UNIVERSITY OF CALIFORNIA, SANTA CRUZ

Campus Parking Infrastructure: Team 3

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

March 20, 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

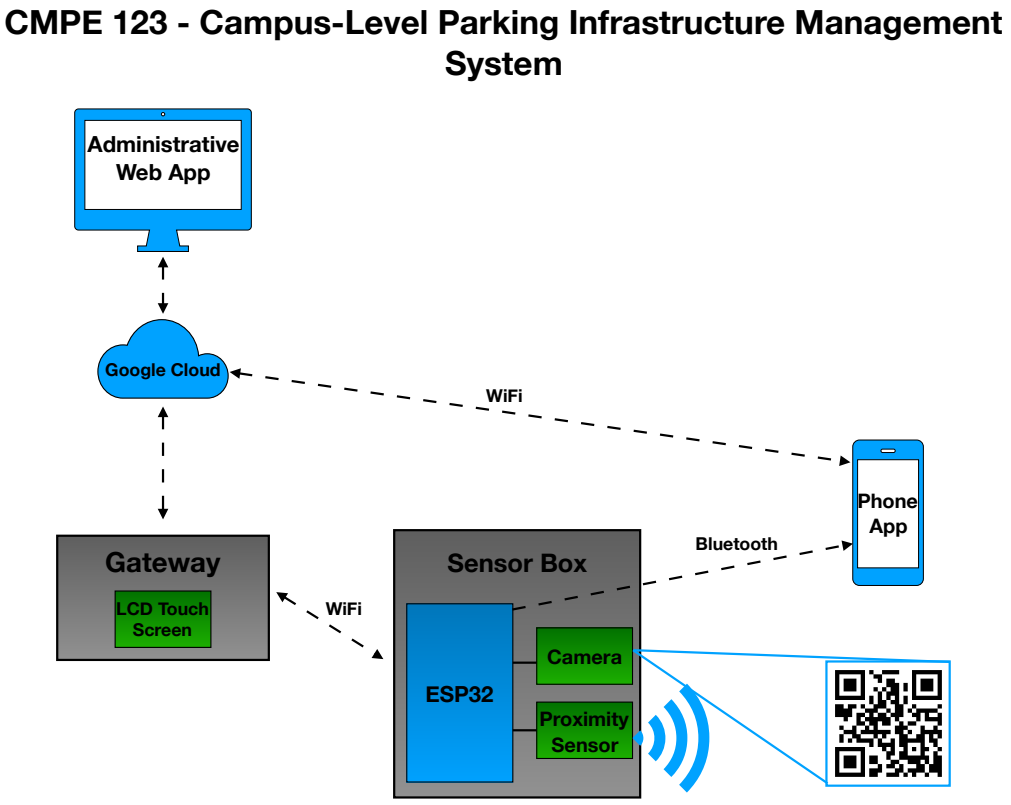


Figure 3.1: Top Level Block Diagram

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. A guest would have the option to pay for parking also using their mobile device.
- 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 Project Goals and Objectives

The goal for our team is to design, implement and deploy a prototype of this project by the end of the spring quarter.

5 Sensor Tower Prototype Design

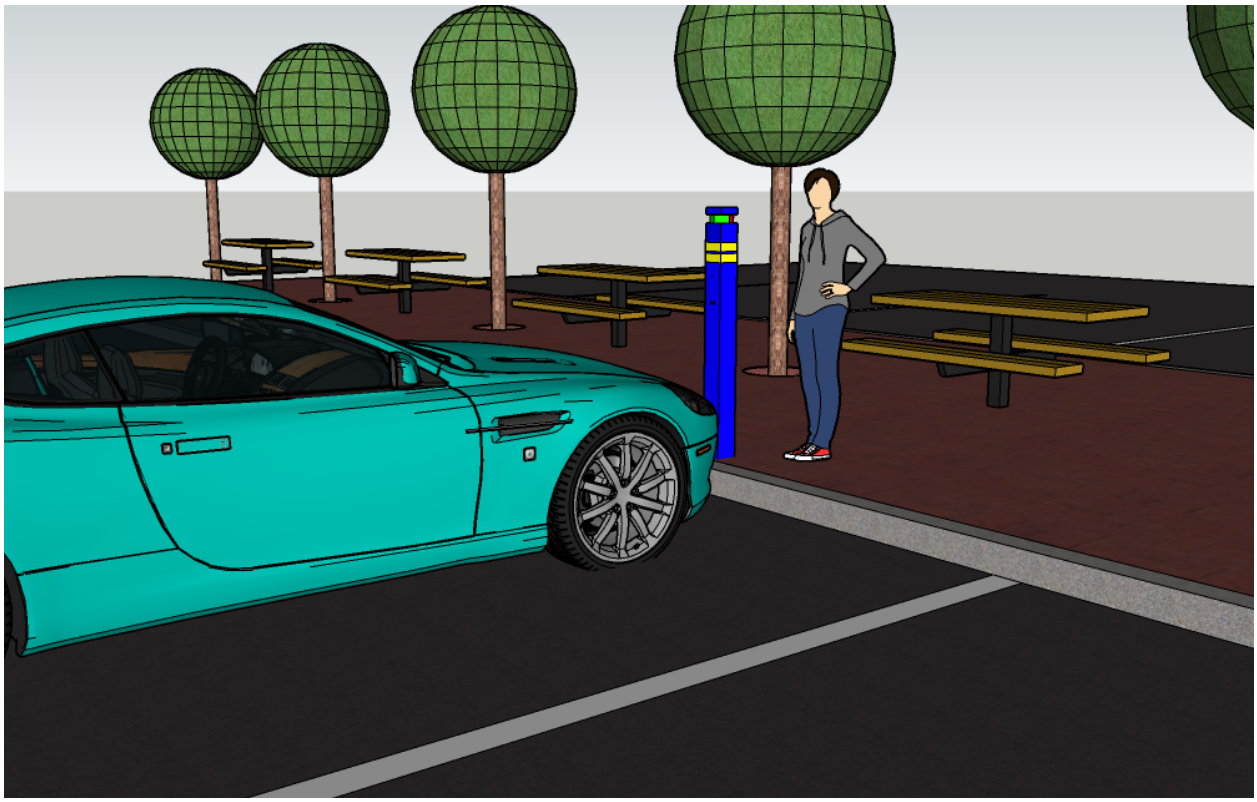


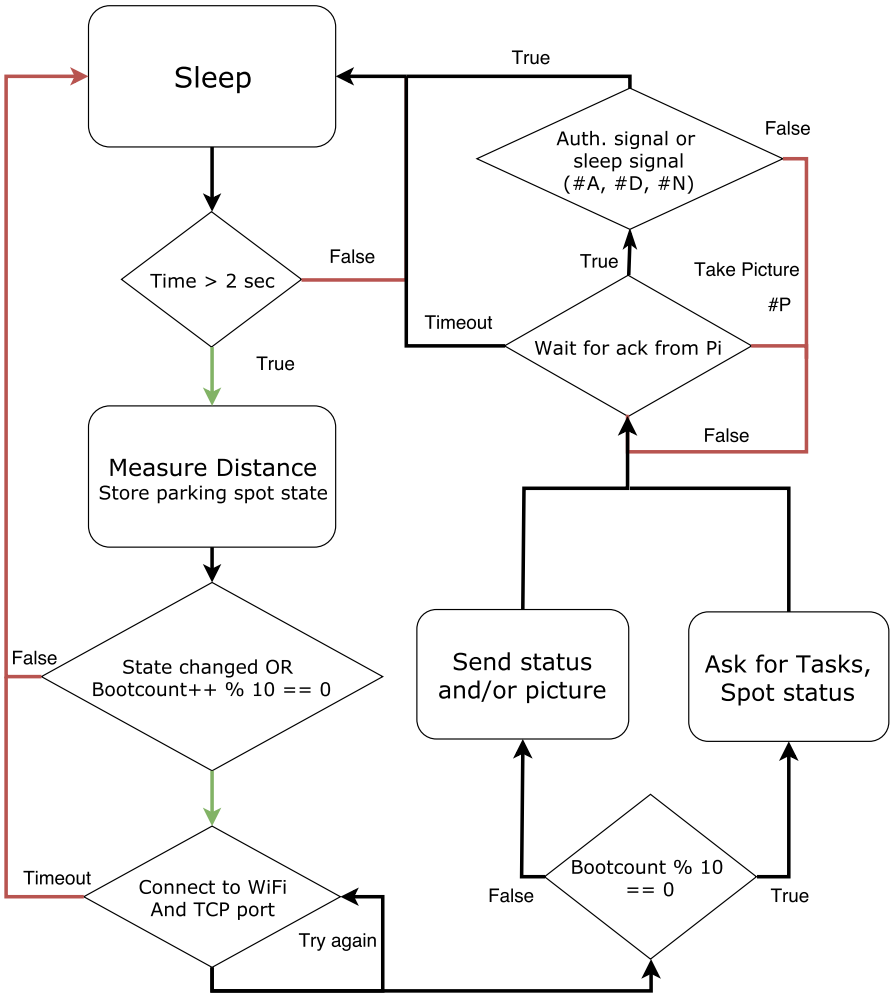
Figure 5.1: Parking Space Sensor Tower Prototype Design

6 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.

6.1 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 6.1: State Diagrams

Figure 6.1 covesr how the sleep function, car detection and gateway messaging works.

6.2 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. 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.

6.2.1 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 6.1: 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 6.2: Gateway to Sensor Box Messages

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

6.2.2 PIN CONFIGURATION

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

Table 6.3: SR04

SR04	ESP32s
VDD	Vin
ECHO	27
TRIG	26

Table 6.4: Camera

OV7670	ESP32s
VDD	3.3v
SDA	21
SCL	22
VSYNC	34
HREF	35
XCLK	32
PCLK	33
D0, D1,..., D7	25, 2, 18, 15, 19, 5, 23, 4

Table 6.5: RGB LEDs

LEDs	ESP32s
Red	14
Green	12
Blue	13

6.2.3 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:

```
http://domain.com/SPOTID
```

The URL can then be used by the app to identify what spot the user is trying to park at. See section 6.8.3 for more details on how the app uses the URL.

6.3 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.

6.4 Display

Each gateway can either come as just a stock standard gateway, or an alternative design can be used. This alternative design not only acts as a gateway to multiplex data from one place to another, but it also acts as a parking kiosk, providing users with a clean interface in which they can pay for parking. Given the relatively low price point of the displays and interfacing components, for large lots like east remote, the administrator might want to add multiple kiosks to the parking structure in addition to standard gateways. The display provides first time users and guests with a method of payment and lets them know about the smart phone application which provides similar easy payment options and lot information.

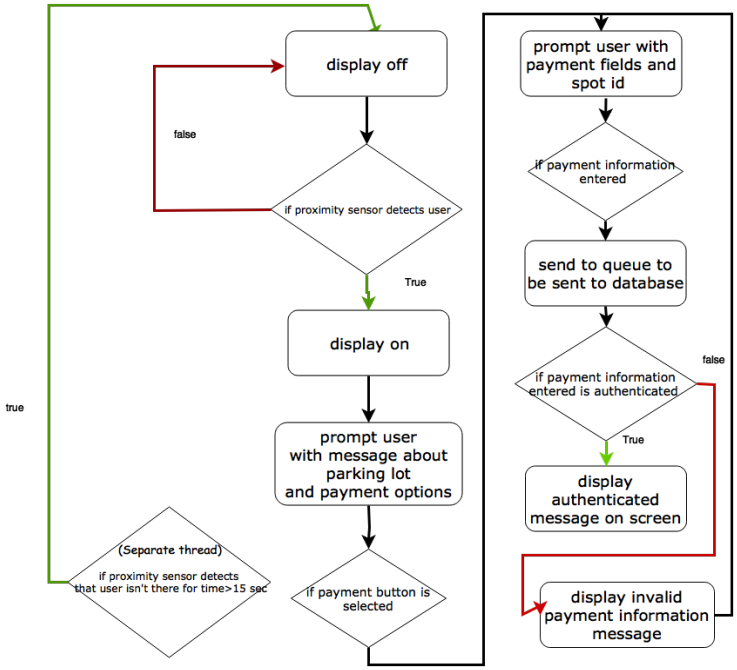


Figure 6.2: Display Flowchart

The figure above shows the basic flow of the display. Essentially, it will use a proximity sensor to detect if someone has walked up to it, and it will turn on the display and prompt them with information about the lot and payment processing options. Once the user has put in valid payment information, the database will be updated accordingly, and the user will receive an authentication message on the display.

6.5 Gateways

The gateways (Raspberry Pi 3) serve as a intermediary step between the sensor tower and the cloud, multiplexing data to and from the sensor tower and the cloud. There is a single level of hierarchy for gateways, but multiple gateways for most parking lots of size. Each gateway will be arbitrating data between multiple sensor boxes and the google cloud datastore. On top of this, it also performs computational and user interface tasks in certain cases.

6.5.1 High Level Overview

The gateway can broken down into several primary functionality. Firstly, hosting a WiFi LAN hot spot for the sensor boxes to connect and disconnect from. Secondly is the intermediary communicator between this network of sensor boxes and the Google cloud datastore. Thirdly, it is responsible for taking the sensor box's photo of the car with the QR Code in the windshield and translating that into a string which is forwarded to the cloud. Finally in certain models of the gateway, it contains a display which primarily functions as a kiosk as an additional method of payment processing.

The basis for the interconnection between the each of the gateway and surrounding sensor boxes is a master slave architecture based off of basic socket protocols, where the slaves are the sensor boxes and the master is the gateway. However, the gateway also functions as a client in the gateway to database relationship.

In the figure below, the basic flow is outlined into multiple chunks laid out in a modularized fashion for the purpose of organization ease of understanding. It outlines the primary sections that make up a gateway that can handle multiple sensor boxes, various tasks, and implemented with techniques to prevent data loss.

6.5.2 UDP Discovery Protocol

After Starting the gateway, a static IP will be assigned to it, and it will then act as a hotspot. From there, it will use UDP as a discovery protocol (on first boot and setup configuration) to find sensor boxes in its proximity and ensure that no sensor boxes have died.

6.5.3 Primary Data Types and Setting up Server

The primary code begins with the declaration of some important data types. Firstly, a class is created with the intention of holding information about the sensor towers. This is

implemented as a list of objects whose fields are initialized to 0. A queue is then implemented for processing and buffering gateway to cloud communication to ensure that the socket connection underlying the google API functions is not disrupted which may cause slow downs or even packet loss. After this is done, the SetupServer( ) function is called which essentially declares a socket of type IPv4, TCP and binds it to its existing static ip address and defined static port number.

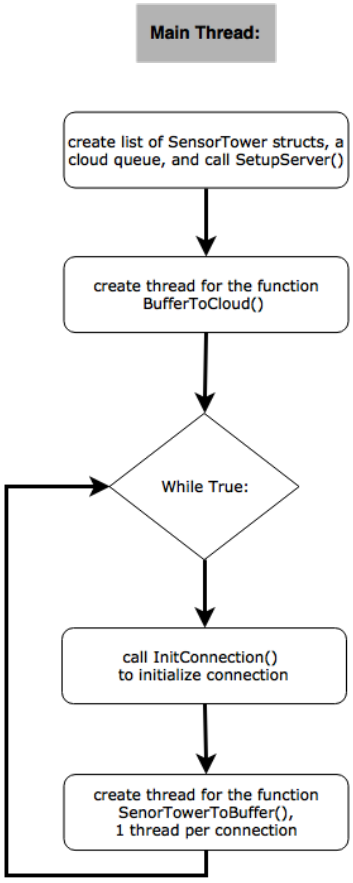


Figure 6.3: Main Thread Flowchart

6.5.4 Buffer to Cloud

The Proceeding the creation of the data types, we must call the BuffertoCloud( ) function as a new thread, to ensure that it does not get blocked by the other components of the system. This function takes from the data that came from the sensor towers and was queued, and pops the value off the top of the stack, and forwards it to the cloud when ready, which then responds with the authentication status, where by the cloud is now free to accept more data.

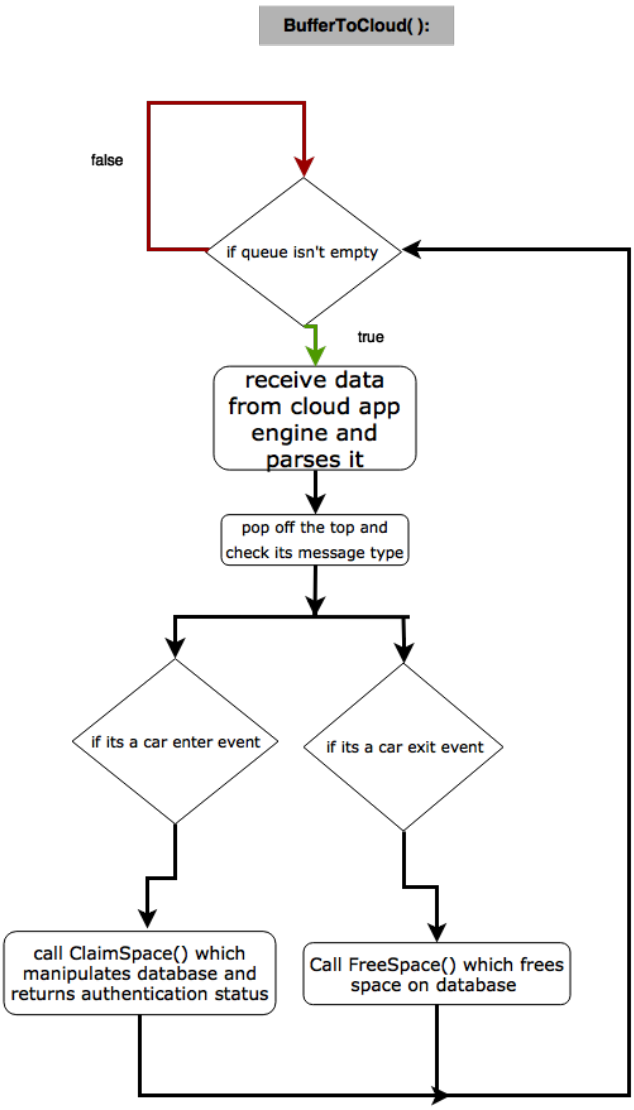


Figure 6.4: Buffer to Cloud Flowchart

6.5.5 Initialize connection and Sensor tower to cloud

After all of these functions and setup tasks have been completed, we are going to go into a while loop and call the initConnection( ) function that waits for an incoming connection and accepts it using the TCP 3 way handshake. After this has been completed, a new thread is made with the function SensorTowerToBuffer( ). This takes the data from the sensor towers, processes any images and tasks, and pushes message into the buffer that will forwarded to the cloud by the function described in the previous section. After forwarding, it then waits for a response from the cloud and forwards the authentication resonse from cloud to the corresponding sensor tower.

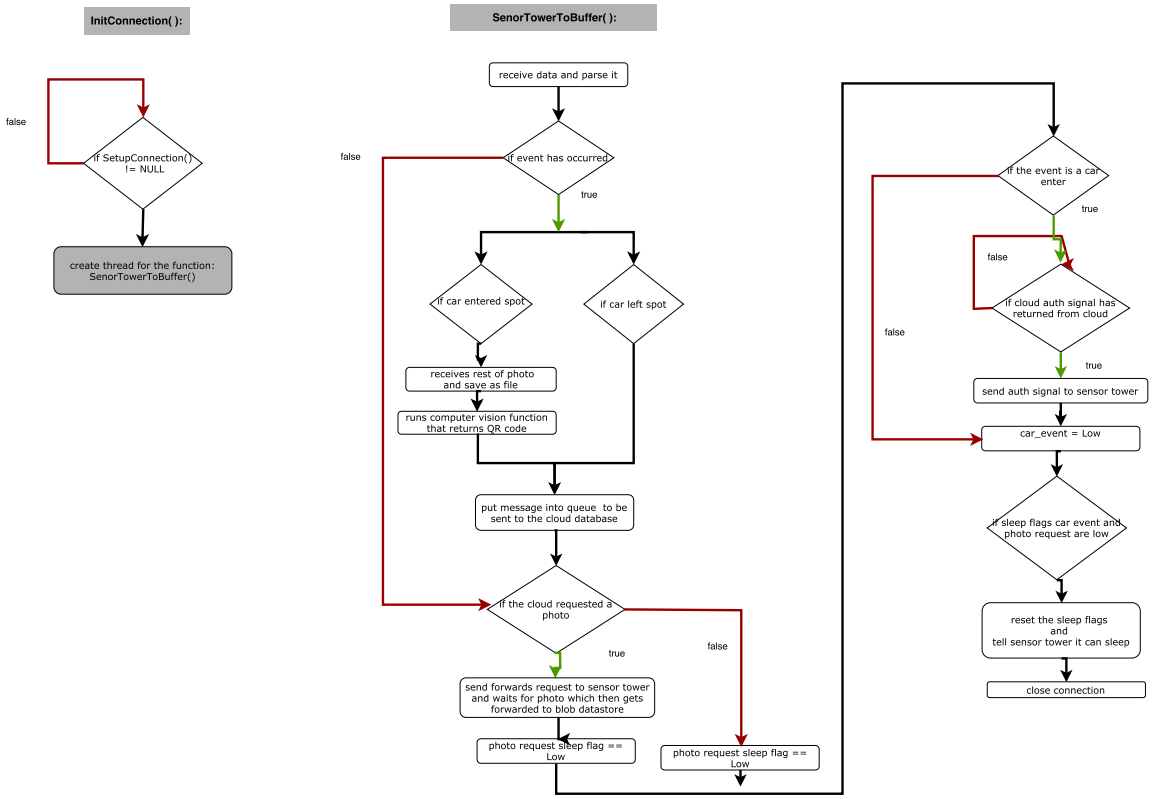


Figure 6.5: Sensor Tower To Buffer Flowchart

6.6 Cloud

6.6.1 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.

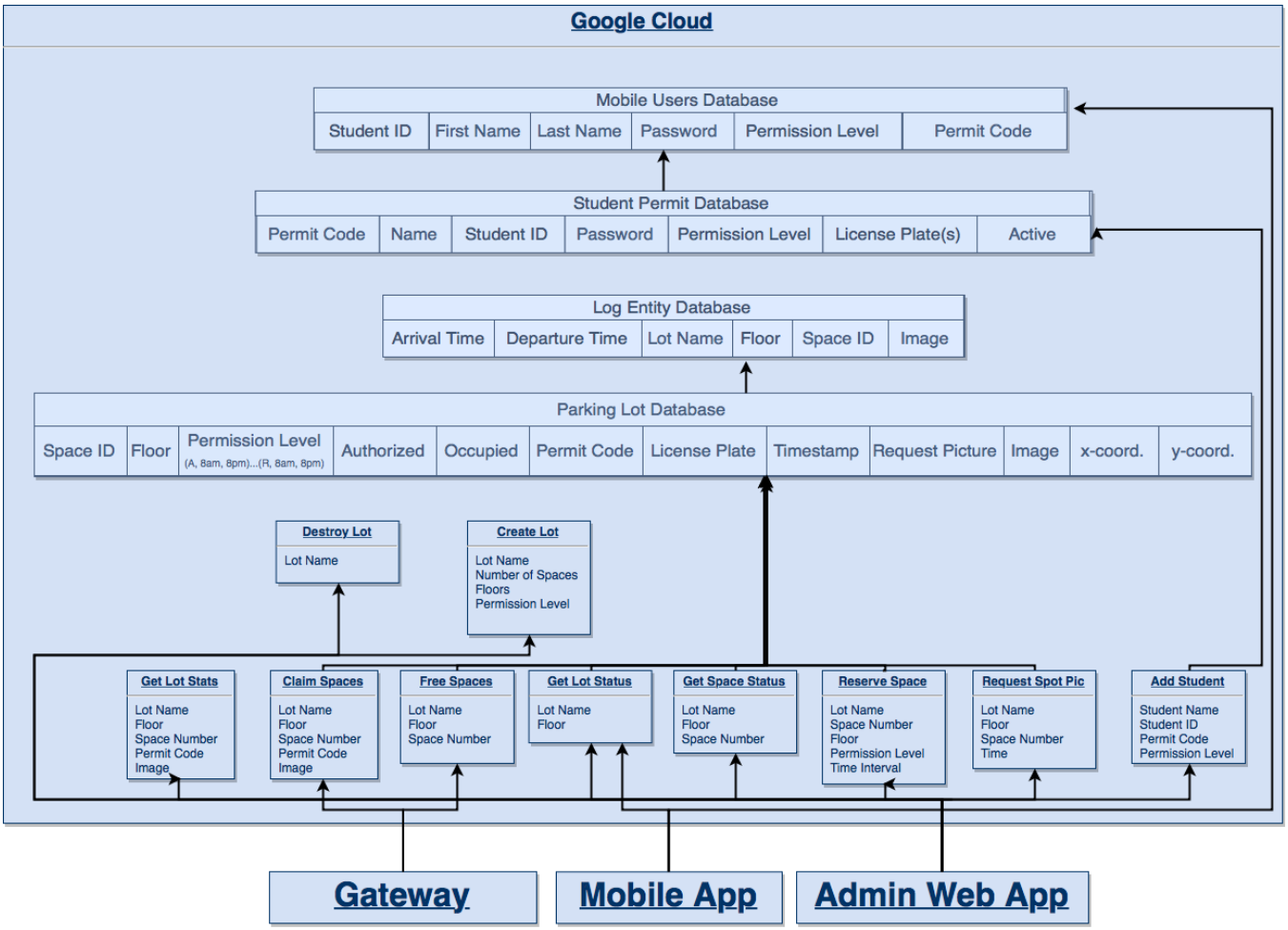


Figure 6.6: Cloud Data/Flow Block Diagram

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

Figure 6.7: Parking Lot Database

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

Figure 6.8: Student Permit Database

Log Entity Database					
Arrival Time	Departure Time	Lot Name	Floor	Space ID	Image

Figure 6.9: Log Entity Database

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

Figure 6.10: Mobile App Users Database

6.6.2 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.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
timeframe = space_['Timeframe'] \\ date and time
space_['Timeframe'] = datetime.datetime.now() \\ date and time
datastore_client.put(space_)
```

The Administrator has the ability to request a picture from any spot by indicating a parking lot, a floor number if applicable, and a space number. They can either view the picture of the car when it first parked, a current picture of the space, or a picture that has been logged into the database that keeps track of cars that previously were in the parking space.

- get_spot_pic : space_num

6.6.3 Administrator Web App to Cloud APIs

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

- create_lot : space_num | lot_name | permission_level | floors

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, floor number if applicable, 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 | floor | permission_level | start_time | end_time

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

- add_student : name | student_ID | permission_level | code

Administrator wants to view the status of a lot

- view_lot_status : lot_name | floor

The Administrator can also erase any lot from the database by specifying the lot name.

- destroy_lot : lot_name

The Administrator has the ability to request a picture from any spot by indicating a parking lot, a floor number if applicable, and a space number. This request will be sent to the Gateway, which will then route the request to the corresponding sensor box and send it back up through the gateway, to the App Engine and back to the Administrator.

- get_spot_pic : lot_name | space_num | floor

At any time the Administrator can see the status of any space in any lot. This information would include the Parking Lot name, floor, space number, permission level, the permit code of the student, and the license plate number currently in the space.

- get_space_attributes : lot_name | space_num | floor

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

6.6.4 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

User is a student that wants to log in:

- student_login : student_ID | password

User wants to view the status of a lot:

- view_lot_status : lot_name | floor

6.7 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.
This module now consists of two parts. The first part is a QR code generator that would take 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 QR code generator would run on the administrator app that allows the admin to generator the QR code based on the database and give the permit to the students. 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.
For security purpos, when the QR code in this module is used for authentication of a spot, the Mobile App would give a push notification to the user telling them they are using their code. If that's not them, they can simply hit deny to deny the authentication. This is preventing other people from printing the same QR code by simply take a picture.
The libraries needed to run the Computer Vision part are specified in the Dependent libraries section.

6.7.1 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.
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. This part would be implemented in the next quarter.

6.7.2 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.

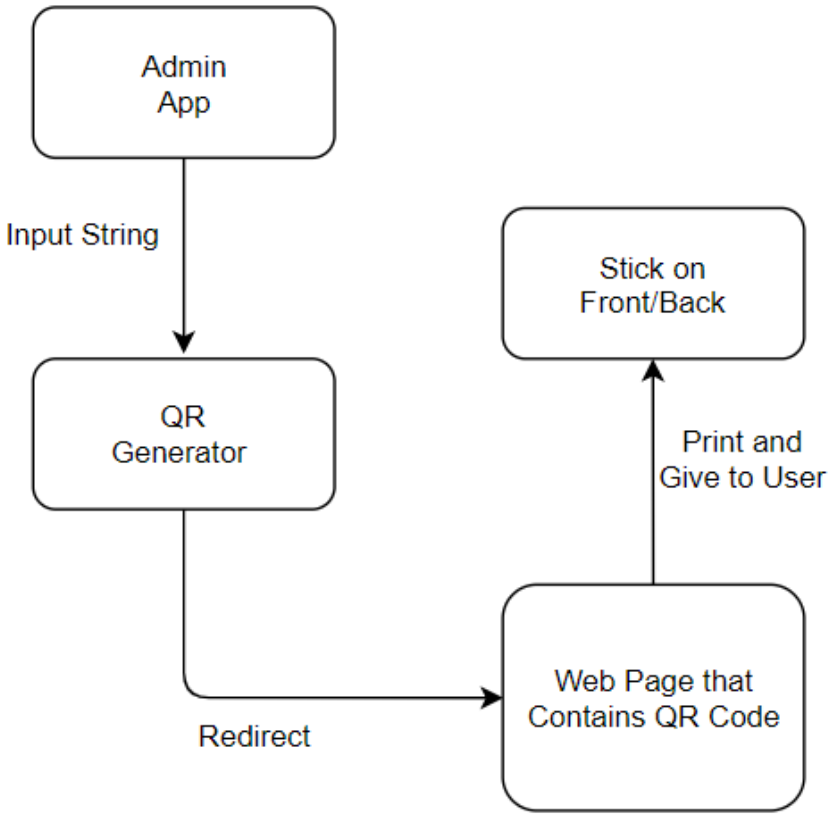


Figure 6.11: QR generator Flowchart

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

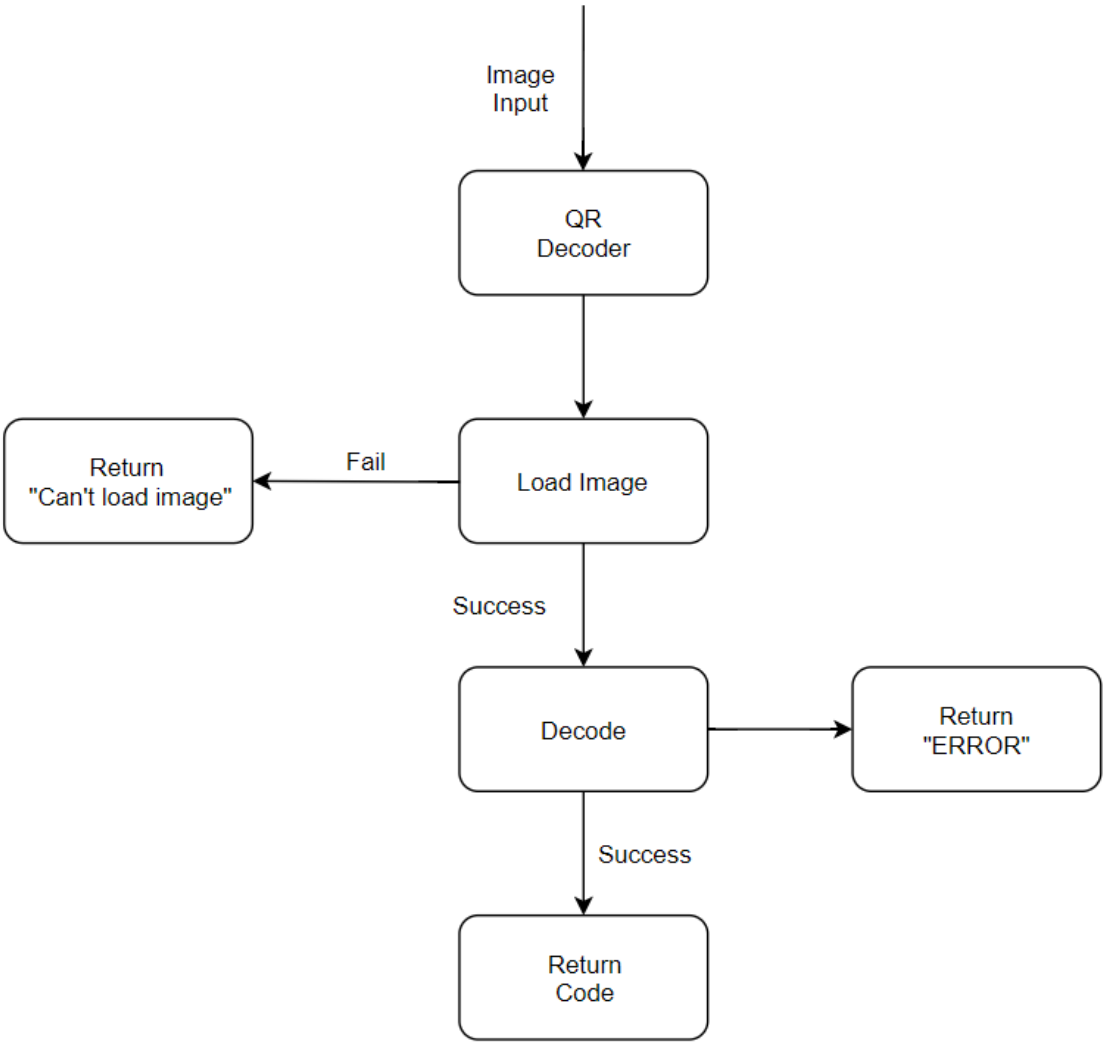


Figure 6.12: QR decoder Flowchart

6.7.3 Dependent libraries

Python: The majority of the Computer Vision part would be written in the Python programming language.
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.

6.7.4 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, the
`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.
`request_image()`
This function takes in the period of day and resend all the pictures taken at that hour to the gateway or the cloud for the administrator.

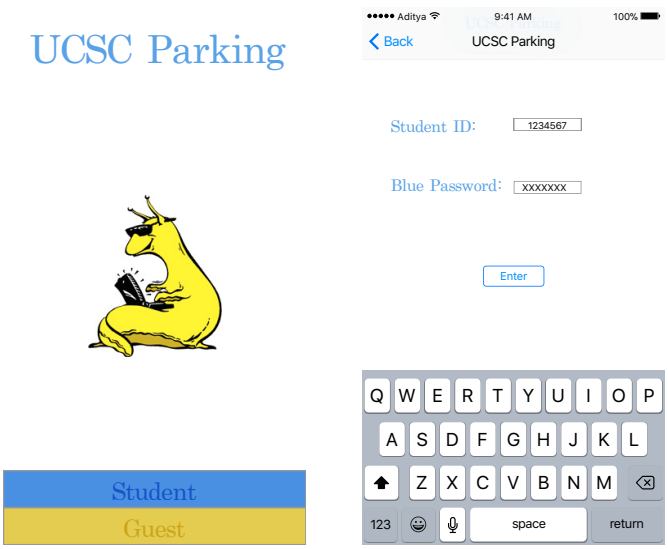
6.8 Mobile Application

The mobile application will be used by the client who wants to park at a spot. The application will give the user information about how many parking spots are open at a particular parking lot, look up what information about them is stored in the cloud database, authenticate themselves with the Bluetooth beacon on the gateway,and have a payment option if the client is a guest.

6.8.1 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



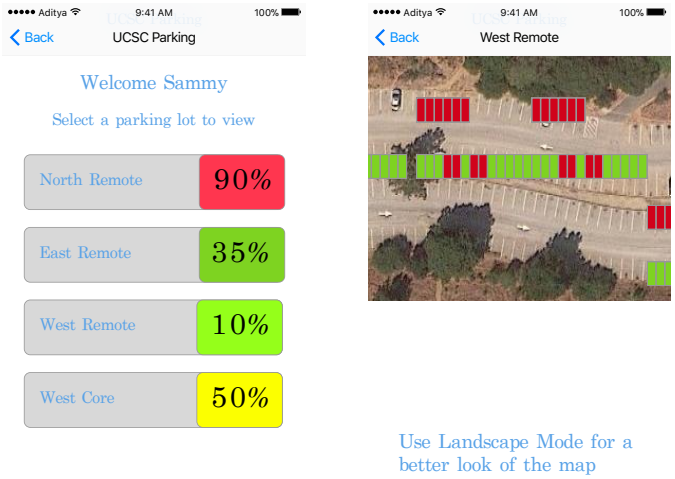
(a) Welcome Page (b) Student Login Screen

Figure 6.13: Screens 1 and 2

The Welcome Page will be the first screen that the user will see when they open the app. It displays a simple opening splash screen and 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 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 authenticate themselves. This information will be sent to the cloud in order for processing.

Screens 3 and 4: Grouped Parking Lots Page and focusing on a specific lot



(a) Parking Lots (b) West Remote

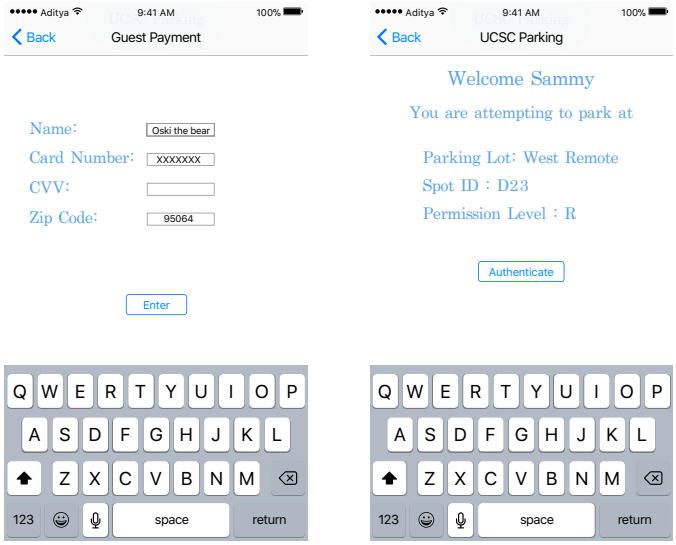
Figure 6.14: Screens 3 and 4

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. The colors correspond to how full a parking lot is, red indicating a high percentage and green indicating a low percentage. Using that spectrum of colors, it ranges from a dark green for 0%, yellow for 50% all the way to dark red for 100%

The right figure shows an example of what the screen would look like if the user were to click the West remote button in the previous screen. A map of west remote would open up and each spot will be represented with a green or red dot to indicate if the spot is vacant or occupied. In our app design, a red would indicate the spot is taken and a green would indicate that it is currently vacant. The top screenshot is displayed in portrait mode, but if the user wants a better view of the parking lots, switching over to landscape will serve as a better option. Here is how it will look in landscape mode:

Figure 6.15: Viewing the West Remote Parking lot in landscape mode

Screens 5 and 6: Payment Screen and Confirmation Screen



(a) Payment Page (b) Authentication

Figure 6.16: Screens 5 and 6

Both screens displayed above will only pop up when a user has parked their car in a spot and have already received a confirmation message back from the cloud. When the sensor box sends the spot ID to the phone, the phone will send the user's information along with the spot ID to the cloud to indicate that the client wants to park their car at that particular spot. The cloud will then check whether the user is allowed to park in that spot and send back a message that will open up one of the two screens:

If the user is a guest, the payment screen will be prompted asking them to enter their payment information to process how much they will have to pay for parking.

If the user is a student, the authentication screen will pop up instead showing all of the information about them that is stored in the cloud to confirm their credentials and parking status. The user can review this information and confirm that they want to park at this spot by pressing the authenticate button. This is how the app design will look for our project.

The screenshots of the app seen above were designed using an application called "Sketch", which allows the user to draw detailed designs of how their app will look like before implementing it on a program. Sketch only allows the user to design apps for an iOs device, and therefore the screenshots indicate how the app would look like for an iPhone version of the app. However, similar architecture will be used to design the app for Android as well.

6.8.2 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.

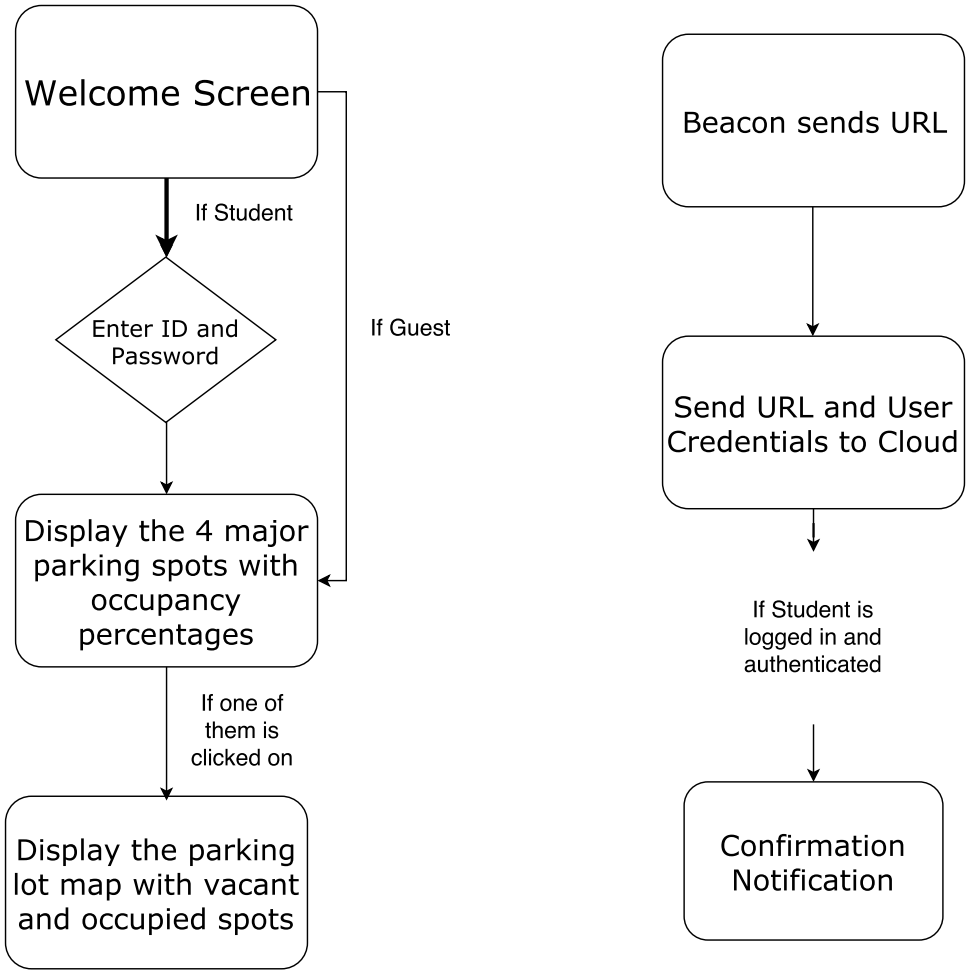


Figure 6.17: Mobile App Flowchart

The flowchart on the right 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 they client presses the guest button, the app will directly go to this page. On the parking lots page, the user can view the parking lot of their choice in more detail if they choose to.

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.

6.8.3 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.

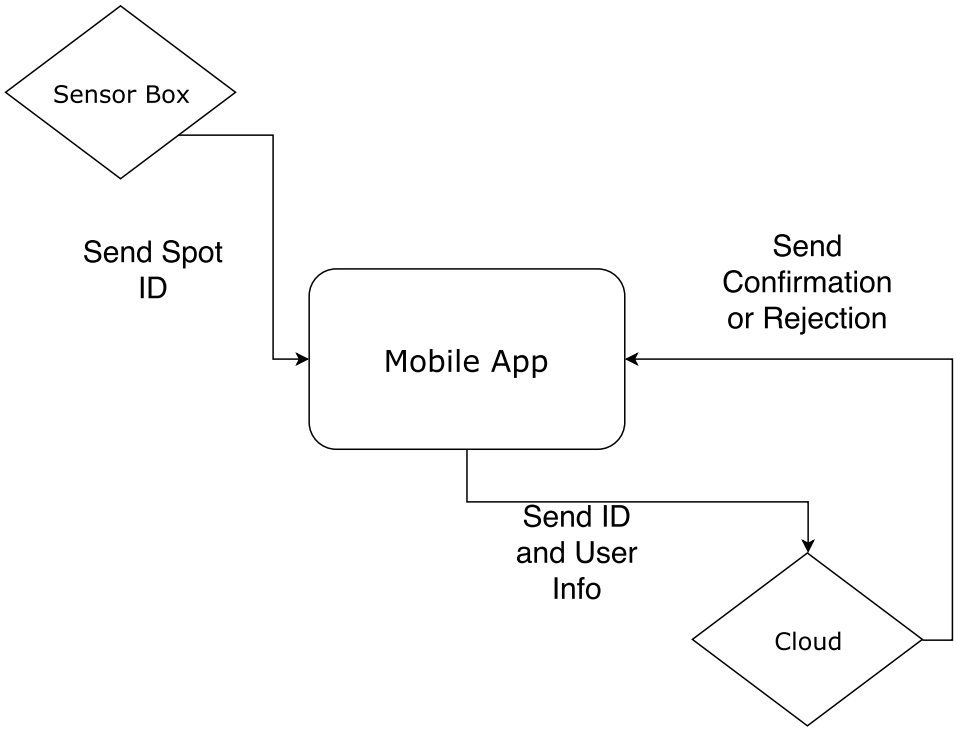


Figure 6.18: Program flow during authentication

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).

6.8.4 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 ReactJS and 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 iBeacon 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 iBeacon protocol and thus personalize it to suit our needs. Android devices utilize the Eddystone protocol, but can be programmed to work with iBeacon as well. To make an iOs device work with Eddystone is extremely hard on the other hand, and therefore, it would be the Android device that will need to import the iBeacon library.

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.

6.8.5 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 parsing 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:

"studentinfo": [
"id": "1234567",
"spasswd": "bananaslug123"

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:

"clientinfo": [
"cname": "Sammy the Slug",
"clot": "West Remote",
"cspotid": "D23",
"cpermission": "R"

One thing to note here is we need to send these messages with a level of encryption since it will contain sensitive data like the person's password or credit card information. By using a commonly used hash function like **MD5**, we can resolve this issue.

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.

6.8.6 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.

6.8.7 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 6.6.4

6.9 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 6.19.

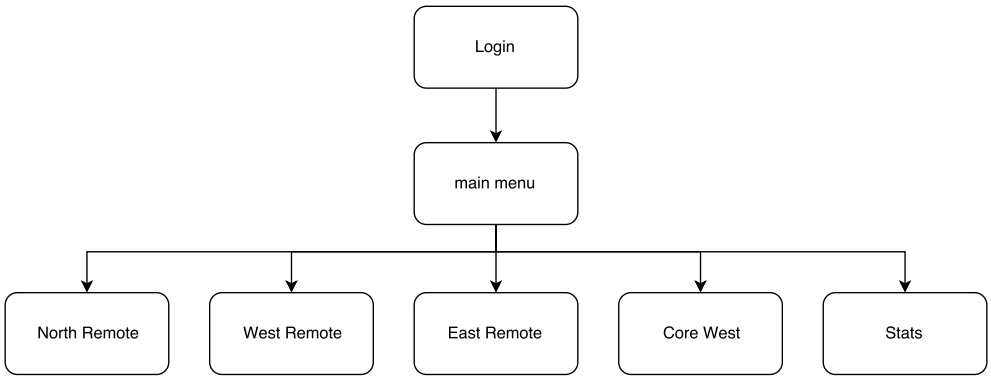


Figure 6.19: General flow of admin app

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.

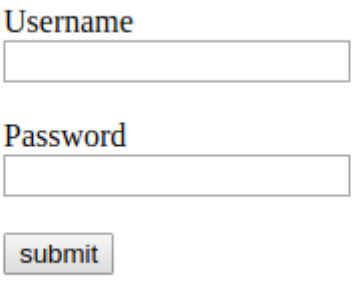


Figure 6.20: The administrator must log in to access the application

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 access 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.

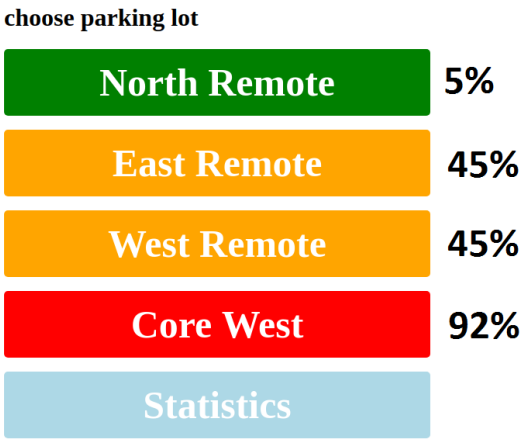


Figure 6.21: Main Menu

What we see in figure 6.21 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. The color of the buttons reflect the fullness of the parking lot. Green is more empty and red is more full. So as we can see above, the North Remote parking lot is fairly empty. The East and West remote lots are beginning to fill up. And finally, the Core West parking lot is completely full. Then the administrator can click on the lot that they wish to examine. 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 details of each spot. Figure 6.22 shows what this page looks like. The administrator app will also have one last page that will be able to add students to the cloud database and generate a QR code to attach to their permit.

What we see here is a Google satellite image of the lot. Laid on top of the image are little green and red rectangles that represent the spots. Green is a free spot, red is an occupied spot. Then when the administrator clicks on a spot, a little bubble will appear with the spot's relevant information. Above we can see that the administrator has clicked on an occupied spot and so it shows that Winston Wang is parked there, He parked at 16:00, he possesses an R permit, and that his license plate is 1ASD234. And at the bottom of the little pop-up there is an option to see the picture that the sensor camera took to get the license plate. On the right of the image is a little blurb about the lot they are looking at. The lot is the West Remote parking lot, the permits currently allowed to park there are A,B,and R permit holders, and the lot is 50% full.

Aside from observing the individual parking space, the administrator is also able to change

Figure 6.22: screen of lot for administrator application

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 between the hours of 12:00 and 14:00. Then when it passes 14:00 it will continue on to default parking permits/times.

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 free parking space, there is an additional prompt to set the parking spot to be reserved. When this happens the page will send a signal to the cloud to 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.

Figure 6.23: Administrator can reserve spots by clicking on unoccupied parking spots

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 6.24 shows what this looks like.

In Figure 6.24, 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 over different time frames. They can observe weekly, monthly, and 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. For example, we have the ability to observe certain days of the week over certain time periods. This means we can observe the usage of the parking lot over all Tuesdays of a couple months. Then fro there the administrator and their team can use this information to update the system and help make the entire process of parking faster and more efficient.

6.10 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.

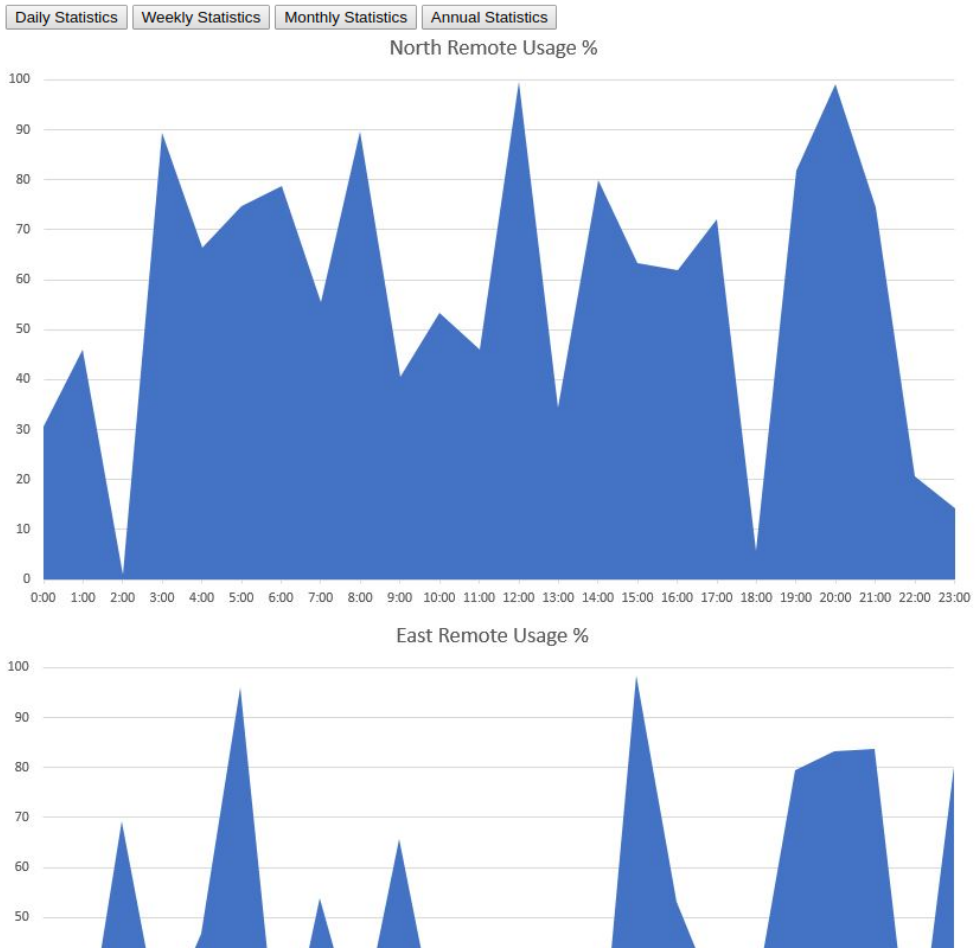


Figure 6.24: Administrator can observe and analyze usage of the parking lots on multiple bases

6.10.1 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 6.6.3

7 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	LANDZO 7 Inch Touch Screen	$86
2	OV7670	$26
2	SR04	$10
Subtotal		$208