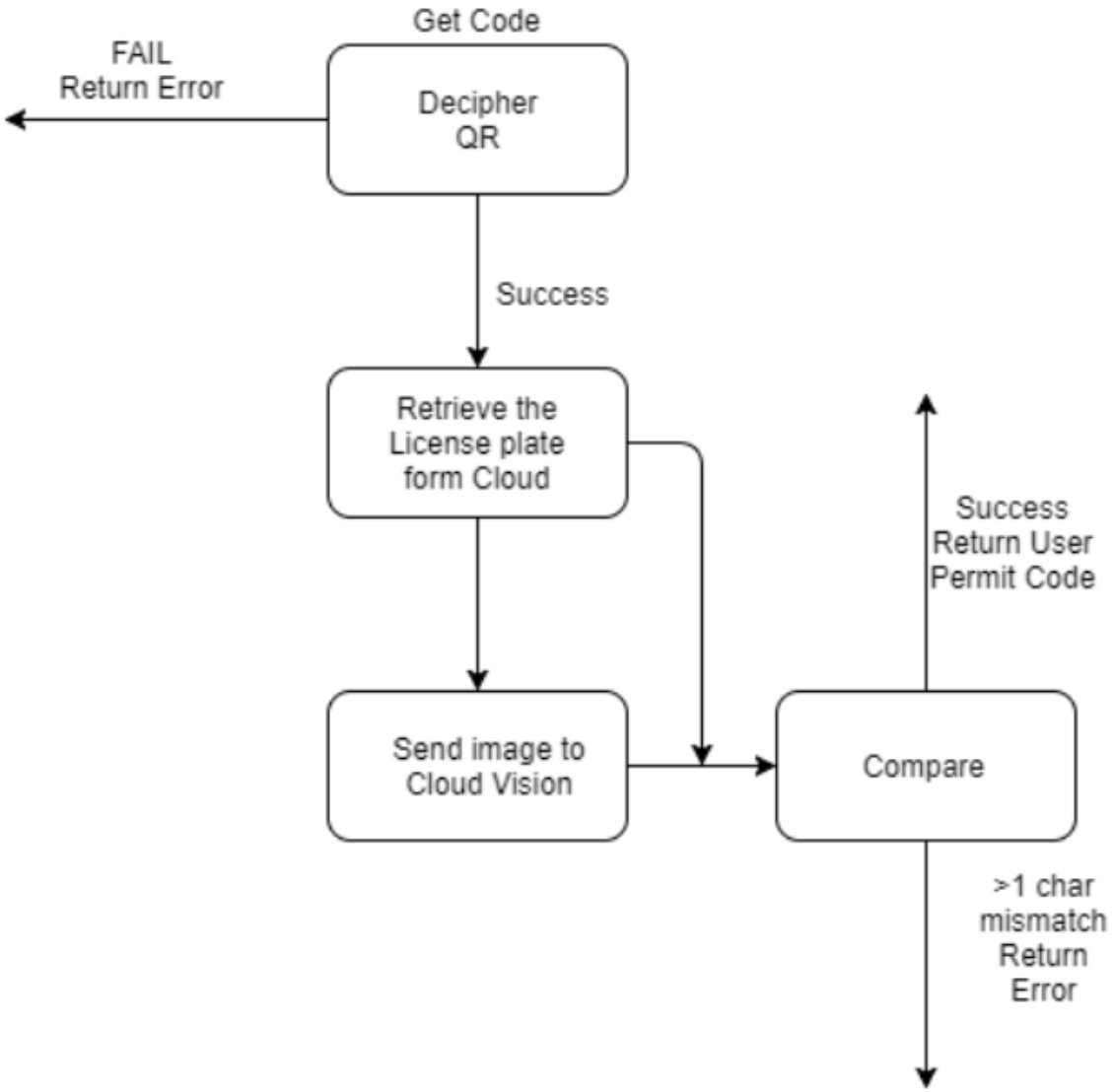



Figure 5.11: QR decoder Flowchart

### 5.6.5 Dependent libraries

Python: The majority of the Computer Vision part would be written in the Python programming language.
Goolge Cloud Vision: This is the service provided by the Google Cloud for image recognition, Vision does text recognition for you.
libpng:This is the png reference library that have rich apis to manipulate png files.
pypng: The libpng wrapper in python which allows the program to load in png image files.
PIL: stands for python image library to deal with images in python.
zbarlight:a python wrapper for zbar library, a library created for identification of a QR code inside an image.

### 5.6.6 APIs

`get_code(image_filename)`
This function get called by the Gateway when received an image from the esp32. The function takes in the name of the image file and then scanns for the QR code inside the image then return the string representation of the QR code.
The function returns "Cannot find image file" when the image cannot be loaded correctly. When the QR code cannot be deciphered from the image, it returns "ERROR". Then retrieves the stored License Plate Number based on the permit code and upload the image to Google Cloud Vision. When receiving all the text strings, compare each string to the License Plate. If there is one or less mismatch, the check would pass. Otherwise, return ERROR because QR code and License Plate doesn't match
`generate_qrcode(input_string,output_filename)`
The generate_qrcode function would be called in the admin app when the admins types in the code and hit the generate button. Then the saved image can be retrieved from the app and displayed for the admin in order for them to print it out and give to the users.

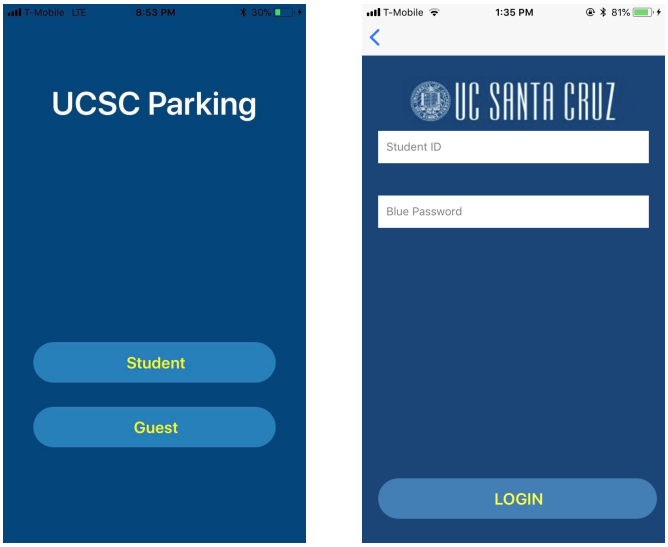
## 5.7 Mobile Application

The mobile application will be used by the client who wants to park at a spot. The application will give the user information about the occupancy of a particular parking lot and allow them to claim a spot with the Bluetooth beacon to authenticate themselves. Our implementation will include a database of registered students stored in the cloud, and only these members can authenticate themselves when parking at a spot. If the user is a guest, they can still view information about the parking lots but will not have the access to authenticate themselves.

### 5.7.1 User Interface Design

To start off building the app, there needs to be a clear vision of how the user interface would look like. The following pictures will show the design for each screen in the app and describe what functionality they have.

**Screens 1 and 2: Welcome Page and Student Login**



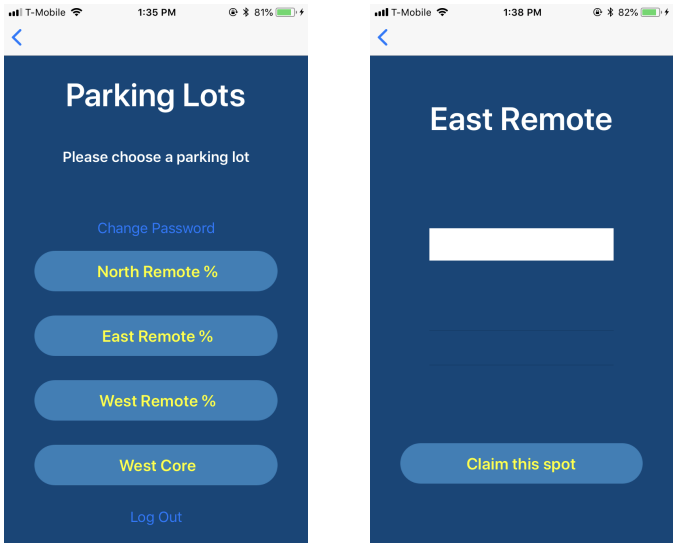

(a) Welcome Page (b) Student Login Screen

Figure 5.12: Screens 1 and 2

The Welcome Page will be the first screen that the user will see when they open the app. It starts off by first displaying a splash screen for 2 seconds, followed with a welcome page. The welcome page asks the user to state whether they are a currently enrolled student at UCSC or if they are a guest trying to find parking on campus. The user will press the corresponding button on the screen, labelled as "student" and "guest" respectively.

The Student Login page will open only if the user presses the Student button on the welcome page. Here, the student will enter their Student ID and their Cruz blue ID password in order to log in. If the user ends up entering wrong credentials, an alert screen will appear telling them to try again. If the user successfully logs in, the app will redirect them to the parking lots screen, discussed in the following section.

**Screens 3 and 4: Grouped Parking Lots Page and Bluetooth authentication**



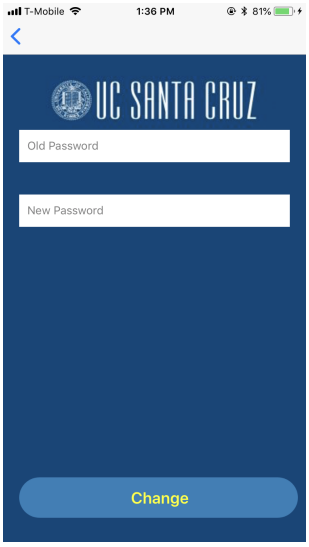

(a) Parking Lots (b) Bluetooth Beacons

Figure 5.13: Screens 3 and 4

The Grouped Parking lots page will showcase the 4 major parking lots on campus (North,West,East Remote and Core West) and display to the user the percentage of parking spots that are filled up during that given time. These stats will be updated in real time to help the user decide which parking lot to go to, as a lower occupancy percentage would increase their odds of finding a spot to park.

If the user has already logged into their account, they can press on one of the parking lots with no interference and can proceed to authenticate themselves at a particular parking spot. The app will scan for nearby blue tooth beacons and will render a list of them on the screen. The user will then choose the appropriate beacon and select it to authenticate themselves and claim the spot. In our project, this manual authentication process is designed to serve as a backup option for the authentication process in case the automated authentication process using image recognition fails. If the user has not logged in, they will be alerted to log in first and will be redirected back to the parking lots screen.

**Screen 5: Change Password**




(a) Change Password

Figure 5.14: Change Password

An additional feature for the user is the ability to change their password. This is helpful for two reasons. First, if a student decides to register themselves, the administrator will assign them a temporary password and a QR code to use. The user can then go on this screen and change their password to something more suitable to them. Secondly, the user can change their password at any time they like. If the user wants to change their password, they will need to enter their current password and the new password they want. An alert screen will indicate them if the changes have been made or whether they have entered the wrong current password.

### 5.7.2 App Flowchart

For our mobile application to function properly, we have designed it to follow a certain protocol as defined in the following flowcharts. By using a properly defined protocol, the user can navigate through the several screens appropriately and use the app seamlessly.

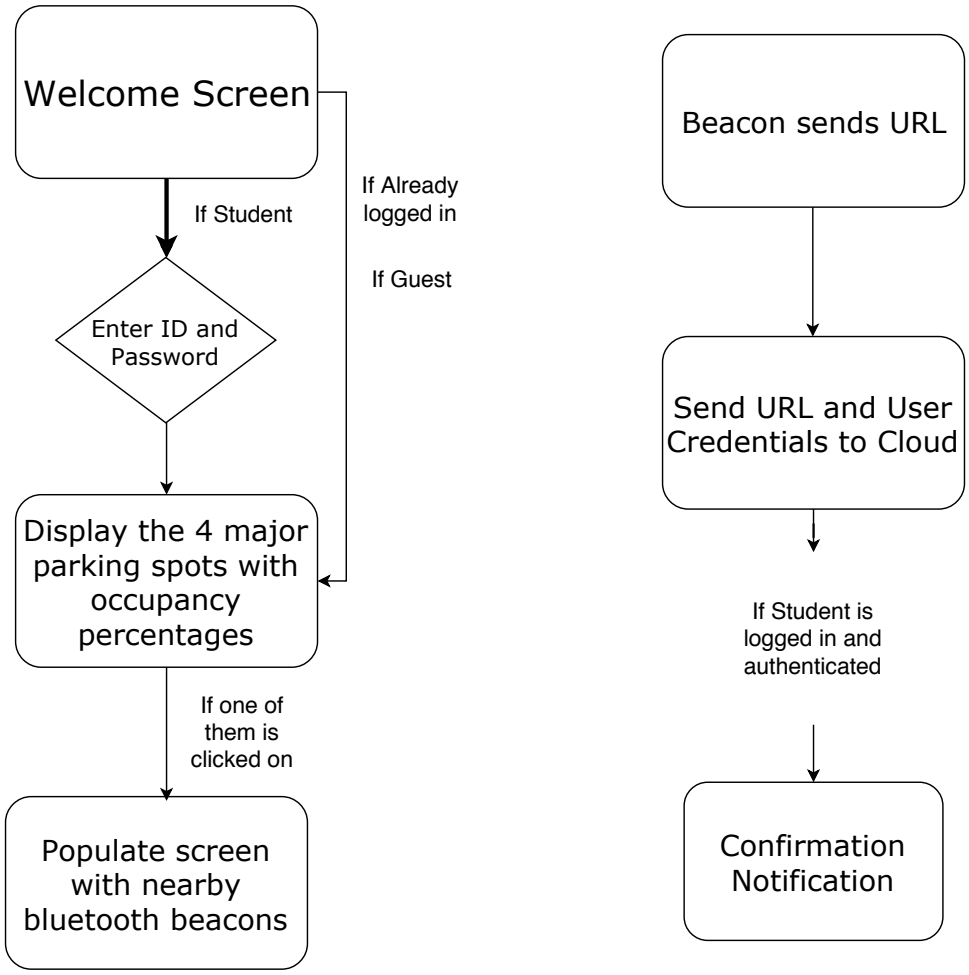



Figure 5.15: Mobile App Flowchart

The flowchart on the left guides us through how the app screens on the phone would function and move depending on the user input. As described in the design section, the app will start off with a welcome screen and will prompt the user to state if they are a student of guest. If they are a student, the student login page will pop up before going to the parking lots page. If the user has already logged in once, they would be automatically directed to the parking lots page. If they client presses the guest button, the app will directly go to this page. On the parking lots page, the user can look at the occupancy percentages of each parking lot. If the user wished to manually authenticate themselves, they would proceed to click on the parking lot they are in. On this screen, the app will scan for nearby bluetooth beacons and populate the screen with a list of them. The user can then choose the corresponding beacon and press a button to claim that particular parking spot. This is how the UI flow is designed for our mobile app.

The flowchart on the right describes what would happen when the user is trying to authenticate themselves once they park into a spot. The user will have already parked at a spot of their choice and simply want to authenticate themselves before exiting their vehicle. If

the student is already logged in, the app will jump straight to the authentication page at any time once it obtains a response message from the cloud. The user would then have to press a confirm button to explicitly state they want to park at the spot and finish the authentication process. If they are not logged in or are using the app as a guest, the welcome page would pop up again and then ask whether they are a student or guest before going to the authentication screen.

### 5.7.3 Communication with other devices

While the Mobile app can be designed and implemented, it is essential that it can communicated with some of the other devices in our project and does so in a proper manner. For the mobile application, there are two other components that it will communicate with for our project: the Google Cloud Datastore and the sensor box. The following flowchart maps out how communication between the app and these 2 devices are set up.

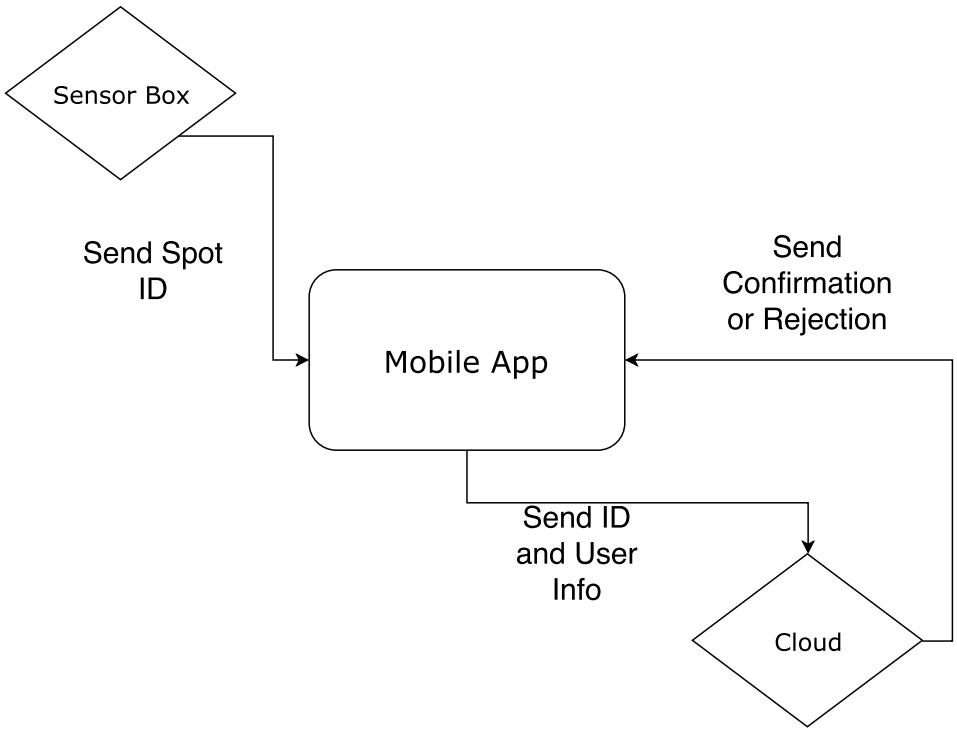



Figure 5.16: Program flow during authentication

To start off the authentication procedure, the sensor box will send a message to the mobile app that will contain the spot ID and the mobile app will parse this and send it to the cloud as a JSON object. The communication with the sensor box is only one way, and therefore as a result, the only time the app will need to consider getting a message from this device would be when a user first pulls into a parking spot and is waiting to be authenticated.

On the other hand, the cloud will be communicating with the mobile app more frequently.

The only time the mobile app will send information to the cloud is when a user is logging in or when the authentication process occurs. During other times when the app is used, the mobile app will request the cloud to send information about which spots are occupied or vacant. The cloud will send this information as a JSON object that will contain the spot ID and the status of the parking space. The mobile app will take that information and use it to display which spots are vacant or occupied using a color scheme of red and green. On the screen where the user can view all 4 parking spots with occupancy percentages, the percentage will be computed by the mobile app by looking at how many spots are occupied and comparing it to the total number of spots available in a particular parking lot. As a result, there is only one instance when the app will send information to the cloud while all other communication between the two is one way (from the cloud to the app).

### 5.7.4 App Software

Our goal is to develop an app that can be downloaded on a users mobile device and have the capability to send and receive data from the cloud. The app will have all of the UI and most of its functionality downloaded on the user's phone, thus making it a native app that will run on the phone. Most of the time, the app will only receive information from the cloud. The only time the mobile app will directly send information to the cloud is when the user logs into their account or when the message from the bluetooth beacon has to be sent to the cloud.

In order to write this program and deploy its first iteration, we decided to write and design the app using React Native. React Native is a framework that makes it easier to design an app for mobile devices and utilizes React Javascript to program not only the functionality of the app but also design the user interface for it. Using React gives us several advantages. For one, most of the app can be written using this approach and can be deployed as a cross platform app, thus making it easy to develop for both iOs and Android. Secondly, React provides extensive libraries that can help with setting up the user interface along with taking care of proving us with the necessary libraries that will enable us to deploy it on the desired mobile device. Third, it is a lot easier to develop the app using React Native since it deploys itself as a single page application, thus updating it frequently every time a change is made during development mode.

In order to utilize the Bluetooth beacon and receive a signal from it for authentication purposes, the app will be programmed differently for Android and iOs. Our sensor box will be programmed to utilize Eddystone to send out the message to the app. On iOs, this is the protocol that is used be default, and as a result, the react Native app will import libraries from xCode (the programming development environment for iOs devices) that will handle receiving messages from the Bluetooth beacon. On Android devices, we can import libraries that can handle the Eddystone protocol and thus personalize it to suit our needs.

We also considered other approaches when designing the app. One idea was to deploy another app on the app engine provided by Google cloud storage and make the user access

the app by using an URL. However, our ultimate goal was always to design an app that is native on the mobile device itself, and thus this approach was not pursued.

### 5.7.5 Sending and receiving data

The Mobile app will be heavily sending and receiving data from our cloud. During this process of interchanging data, well formatted and structured data is required. We plan to use JSON for carrying out this process since it is good and well structured, lightweight and easy to parse and human readable.

The mobile application is responsible to send information to the cloud if the user is a student and enters their credentials. Our JSON object would be formatted in this following way:

**"data":** [
**"userid": "1234567",**
**"password": "PassWo$d"**
]

Our cloud utilizes Google Cloud Datastore to keep a log of students who are authorized to park on campus. One of the databases on the cloud stores student's first and last names, their Student ID, their password and the permission levels they have for parking. When the student logs in on their phone and the app sends their student ID and password to the cloud, the cloud will process this information and send back the name of the client to confirm who is using the app. Similarly, when the mobile app receives a message from the bluetooth beacon, this message is parsed on the mobile App and the spot ID is sent to the cloud along with the user credentials. This JSON object will contain information of the parking lot and the client's parking privileges to authenticate if they can park in that spot. When the cloud sends back its message, we will use a JSON object once again to send back this information to the phone.This JSON object will look as follows:

**"data":** [
**"userid": "1234567",**
**"Parking Lot": "West Remote",**
**"Spot ID": "D23",**
]

To summarize, the app will utilize a standard GET or POST request when sending or receiving data from the cloud and will use a JSON object to send or receive the necessary information.

### 5.7.6 Dependent Libraries

**crypto-js**: This library in javascript contains a large collection of standard and secure cryptographic algorithms implemented in JavaScript. They are fast, and they have a consistent and simple interface. This will be necessary to process JSON objects that contain a student's blue password or a guest's payment information.

### 5.7.7 API's

The APIs of the mobile application are not so much APIs, but rather calls to APIs that are on the cloud. The cloud server will host written functions and scripts, and the mobile application will simply call the appropriate API's based on what operation the user wants to execute. For the list of API's that will be used for the mobile application, refer to section 5.5.4

### 5.7.8 Bloom Filter Programming

The mobile application calls a function written in Swift to encrypt the secret message with secret symmetric keys. The encryption uses AES algorithm. After encryption cast the cipher text into an integer and modulo the integer based on the length of the bloom filter. Then for each of the result set the corresponding bit in the bloom filter array to 1.
When sending the message to the Admin Application, the Initialization Vector as well as the Bloom Filter array would be send for the authentication scheme in between two separate system.

## 5.8 Administrative Application

The web application is an application that can be run on a browser. The application gives TAPS personnel unlimited access to the information on the cloud. This means they are able to designate parking spots as taken, even if they are not, essentially reserving them for people/events. The application is written in HTML and a series of other languages that are related to HTML including, but not limited to CSS, javascript,and python. In the following pages we have images that help outline the capabilities and the design of the administrative application. We will first begin with the general flow of how the Administative application looks like in figure 5.17.

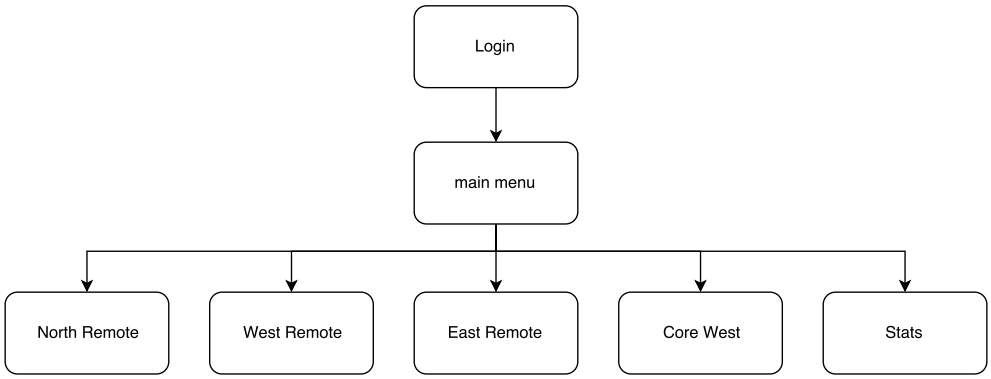



Figure 5.17: General flow of admin app

So what we see in the general flow of the application is that the administrator will log in, and if their credentials are valid, be taken to a main menu page. then from the main menu they can choose from the various options to be able to look at each of the lots and the overall statistics if they want. Now we will show what each of the various pages looks like and their wide range of capabilities.

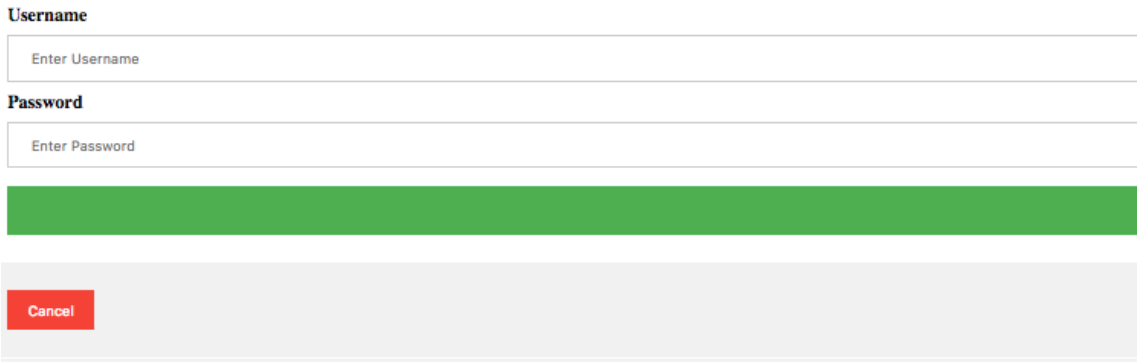



Figure 5.18: The administrator must log in to access the application

For fairly obvious security reasons, the administrator will have to log in. They will be able to set their username and their password, but they will have to type them in every time they want to use the application. This prevents unauthorized users such as students from accessing the application and abusing it for their own purposes. Once the adminstrator has input their credentials they are then redirected to the main menu of the application.

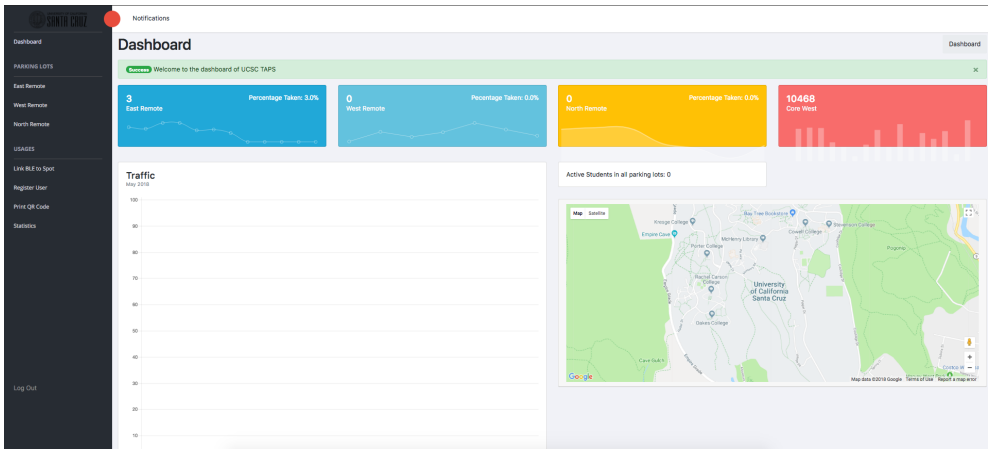



Figure 5.19: Main Menu

What we see in figure 5.19 is a picture of our main menu for the administrative application. We are currently planning on supporting the North Remote, East Remote, West Remote, and Core West parking lots for out project. each of the boxes in figure **??** will tell you the fullness of each lot as a percentage. So as we can see, the North Remote parking lot is fairly empty. Then the administrator can click on the lot that they wish to examine on the sidebar menu. This will redirect them to another page that will contain a google satellite image of the parking lot with little objects overlayed over each parking spot. Then when the administrator clicks on a spot, a little window will show up that will show the available actions of each spot. Figure 5.20 shows what this page looks like. The administrator app will also be able to add students to the cloud database and generate a QR code to attach to their permit. These pages are fairly simple with only text boxes on them to submit the relevant information needed to generate what the administrator wants.

What we see here is a Google satellite image of the lot. Laid on top of the image are little dots that represent the spots. white is a free spot, green is an authorized occupied spot, and red is a spot that is occupied but unauthorized. Then when the administrator clicks on a spot, a little dropdown menu will appear with a list of actions that the admin is allowed to do. This dropdown menu can be observed in figure 5.21

Aside from observing the individual parking space, the administrator is also able to change the various parameters of the parking lot. On the preceding page, in figure 5.11 we see that the administrator is changing the permits that are allowed to park there and the times that they are able to park there. So, when the administrator hits the submit button the West Remote parking lot will only allow A and R permit holder to park there

The administrator is also able to manipulate the individual parking spaces in certain ways. The administrator should be able to reserve parking spaces in case there is a guest visiting the school. If that happens and there is no parking, that would reflect very poorly on the school. So, when the administrator clicks on a parking space, there is an option to set the parking spot to be reserved. When this happens the page will send a signal to the cloud to

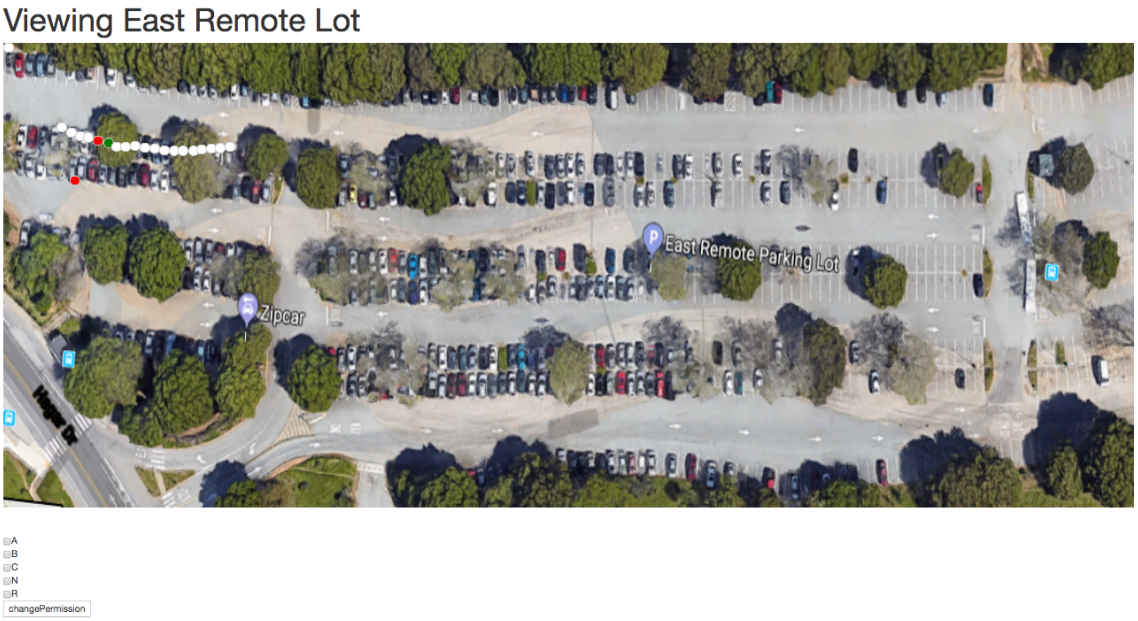



Figure 5.20: screen of lot for administrator application

set that space to occupied. Then the mobile application will pull this information from the cloud and that particular parking space will be seen as reserved on the application, thus securing a spot for the school's important guest. This can be observed in the figure on the next page.

Looking at the dropdown menu, we can also observe a multitude of other actions that the administrator is allowed. They are allowed to claim,free,view,and change the permission of each individual spot. Of course, this means that it wont free the spot in reality, but in the system it would reflect as free. The other functions will work in the same way. However, most important is the view the tab. The view tab is a link that will redirect the user to another page that will display various information about the spot in question. Figure 5.22 shows what information is available on the webpage. As you can see, the admin can see who the name, permit type, license plat photo, QR code photo, and student ID of whoever is parked there. This will help the admin keep track of who is parked.

Aside from looking at the individual parking spaces and the immediate fullness of the parking lot, the administrator should also be able to look at the general statistics of the parking lots. They should see charts and graphs that depict the usages of the parking lots on an average basis and be able to look at the usage charts. Figure 5.23 shows what this looks like.

In Figure 5.23, What we have above is fictitious data of the usage of the parking different parking lots throughout the day. As we can see in the figure, we can see how the North Remote parking lot becomes full and empty as the day progresses. At the top of the figure there are buttons that allow the administrator to look at the general usage of the parking lots

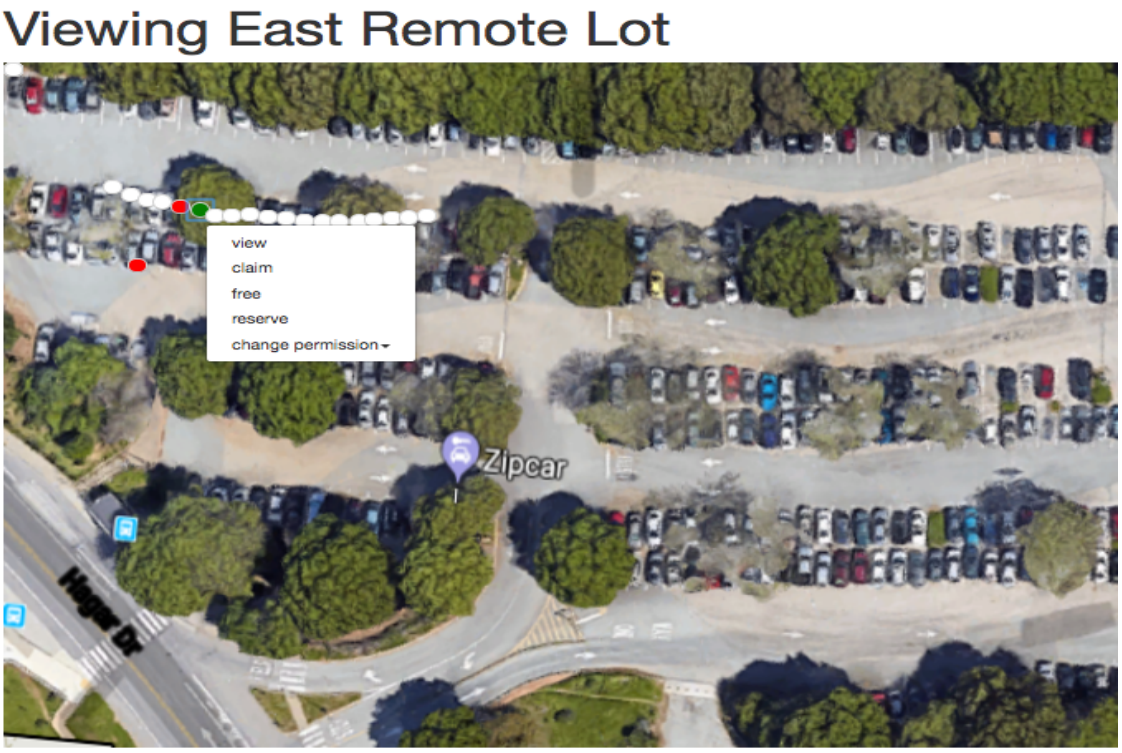



Figure 5.21: Administrator can reserve spots by clicking on unoccupied parking spots

over different time frames. They can observe annual statistics of the usage of parking lots. Aside from those functionalities observed in the figure, we also have the ability to observe more focused statistics. By selecting a date from the calendar menu at the top of the page, the administrator can observe parking usage among all lots on any given day.

### 5.8.1 QR CODE GENERATOR

Our administrative application will also have a page to help the administrator create a QR code that each permit will have attached to it. The page will also have a button to create a new student in the database. After the new student is created, the administrator will then also be able to type in a QR code that will be attached to that student's entry in the database. Then by pressing another button, the QR code will be generated and the administrator can then print it out and attach it. Figure 5.24 shows the page to register a student in the database. The admin simply puts the corresponsing information in the boxes and hits the submit button at the bottom and they are done.

### 5.8.2 BLOOM FILTER QUERY

Without authentication scheme, the bloom filter aims to establish an authentication method for the Web application and the mobile application. When receiving a POST request from the Publicly available URL, the Web Application receives an Initialization Vector and a

| | |
|---|---|
| Space ID | 1 |
| Authorized | True |
| Student ID | 1562575 |
| First Name | Troy |
| Last Name | Fine |
| License Plate | 5RUC649 |
| Permission Level | A |
| Permit Code | Fia9TZj9PWgUk |


Figure 5.22: Administrator can who is parked in each spot and a picture of the license plate/QR code

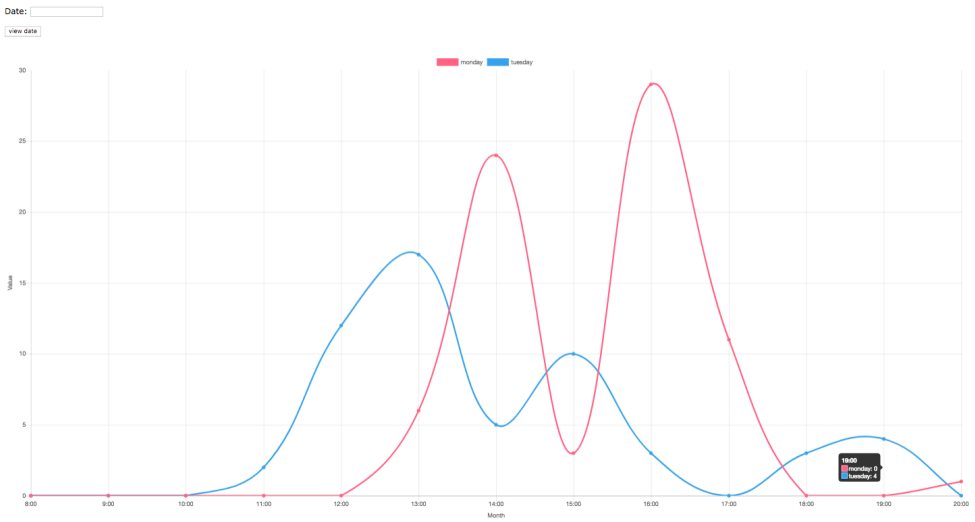



Figure 5.23: Administrator can observe and analyze usage of the parking lots on multiple bases


Figure 5.24: Administrator can create a new student in the database and then print out a QR code for them to put on their car

Bloom Filter Array. Then Web Application encrypt the Secret Message with this IV and secret symmetric key and turn the secret message into a integer to modulo the length of the Bloom Filter Array. If all the result bit is 1 in the bloom filter array, indicating the same message was encrypted from the sender and the sender is a trusted party. If any of the bit is zero, the message is definitely not from a trusted party, and the message will be droped.

### 5.8.3 Administrative Application APIs

The APIs of the administrative application are not so much APIs, but rather calls to APIs that are on the cloud. The cloud server will host written functions and scripts, and the administrative application will simply call the appropriate API based on what changes that the administrator will make. to see the APIs that the administrative application can call refer to section 5.5.3

# 6 Testing and Simulation

For any product to be robust and reliable, we need to ensure that it has been tested thoroughly in order to minimize potential bugs. This section covers the testing and simulation plan that we utilized for our prototype. It contains a summary of each component tests, an outline of the implementation, and the various tests.

The overall test will make sure all of the individual components are able to communi-

cate with each other and work seamlessly in real time. When a user pulls into a parking spot, the sensor will communicate with the gateway and the gateway will communicate with the cloud. Meanwhile, the user can use their mobile app to authenticate the user which will require the app to act as a medium between the sensor box and the cloud.

## 6.1 Sensor Box

- The sensor box will be connected to a pseudo-gateway during testing.
- The pseudo-gateway act almost the same as the regular gateway with two differences. Firstly, it does not connect to the cloud. Authorization can be done manually by the user or be done randomly. Secondly, the gateway will have additional logs to help with debugging.
- The simulation will focus on edge cases that could cause the sensor box code to break.
- Making sure the sensor box can remove unwanted noise from the distance sensor is a primary concern.

### 6.1.1 Implementation Outline

- Create pseudo-gateway with logging capabilities
- Log serial output of the esp32

### 6.1.2 Tests

- Basic
  - Car entering with valid QR code
  - Car entering with invalid QR code or no QR code
  - Car entering slowly
  - Car leaving spot
  - Car leaving spot slowly
- Debouncing
  - Walking across the sensor with car and without car in spot
  - Car immediately leaving after entering the spot

## 6.2 Gateway

Multiple instantiations of simulated sensor box processes will test the gateway's ability to function correctly in that it can manage and exchange data between multiple sensor boxes and the Google cloud data store.

### 6.2.1 Implementation Outline

- Created a script whose purpose is to read a file with inputs that correspond to what would be inputs from the actual environment (ie hardcoding environmental values like time arrival) and to instantiate multiple simulated sensor box threads that take in the various inputs defined in the input file.
- Each simulated sensor box will perform the same as a regular sensor box, except with hard coded input values, depicted in an input file which should be taken as the first argument on the command line. Namely, a spot ID of sensor box, a photo in .png or .jpg format, a starting delay, and an exit delay, respectively.

  AB504D6E8F FS25U1R.jpg 5 13

### 6.2.2 Tests

- trivial-input
  - A single vehicle with an enter and exit event with normal flow.
- basic-input
  - A few vehicles with different enter and exit events with normal flow.
- semi-basic-input
  - A few vehicles with similar enter and exit events with normal flow.
- mediocre-input
  - A fair amount of vehicle with similar and mixed enter and exit events.
- strenuous-input
  - A lot of vehicles with similar and mixed enter and exit events pushing the gateways limit.

## 6.3 Mobile App

- The Mobile app will be downloaded natively to multiple phones and will then be used to simulate the test cases listed below. The mobile app will require the user to have their bluetooth enabled for the test cases
- The simulation will focus on cases where the user is a registered student with permission to park at a spot, a registered student who is not authorized to park at a spot, and a guest who has no authorization at all
- Making sure the mobile app can authenticate via bluetooth and can transfer data to the cloud is the primary objective and concern for these tests

### 6.3.1 Implementation Outline

- Export the app and download the it to the phone
- Establish connection with the cloud
- Authenticate the user using bluetooth

### 6.3.2 Tests

- Basic
  - Student entering a spot with permission to park
  - Student entering a spot without permission to park
  - Guest entering a spot without permission to park
- Advanced (Debouncing)
  - User already authenticated via QR code
  - User authenticates with the wrong sensor box (wrong spot)

## 6.4 Administrative App

For the administrative web app there is not really anything to live test perse. The webapp really only communicates with the google cloud server, so all we need to do it make sure that the information on the app matches the information on the cloud.

### 6.4.1 Implementation Outline

We will simply deploy the app to the google cloud app engine and use it on a computer/phone.

### 6.4.2 Tests

Test the functionality of the app, independently, as the rest of the test runs. As the rest of the test runs and make sure the information on the app matches what is on the cloud.

## 6.5 Cloud

All of aforementioned tests will utilize the Google Cloud Datastore

### 6.5.1 Implementation Outline

Run the tests above and while they are in the middle of operation see if the status of the datastore is what we expect it to be

### 6.5.2 Tests

We can use functions that can return the state of the lot overall and when we know what spots are supposed to be claimed or free, based on where we are in the tests, we can run this function to see if the state of the parking lot matches up with what it should be.

# 7 Demonstration

The demonstration serves as a real life example of how the parking lot works and how it benefits both users and the administration.

There are three basic scenarios we will need to demonstrate in order to show our overall system is functioning correctly:

- A car pulls in with a valid permit that should be recognizable by the sensor tower, which will cause the space in the cloud to be occupied and the correct user becoming active in the system.
- A car pulls in and the permit cannot be deciphered, for whatever reason, which will force them to authenticate themselves using the Mobile Application.
- A car pulls in with no permit at all and, again, the user will have to use the Mobile Application to register and authenticate themselves.

# 8 Hardware Parts List

The following list comprises of the hardware parts that were used to build and design the first prototype of the project.

| Quantity | Items | Price |
|---|---|---|
| 2 | Raspberry Pi 3 | $60 |
| 2 | ESP32 | $26 |
| 2 | WS2812B LED Strip 144 LED/m | $34 |
| 2 | OV7670 | $26 |
| 2 | SR04 | $10 |
| **Subtotal** | | $156 |