B. TECH. PROJECT REPORT

On

Smart Parking - An IoT application for Smart Cities

BY

Hursh Tiwari 130001016
Aditya Kamble 130001019

DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE

November 2016

Smart Parking - An IoT application for Smart Cities

A PROJECT REPORT

Submitted in partial fulfillment of the requirements for the award of the degrees

of
BACHELOR OF TECHNOLOGY
in

COMPUTER SCIENCE AND ENGINEERING

Submitted by:
Hursh Tiwari - 130001016
Aditya Kamble - 130001019

Guided by:
Dr. Abhishek Srivastava
Assistant Professor

INDIAN INSTITUTE OF TECHNOLOGY INDORE
November 2016

CANDIDATE'S DECLARATION

We hereby declare that the project entitled **"Smart Parking - An IoT application for Smart Cities"** submitted in partial fulfillment for the award of the degree of Bachelor of Technology in 'Computer Science and Engineering' completed under the supervision of **Dr. Abhishek Srivastava, Assistant Professor,** IIT Indore is an authentic work.

Further, we declare that we have not submitted this work for the award of any other degree elsewhere.

Aditya Kamble **Hursh Tiwari**

CERTIFICATE by BTP Guide(s)

It is certified that the above statement made by the students is correct to the best of my/our knowledge.

Dr. Abhishek Srivastava,

Assistant Professor, CSE,

Indian Institute of Technology, Indore

Preface

This report on “Smart Parking - An IoT application for Smart Cities" is prepared under the guidance of Dr. Abhishek Srivastava .

Through this report, we have tried to give a detailed design and steps followed during implementation of an innovative Car Parking system that will prove to be a significant contribution to the Smart Cities initiative of Government of India.

We have tried to the best of our abilities and knowledge to overcome the limitations involved in this design and explain the content in a lucid manner. We have also added workflow diagrams and screenshots to make it more illustrative.

Hursh Tiwari
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

Aditya Kamble
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

Acknowledgements

We wish to thank Dr. Abhishek Srivastava for his kind support and valuable guidance.

It is their help and support, due to which we were able to complete the design and technical report.

Without their support this report would not have been possible.

Hursh Tiwari
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

Aditya Kamble
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

Abstract

Ubiquitous sensing, enabled by various sensors and similar technologies cuts across many areas of modern day living. This offers the ability to measure, infer and understand environmental indicators, from delicate ecologies and natural resources to urban environments. The proliferation of these devices in a communicating data creates the Internet of Things (IoT), wherein, sensors and actuators blend seamlessly with the environment around us, and the information is shared across platforms in order to satisfy the requirement of users. Fuelled by the recent adaptation of a variety of enabling device technologies such as RFID tags and readers, near field communication (NFC) devices and embedded sensor and actuator nodes, the IoT has stepped out of its infancy and is the the next revolutionary technology in transforming the Internet into a fully integrated Future Internet.

Many efforts are centered around creating large scale networks of "smart things" found in the physical world (e.g., wireless sensor and actuator networks, embedded devices, tagged objects). Rather than exposing real-world data and functionality through proprietary and tightly-coupled systems, a newly proposed system is to make them an integral part of the Web. This is Web of Things.

The aim of Internet of Things is to expand the boundary of the current virtual Internet into the physical space by connecting new devices accessible over the Internet. These devices can then be used in a variety of ways as efficient solutions to real life problems. A Web of Things (WoT) abstraction has been proposed in the recent past as an approach to come up with novel IoT solutions.

Table of Contents

Chapter 1

Introduction

One of the major concern of people living in major cities today is that of “Parking”. While planning to go out to there are many concerns. Firstly, people are not sure whether they will get a parking space for their car there. So they take a chance and take their own car or just hire a cab. People have difficulty finding a reliable parking space near them and sometimes unknowingly they end up parking in a “No parking” area. This leads to towing of their car. Moreover, some parkings are just not safe enough. There have been cases where returning to the car people find out that the tyre has been punctured or glass has been cracked or scratches on the car body. At the worst, they don’t find their car in place. In large and somewhat reliable parking areas like those of malls, people get lost while finding their parked car.

Govt of India launched the Smart Cities Initiative in 2015 to create citizen-friendly and sustainable urban areas. As part of this Initiative, the Govt of India has recently identified parking to be a major pain point in the large scale traffic management of these Smart Cities.

To tackle the above mentioned problems there has to be centralised entity that acts as an interface between the users and each and every parking spot in the city. Through our project we present the workings of such an interface. Starting from giving the status of each spot, this system allows users to book or reserve them. All this at the fingertips of the user. Even during the parking, the car is completely monitored and the user is assured that the car is safe.

We now move over to the concept and design part involved in developing this project.

Chapter 2

Concept and Literature Survey

2.1 Concept

As we had clearly described in the motive earlier, we need something placed at every parking spot in the city which should be connected to the centralised system showing status of that particular spot seamlessly. The “Internet of Things” described next provides a viable solution to this kind of problem.

2.1.1 Internet of Things

The Internet of Things is a system of physical objects that can be discovered, monitored, controlled, or interacted with by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider internet. In the last few years, the Internet of Things has become one of the most promising and exciting developments in technology and business.The IoT field promises simplification of human lives by providing large scale automated solutions to simple procedural tasks that are completed manually at present.

However,the current IoT projects are devised to be very use case specific. Hence, the re-use and expansion of such IoT solutions is a challenging and cost-ineffective task.This led to the entry of a idea called the “Web of Things”

2.1.2 Web of Things

Currently a lot of research and debate exists in the world on the idea of usage of a common protocol for connecting Things to the Internet. In an effort to use already existing infrastructure of the World Wide Web as opposed to designing an entire new protocol stack or using a specific protocol for creating ubiquitous IoT solutions, the Web of Things abstraction was proposed. The idea for the Web of Things is simple. Each Thing connected to the internet should expose a web-API. Simply put all attributes of a physical thing must be accessible-modifiable (and thereby usable) as Hypermedia over the internet.

A common practice involved in development of new services and applications over the Internet is to expose them through RESTful API’s. This concept is dealt with next.

2.1.3 REST (Representational State Transfer):

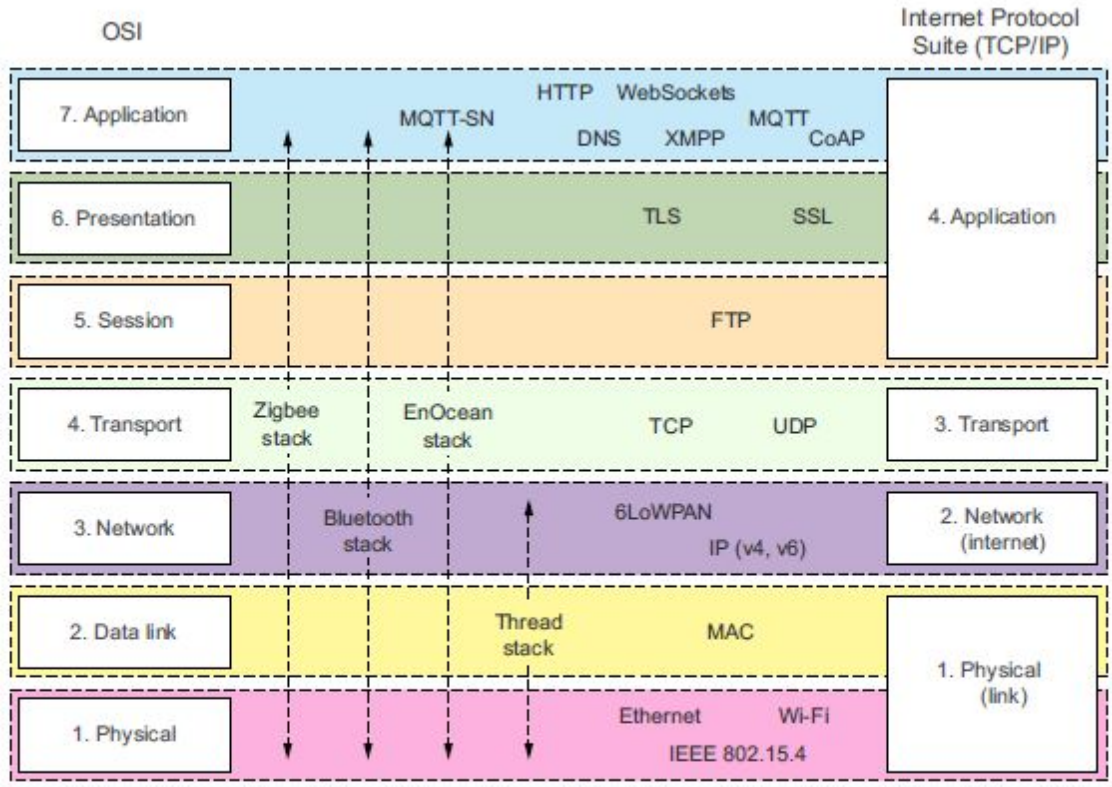
The goal of the Web of Things is to make it possible for any physical object to be accessed via the same uniform interface as the rest of the web. The uniform interface of the web is based on these four principles:

- Addressable resources: A resource is any concept or piece of data in an application that needs to be referenced or used. Every resource must have a unique identifier and should be addressable using a unique referencing mechanism. On the web, this is done by assigning every resource a unique URL.
- Client-Server Communication model: Interactions between components are based on the request-response pattern, where a client sends a request to a server and gets back a response. This maximizes decoupling between components because clients don't need to know anything about the implementation of the server, only how to send the request to get the data they want. Likewise, servers don't need to know about the state of clients or how that data will be used.
- Stateless: The client context and state should be kept only on the client, not on the server.Because each request to the server should contain the client state, visibility (monitoring and debugging of the server), robustness (recovering from network or application failures), and scalability are improved.
- Layered System: Uniform interfaces make it easy to design a layered system, which means that several intermediate components can hide what's behind them. Layered systems make it possible to use intermediary servers to further improve scalability and response times.Another benefit of layered systems is that it enables encapsulation of legacy protocols and systems–for example, gateways to proprietary protocols–which makes it simpler to enforce various security policies.

2.2 Literature Survey

2.2.1 Protocols

We defined the Internet of Things as "a system of physical objects that can be discovered, monitored, controlled, or interacted with by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider internet." But what does it mean for a Thing to be "connected to the internet?" It means that you can interact with it using the communication protocols of the Internet Protocol Stack shown in figure



The OSI model (left) compared with the Internet Protocol Suite (aka TCP/IP, right), along with examples of some of the most relevant protocols and protocol stacks for the IoT (center)

The choice of protocol to be used depends on many considerations and tradeoffs:

1. Power consumption: Some protocols like WiFi are power consuming because of the resources they use while others like LPWAN are less power consuming.
2. Cost: It determines which type of embedded device you can build into your Thing. In our case, considering lakhs of nodes across the city we chose to transmit the data between sensors and Pi through wires. Since, if every node was made wireless just to transfer data in terms of 0 and 1, the project would not have been cost efficient.
3. Range and network topology: How far a Thing will be from a gateway or from other Things. This is tightly coupled with how much power your device can use.
4. Bandwidth, latency and sensing: Do devices only need to send data to other devices or to the cloud? How often will you send messages; that is, will the device send data a few times per minute? Or will it sleep most of the time and send messages only a few times per day? Also, will you require sending commands to the device (actuation), and if so, how much latency can your application tolerate? Finally, how large will those messages be?
5. Internet integration and openness: Is it an open or proprietary standard? How open and accessible are the specifications? How well supported is the protocol in the real world? How integrated with the internet protocols stack is that protocol? If a protocol stack doesn't provide ways of translating

to internet protocols (IP, TCP, or UDP) easily, it might be fine in closed networks (industrial machines in a factory, for example), but not if those devices need to be accessible through the internet.

2.2.2 HTTP GET REQUEST

GET is a read-only operation. It's both a safe and idempotent operation. Safe means that invoking a GET doesn't change the state of the server at all (read-only). Idempotent means that no matter how many times you apply the operation, it won't have an effect on the resource state. Reading an HTML document with an HTTP GET request once or 10 times won't change the resource state.

2.2.3 HTTP POST REQUEST

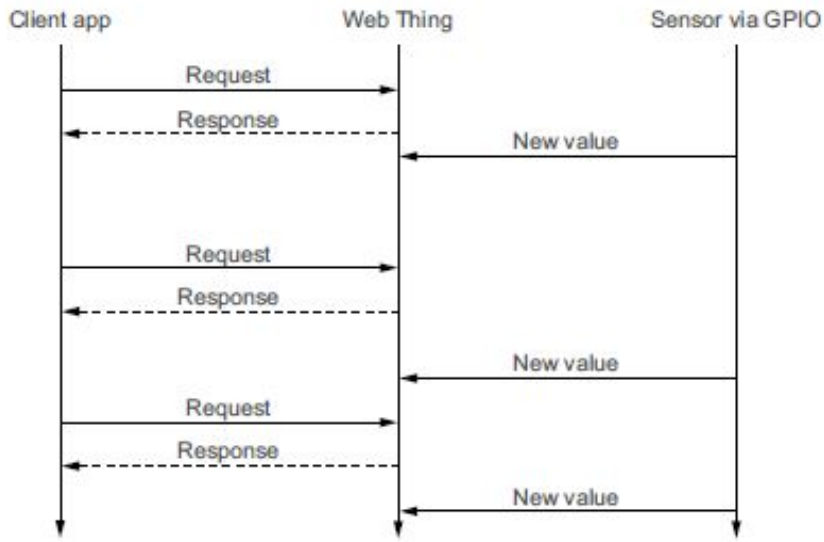
POST is both a non-idempotent and unsafe operation of HTTP, which means it not only will change the server status but also will have a different result each time it's called. POST should be used only to create a new instance of something that doesn't have its own URL yet, such as a new user in a system or bank account.

2.2.4 Hypermedia as the Engine of Application State (HATEOAS)

Hypermedia is the idea of using links as connections between related ideas.The application state refers to a step in a process or workflow, similar to a state machine, and REST requires the engine of application state to be hypermedia driven. It means that each possible state of your device or application needs to be a RESTful resource with its own unique URL, where any client can retrieve a representation of the current state and also the possible transitions to other states. Resource state, such as the status of an LED, is kept on the server and each request is answered with a representation of the current state and with the necessary information on how to change the resource state, such as turn off the LED or open the garage door. In other words, applications can be stateful as long as client state is not kept on the server and state changes within an application happen by following links, which meets the self-contained-messages constraint. Links are very important in the Web of Things because they enable clients to discover related resources, either by browsing in the case of a human user following links on pages, or by crawling in the case of a machine. In short, linking resources allows them to be dynamically discovered and rearranged without having to keep a sitemap somewhere.

2.2.5 WebSockets

Via HTTP, clients always initiate the communication with a server by sending requests and expecting a response in return; this is known as request-response communication. In the Web of Things this pattern works well when the clients only need to send requests to a Thing. This is the case, for example, when a mobile application wants to retrieve the value of a sensor reading or when a web application is used to unlock a door. But unfortunately, this doesn't match event-driven use cases where events must be communicated (pushed) to the clients as they happen. WebSocket is an application protocol that extends the request-response model of HTTP.

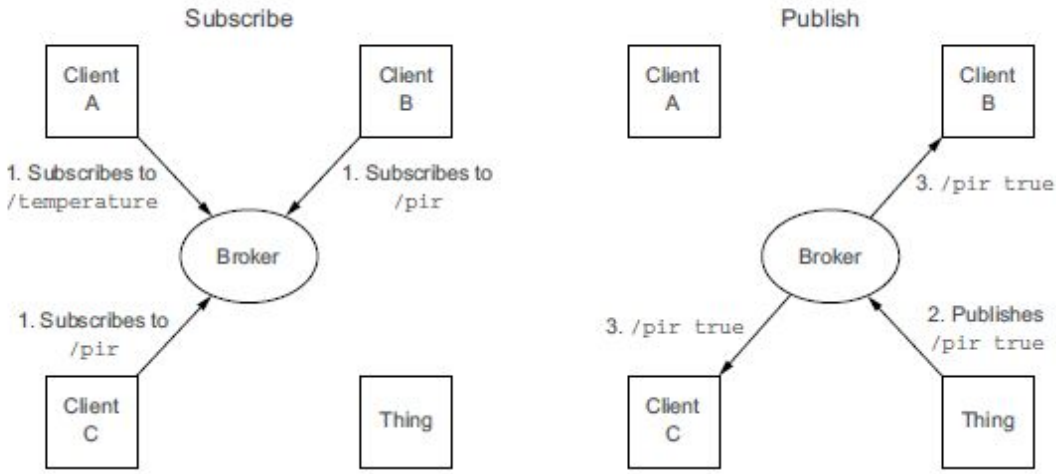


The idea is that clients can request updates periodically from a web Thing by sending a GET request to the Thing on a regular basis. Although near real-time behavior can be simulated by a client sending the same request continuously—for example, every second— this approach is inefficient for most applications because it consumes unnecessary bandwidth and processor time. Most of the requests will end up with empty responses (304 Not Modified) or with the same response as long as the value observed remains unchanged. This is suboptimal for two reasons. First, it generates a great number of HTTP calls, and a large part of these calls are void. Because reducing the number of HTTP calls to the minimum is key in scaling web applications, this model doesn't scale well when the number of clients increases. Second, a large amount of HTTP calls is a problem for battery-powered devices where only strictly necessary data should be sent.

2.2.5.1 Publish/Subscribe

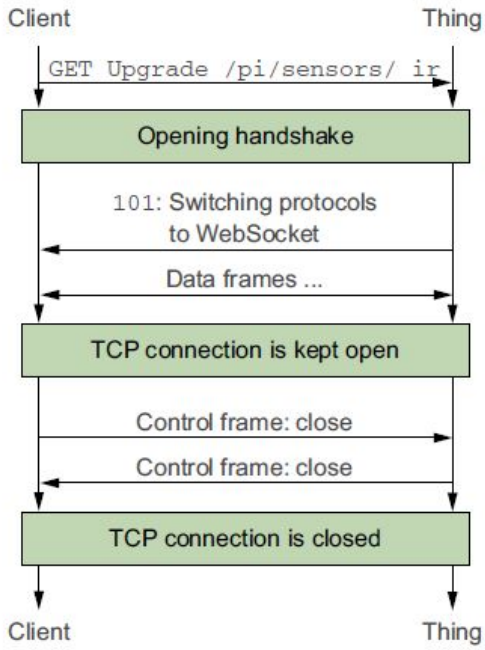
Interactive and reactive applications for the Web of Things require a simple and flexible mechanism to send events or receive notifications. What's really needed on top of the request-response pattern is a model called publish/subscribe (pub/sub) that allows further decoupling between data consumers

(subscribers) and producers (publishers). Publishers send messages to a central server, called a broker, that handles the routing and distribution of the messages to the various subscribers, depending on the type or content of messages. The simplest analogy is a chat room—some are public, and some are private. Sometimes you chat with only one person, and sometimes thousands. For devices, it would be the same thing. A publisher can send notifications into a topic (think chat room). Interested consumers can subscribe to one or several channels to receive all the notifications pushed by producers in that channel. Topics in pub/sub protocols are usually specified as arbitrary strings so it's easy for us to map the REST resources of our web Things to pub/sub topics.



2.2.6 Websocket Protocol handshake

WebSockets enables a full-duplex communication channel over a single TCP connection. In plain English, this means that it creates a permanent link between the client and the server that both the client and the server can use to send messages to each other. Unlike techniques we've seen before, such as Comet, WebSocket is standard and opens a TCP socket. This means it doesn't need to encapsulate custom, non-web content in HTTP messages or keep the connection artificially alive as is needed with Comet implementations. A WebSocket connection is initialized, creating a handshake, as networking nerds would put it, in three steps, as shown in figure 6.9. The first step is to send an HTTP call to the server with a special header asking for the protocol to be upgraded to WebSockets. If the web server supports WebSockets, it will reply with a 101 Switching Protocols status code, acknowledging the opening of a full-duplex TCP socket.

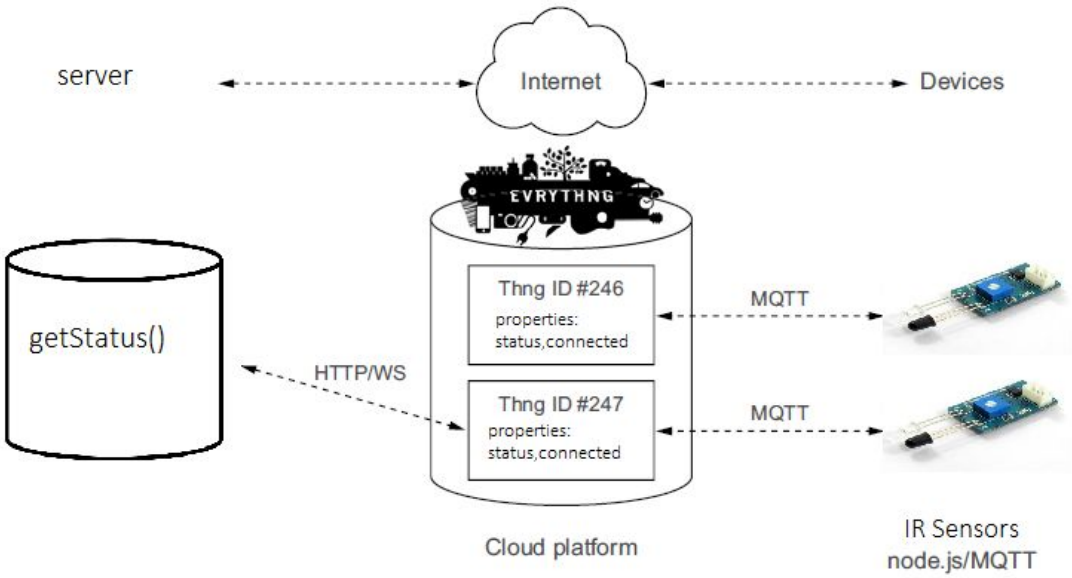


What's really interesting with WebSockets for the Web of Things is that they use standard internet and web technologies. Because they open a TCP connection over port 80, WebSockets aren't blocked by firewalls and can traverse proxies. Then, because they work in the browser and are bootstrapped via HTTP, they let us use a lot of the principles we looked at when exploring HTTP and REST. First, the hierarchical structure of Things and their resources as URLs can be reused as-is for WebSockets. As you saw in listing 6.14, we can subscribe to events for a Thing's resource by using its corresponding URL and asking for a protocol upgrade to WebSockets.

Moreover, Web- Sockets do not dictate the format of messages that are sent back and forth. This means we can happily use JSON and give messages the structure and semantics we'll work on in chapter 8. Moreover, because WebSockets consist of an initial handshake followed by basic message framing layered over TCP, they can be directly implemented on many platforms supporting TCP/IP—not just web browsers. They can also be used to wrap several other internet-compatible protocols to make them web-compatible. One example is MQTT, a well-known pub/sub protocol for the IoT that can be integrated to the web of browsers via WebSockets.

The permanent link created by WebSocket communication is interesting in an Internet of Things context, especially when considering applications wanting to observe—or subscribe to—real-world properties such as environmental sensors. Finally, WebSockets offer all of these benefits with significantly reduced bandwidth consumption when compared to HTTP polling, for example. The drawback, however, is that

keeping a TCP connection permanently open can lead to an increase in battery consumption and is harder to scale than HTTP on the server side.



2.2.7 MQTT (Message Queue Telemetry Transport):

The Message Queuing Telemetry Transport (MQTT) is an Application layer protocol which is a lightweight messaging protocol built on top of TCP/IP that allows constrained devices with limited bandwidth to talk to each other.Over the years, MQTT has become an important Application layer protocol for machine-to-machine (M2M) communication.MQTT clients subscribe to a topic of interest and receive notifications whenever a new message for this topic is published. The publisher and subscribers of messages don't speak to each other directly but through an intermediate called a broker.

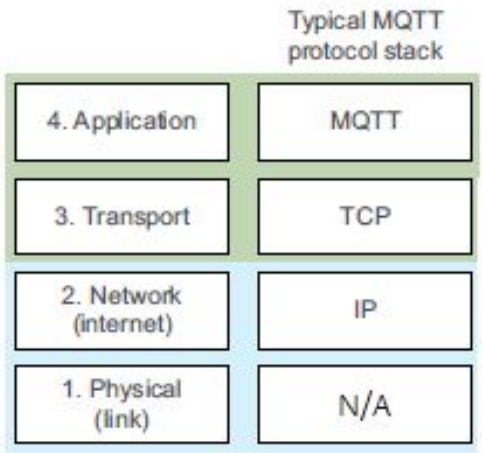
2.2.7.1 QUALITY OF SERVICE

An interesting feature of MQTT is that it offers three levels of quality of service (QoS) to guarantee what a client application can expect when it comes to the delivery of messages. Note that this is complementary to the Transport layer delivery guarantee of TCP you saw before because it relates to delivery between subscribers and publishers at the Application layer. Clients can request the following QoS levels from brokers that support them:

■ QoS 0: fire and forget—A published message might be delivered to the subscribers, but this is not guaranteed. Receivers won't acknowledge a message and brokers won't store or redeliver them.

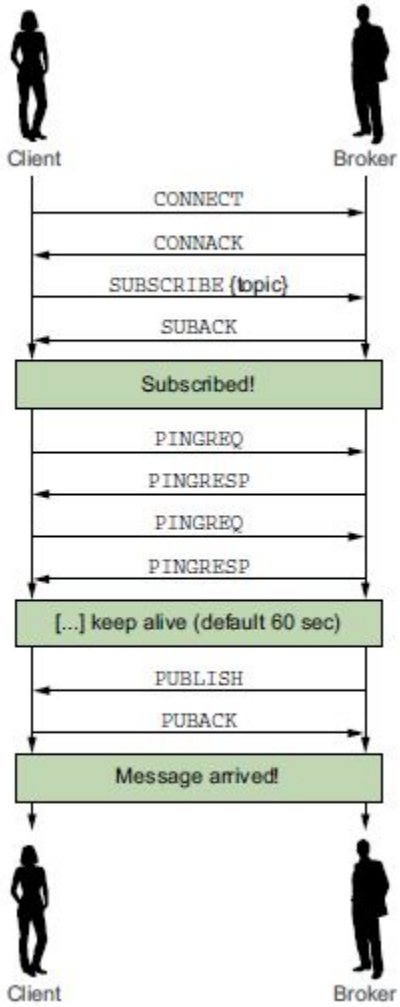
■ QoS 1: deliver at least once—A published message will be delivered at least once to the subscribers. This means that if a subscriber temporarily disconnects, it will receive the message as soon as it reconnects. An example of the messages required for such a connection is shown in figure 5.11.

■ QoS 2: deliver exactly once—A published message will be delivered once—and only once—to each subscriber.



2.2.7.2 PERSISTENT CONNECTIONS

As shown in figure, MQTT is built on top of TCP and IP. This means that all the publication and subscription happens via TPC/IP. But one interesting aspect of MQTT is that it keeps the connection between a client and a broker open for as long as it can. To maintain this open connection, the client sends regular ping requests (PINGREQ) to the server, as shown in figure ___, where a client is connected to a broker with QoS 1.



2.2.7.3 SECURITY AND ENCRYPTION

Security in MQTT works via Transport Layer Security (TLS), the successor of SSL that's used to encrypt traffic on the web. On top of encryption, the broker can request a username and password to identify the clients. freely available to date is the Really Small Message Broker provided via the Eclipse foundation.

2.3 SUMMARY

With these concepts we now move on to our design and workflow.

Chapter 3

Design and Workflow

3.1 Design

The parking problem here is broken down into small simpler problems. There solution designs are correspondingly described along the way . We start with the parking terminologies used throughout the report.

3.1.0 Terminology

The whole parking domain of the city will be divided into two major parts:

1. **Parking Spot :** Parking spot is the basic entity in this domain which represents a place where only one vehicle can be parked.
2. **Parking Area :** Parking area, as the name suggests, is an area made up of various parking spots. It can be identified as a parking area of an office or a mall or a college.

Client: Refers to any end user that uses our application to carry out the activities specified in section 2.2.2-2.2.6 of this report.

Server : Refers to our backend cloud server that is hosted on the online cloud service Heroku. This server performs majority of the business logic involved in developing an automated parking application.

Evrythng Cloud: Refers to the cloud Platform provided by Evrythng. This acts as our main middleware in getting connected to the physical world.

Sensor: Refers to the IR sensor used for obstacle detection at parking spots.

3.1.1 Sub-Problem 1: Selecting parking areas

To the client a major headache in any city is knowing the areas where he can park safely. To get the parking areas in the location that a client wishes to know the following design is proposed.

Server holds a list of parking areas in its database. On request by client a list of nearby parking areas should be made available to him. This constitutes one atomic transaction in the request-response cycle between client and server.

3.1.2 Sub-Problem 2: Selecting parking spot

When a client decides on an area where he would want to park his car, the client must know the current free spots in the particular area. This is achieved as follows:

Client sends a request to the server specifying a particular area where he wishes to know the status of parking spots. The server communicates to sensors in a particular area querying their current status. On getting responses from the sensor the server responds to the client with a list of free parking spots.

3.1.3 Sub-Problem 3: Notification to user entry and exit at parking spot

To make the client aware of his car in the parking lot, the server must communicate to a client at proper states of the parking process with appropriate messages. These states and corresponding messages are enumerated below:

1. Client enters the parking spot: The server must inform the client when he enters the parking spot to make the client aware that his parking spot is now occupied. This is important because the client is getting this notification it is now evident to the client that he is present at the right parking spot that is being securely monitored by a intelligent parking system.
2. Client exits the parking spot: The server must inform the client when he exits the parking spots to make the client aware that his parking spot has now been vacated. This is important for 2 reasons:
    a. If the car is moved through illegal means(i.e. being stolen) he client now knows that such an activity is taking place. This increases the safety profile for any end user
    b. Billing for parking is now correctly and efficiently possible since we can know the exact usage of the parking spot since its monitored by a machine.

To design for such a behaviour the following method is proposed:

The client must specify a particular spot of a particular area in his request to book a spot. The server takes this request and ,if the spot is free, books the spot for a buffer time for this particular user. If the user reaches this spot in this buffer time the server notifies the user that he has entered his parking spot and the parking time has begun. Upon exiting the spot the server again sends a notification to the client that the spot that the user had parked on has now been vacated and his parking time is over.

3.1.4 Subproblem 4: Server to know the data of individual spots efficiently

Since each unique booking requires the server to continuously monitor the corresponding spot for changes, efficiently performing this operation is of prime importance to enabling this application to expand at the scale where it can be used to serve for cities. The following design is proposed:

For each request that a server encounters where it needs to book and monitor a particular spot it must add the request to a current queue of unserved requests and respond instantaneously to the client with the buffer time. This end the client-server request-response cycle. For each of the unserved requests the client must open individual connections to sensors that are lightweight and asynchronous so that only on communication from the sensors the server will take any further actions. When the sensor sends a status update to the server the server searches which client has booked for this particular sensor/spot and correspondingly notifies the client of updated sensor status.

3.2 Workflow

This section tries to explain the whole workflow involved in usage of the SmartParking system.

User has the SmartParking android application installed on the phone. The first thing that user has to do is sign in using his existing user account. Once this is done, an ID token obtained from Google is stored on the phone. In all of the following requests, this token is added as an header. The server first checks this token to verify the user and only if the user is genuine, requests will be served else, "401:Unauthorized".

On successful Sign-in, the user is redirected to a dashboard. User has many options to choose from here. First, he can locate himself and various parking areas across the city on the map. This is done by sending a GET request to the server with gps coordinates embedded in the url. http://salty-anchorage-75968.herokuapp.com/api/area?longitude=*user_long_coordinate*&latitude=*user_lat_coordinate*. If the user is genuine, database is searched for parking areas within 20 kilometers and the list is returned as an JSON array. This same request is again sent when user wishes to reserve a spot since first he has to be provided with the list of parking areas.

After selecting an area, a second type of request is sent to the server to get a list of free spots in an area. This time it is a POST request. http://salty-anchorage-75968.herokuapp.com/api/area?area=*area_id*. The array of spot IDs corresponding to this area_id is traversed on the database and each spot is checked if it is reserved or its status is 1. If not, the spot ID is added to response array. To check if it is reserved, simply the spot with the spot ID is located in the database and checked if the "reserved" field is set to 1. To check if the sensor status of that spot is 1, the thng ID and thng key of the spot is retrieved from the database and the status is requested from the Evrythng cloud server. This same request is sent when user selects an option to "book" provided that there is a parking area nearby and then the area selected ID is of that area.

After receiving the array of IDs of free spots, user chooses one from a dropdown menu and taps book/reserve. This leads to a third type of request: http://salty-anchorage-75968.herokuapp.com/api/spot?spot=*spot_id*&type=*book/buffer_time*&key=*firebase_client_id*

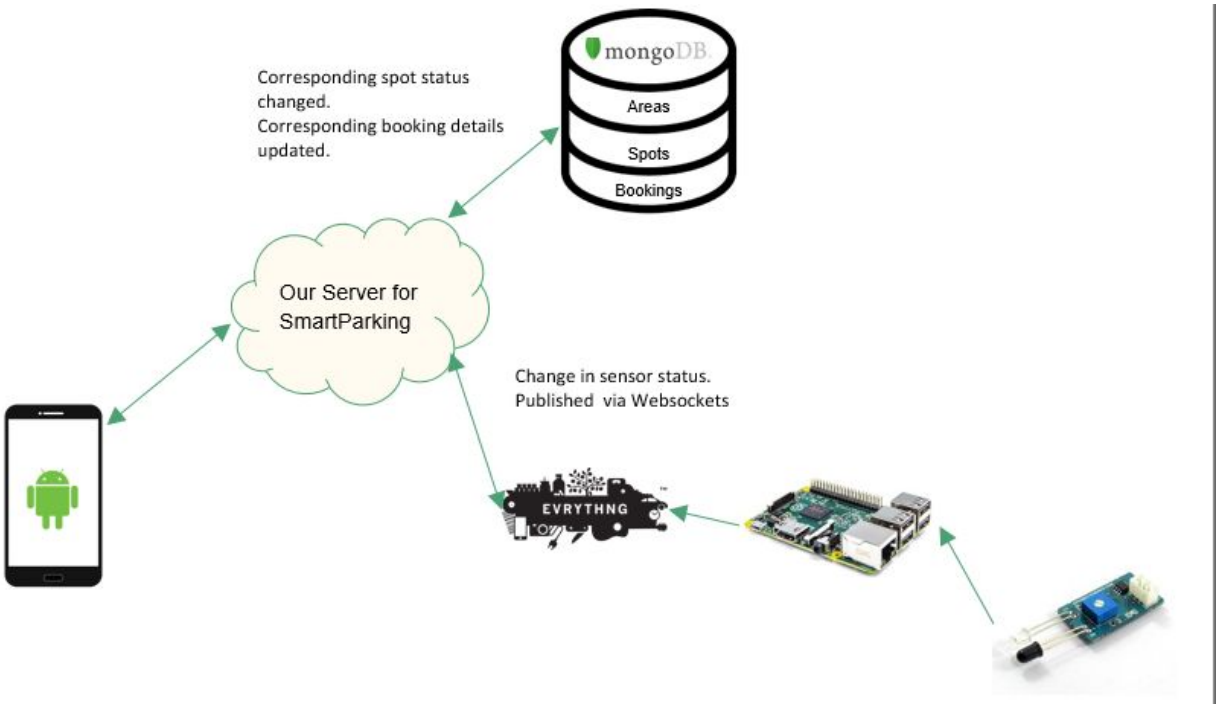
This is also a POST request. In case of a booking, type value will be "book" else, in case of a reservation, it will be the time to be allotted to the user depending on his distance from parking area. Parameter "key" is the client ID obtained from firebase server. This key will be further used to send push notifications related to this booking.

As a response to this request, buffer time is sent to the user, the reserved field of spot is set to 1, a booking is made in the booking table and a websocket is opened to Evrythng cloud to detect a change in corresponding spot sensor. There are 2 scenarios from here on:

If change in status is not detected within time, the booking is expired, it is deleted from the table, reserved field is set to 0 again, user is notified to book again and the websocket is closed.

If change is detected within time, the start time of the booking is updated in the table, user is notified that booking has started and the same websocket is again configured to detect a change in value from 1 to 0 i.e. to detect end of parking.

Once the value of the sensor changes from 1 to 0, the user is notified that booking time is over and presented with the total time, the end time of the booking is updated, the "reserved" field of the spot is set to 0 and the websocket is closed.



3.3 Summary

In this system, each parking spot in the city will be monitored by an IR sensor. These IR sensors transmit the value of 0, if no obstacle is in front of it and 1, if there is an obstacle. At an intermediate level there will be a Raspberry Pi which will be acting as a gateway. First, it will be reading values from the connected sensors and then pass it over to the cloud. Further, the data received on the cloud will be sent over to the server from where it will be served only to the authorized users requesting it.

Considering the wide impact of this project, there will be lakhs of sensors placed all over the city. Trying to preserve the essence of IoT, if each of this sensor is to be made wireless then it will have to be attached with a wireless module. The problem is that the cost of a single wireless module is almost thrice the cost of sensor itself and if it is followed, the project will turn out to be very cost inefficient. Keeping this in view, as of now, we have decided to proceed with the wired connections between sensors and Raspberry Pi. A Raspberry Pi has several GPIO (General-purpose input/output) pins. Through these pins, the Pi will receive the data from the sensors in case there is a change.

The proposed idea is so scalable that the system is not restricted to just IR sensors. Any accurate, advanced and complex sensor can be used which can monitor a parking spot and send over the data to the cloud in form of 0 and 1.

Chapter 4

Implementation

4.1 Software

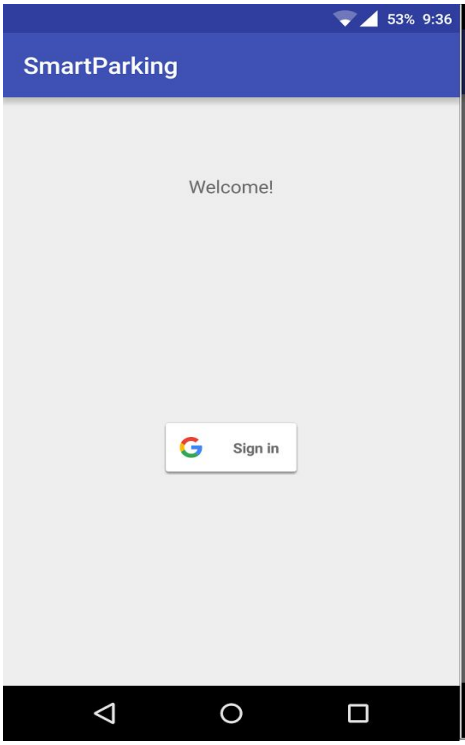
A Front-end development

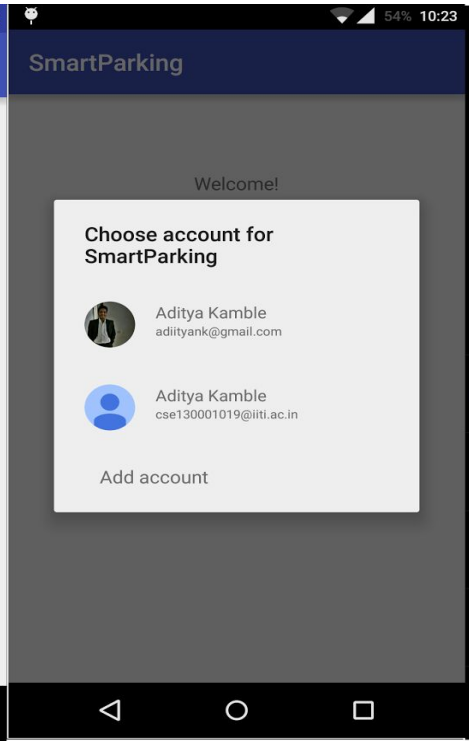
To ensure that the main purpose of this application is served, the end application should be such that it can be accessed from anywhere and at any time. In this era of Smartphones, nothing else is a better omnipresent and user friendly option than an android app. So the front end basically involves android app development.

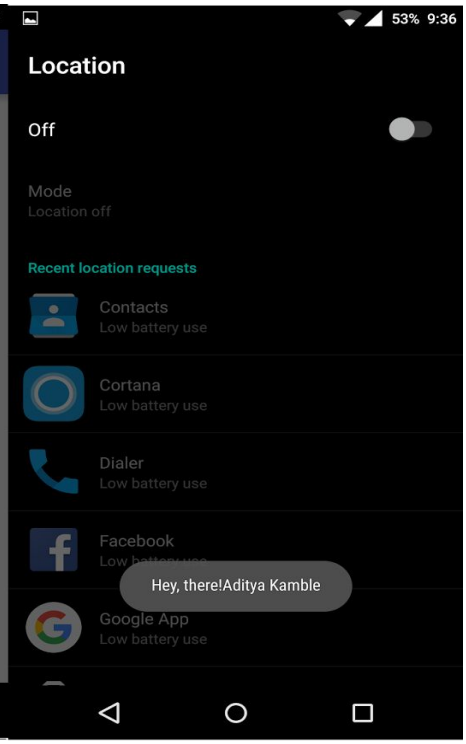
The whole app is divided into multiple activities. Starting from the Sign-in Activity, after successful login the user is directed to the Dashboard activity. From here, the next activity depends on the option selected. If the user opts for booking a slot Book activity is opened else if user opts for reserving a slot Reserve activity is opened. After selecting a free spot, user is directed to a countdown timer displaying the time left for the user to reach the spot in time in order to avoid the cancellation.

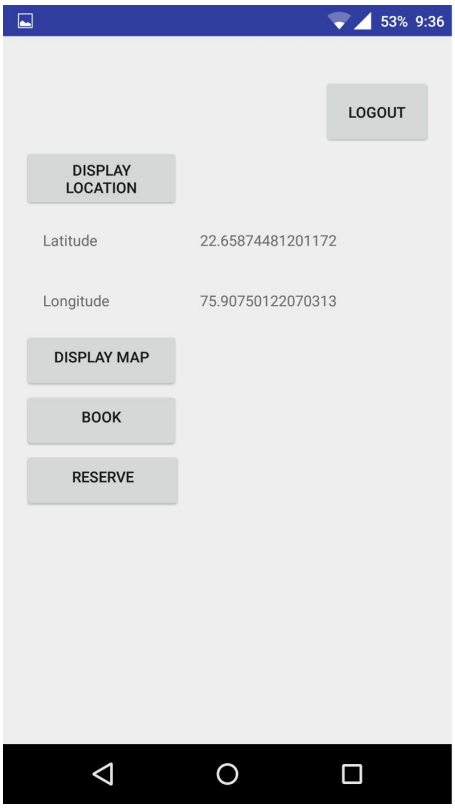
For sign-in, Google OAuth 2.0 is used. To get an user's current location accurately, the app makes use of GPS as well as the Internet. Hence, these are the additional permissions that user must grant the application in order to ensure the smooth functioning. To make users familiar with various parking locations present in the city and to navigate them there from their current location we use the Google maps API. To send POST and GET requests to the server we use the Volley Library. To receive updates from the server in form of push notifications we use the Firebase Cloud Messaging service.

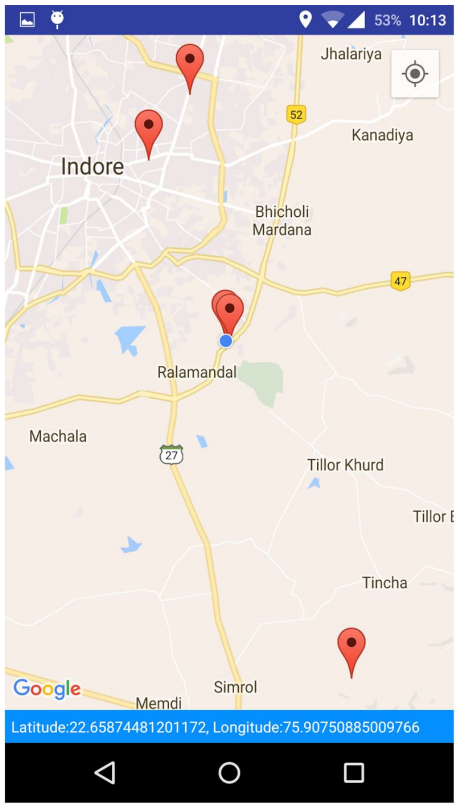
Screenshots:

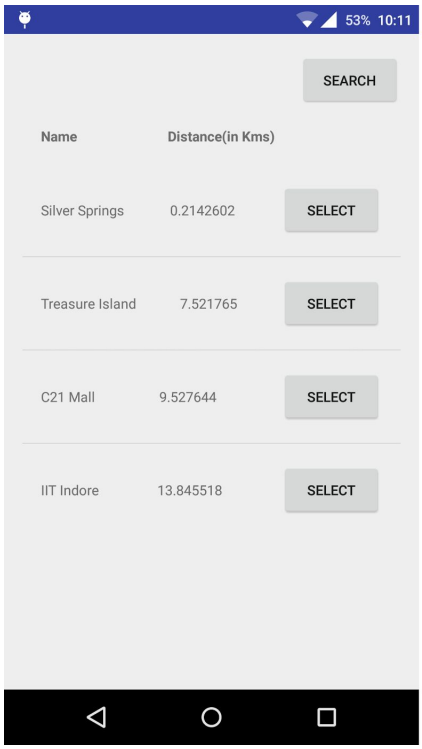


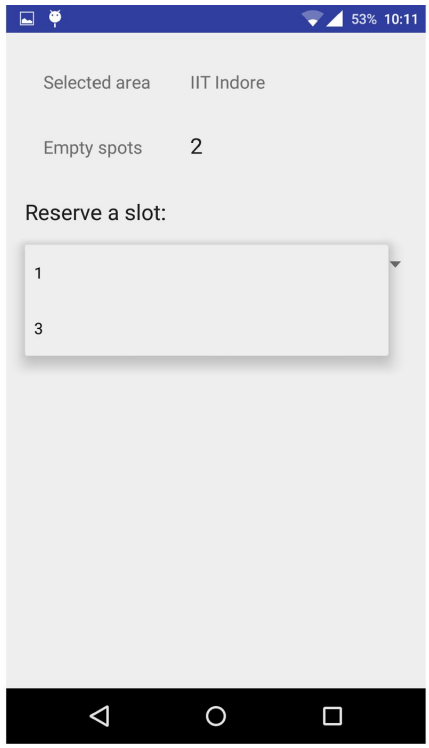


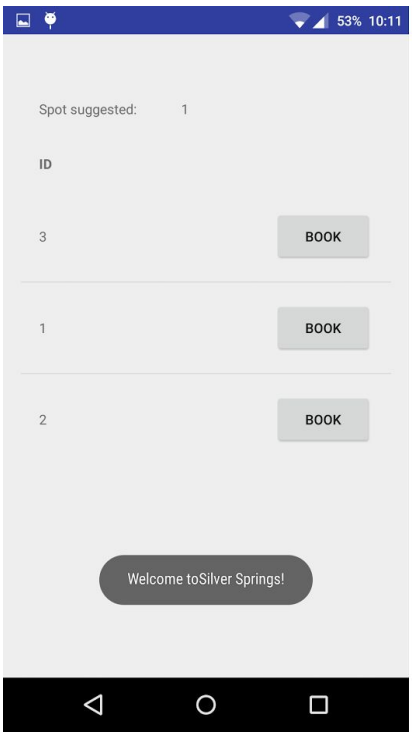




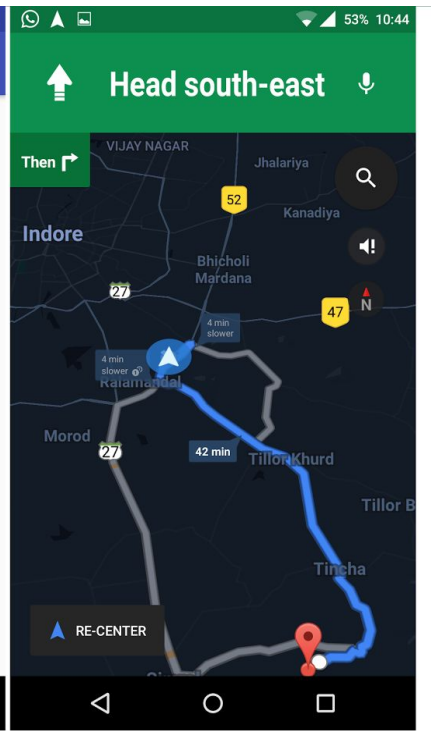


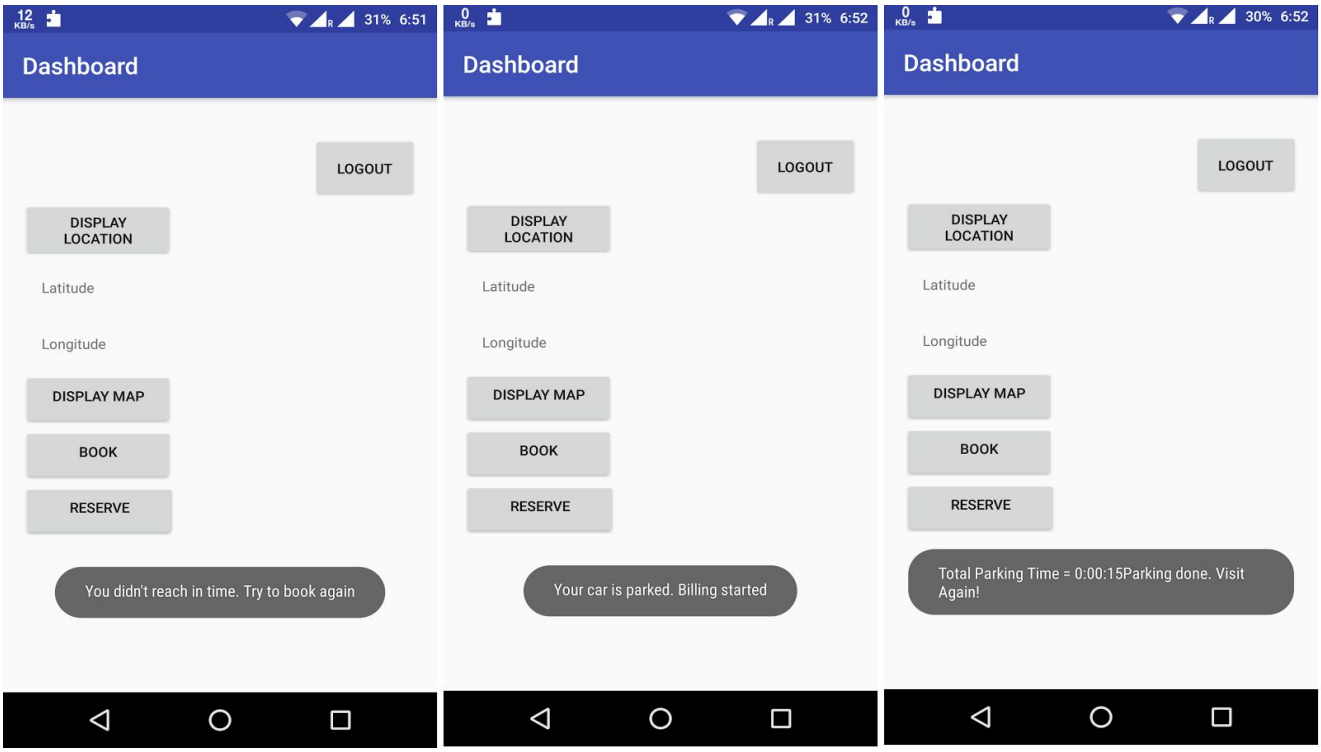

53% 10:11
SEARCH
Name
Distance(in Kms)
Silver Springs
0.2142602
SELECT
Treasure Island
7.521765
SELECT
C21 Mall
9.527644
SELECT
IIT Indore
13.845518
SELECT


53% 10:11
Selected area
IIT Indore
Empty spots
2
Reserve a slot:
1
3


53% 10:11
Spot suggested:
1
ID
3
BOOK
1
BOOK
2
BOOK
Welcome toSilver Springs!


34% 6:14
Timer
13
START NAVIGATION


53% 10:44
Head south-east
Then
VIJAY NAGAR
Jhalariya
Kanadiya
Indore
Bhicholi Mardana
Morod
42 min
RE-CENTER



B Back-end development

B.1 Server

B.1.1 Javascript and Node.js

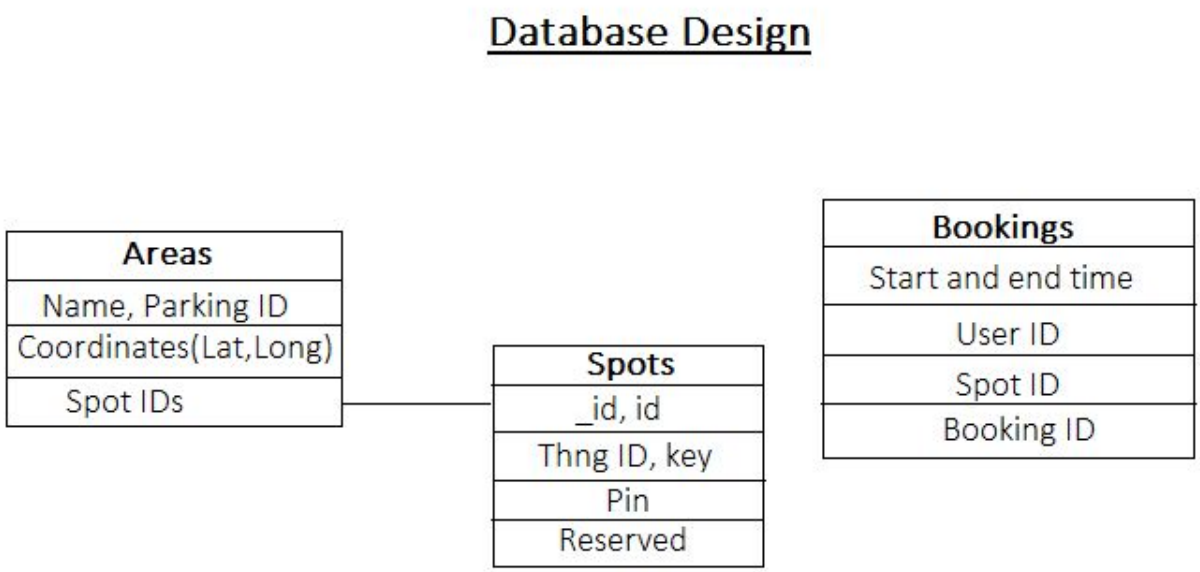
The back-end development involved mainly server development using node.js. JavaScript is a dynamic programming language where client-side scripts executed by web browsers can process data asynchronously and alter the page being displayed. Thanks to its widespread support by virtually all web browsers, relative ease of use, and flexibility, JavaScript has become the de facto solution for writing dynamic, client-side applications.This ongoing JavaScript revolution aligns well with the core idea of the Web of Things, which is to integrate devices to the web so they become more accessible and easier to program. In other words, make it possible to interact with devices just like any other resource on the web by using well-known web standards. Node.js provides an event-driven architecture and a non-blocking I/O API that optimizes an application's throughput and scalability. This model is commonly used to design high-performance real-time web applications. The idea behind Node is to provide a framework in which high-performance server-side web applications can be written. Unlike other servers where you deploy your application in a running server instance, with Node your application is the server. Node isn't JavaScript, but JavaScript is the language used to build Node applications.

B.1.2 Asynchronous Programming

Node is largely based on the principle of asynchronous programming which is a good step toward scaling a server. It offers two main patterns for dealing with asynchronous calls: callbacks and event listeners.

B.2 Database

The database used to store data is MongoLab. It is a cloud hosted MongoDB. The database is divided into collections and each collection has its own set of entries called documents stored in form of a JSON object.



The database consists of 3 collections.

1. areas:
    a. _id: This is an unique id for every parking area.
    b. name: This denotes the name of a particular parking area.
    c. Loc: This denotes location of the parking area in form of a tuple [Longitude, Latitude]
    d. spots: This is array of "_ids" of the spots in that area.

2. spots:
    a. _id: This is an unique id for every parking spot in the city.
    b. id: This is assigned id respective to parking area. This id number will help user to select a spot while booking since it signifies the distance of parking spot from the entrance of parking area. For example, between spot of id=2 and id=4, it would be better for the user to book the spot with id 2 since it would be closer and less confusing for the user to find it.
    c. thngId: This is unique id of the sensor generated when the sensor was connected and configured at evrythng cloud.
    d. thngKey: This is unique key of the sensor generated when the sensor was connected and configured at evrythng cloud.
    e. pin: This is the GPIO pin number of the Raspberry pi to which the corresponding sensor is attached.
    f. reserved: This is a flag variable of value 0 or 1. 0 meaning the spot is empty and is not involved in any booking or reservation. 1 means that either the spot is booked or reserved.
3. bookings:
    a. _id: This is the unique booking id.
    b. user_id: This denotes the user id of the user involved in the booking. It is actually the FCM registration token which is further user for sending push notifications to this user related to this booking.
    c. spot_id: This denotes the id of the spot involved in the booking.
    d. start_time: The timestamp at which the car was parked.
    e. end_time: The timestamp at which the spot was vacated.

B.3 Evrythng Cloud

The first step to using the Evrythng cloud is setting up a project in their dashboard[5].

In the evrythng cloud a product is a class of physical objects (think TV model or car type) but not a unique instance (think serial number). They're a conceptual entity, a model of a physical object, and should only contain information that many physical objects of this class share—attributes such as size, image, weight, and color—but no real-time information such as location, sensor readings, or current state.

A product can be created using the following request:

```
curl -X POST "https://api.evrythng.com/products?project=$PROJECT_ID"

\-H "Authorization: $EVRYTHNG_API_KEY"

\-H "Content-Type: application/json"

\-d '{ "fn": "WoT IR Sensor", "description": "A Web-connected IR Sensor"}'
```

Note the ?project=$PROJECT_ID query parameter, which tells EVRYTHNG to store this product inside the project that we created in their cloud.

In the Evrythng cloud Thngs are the digital representation of unique instances of physical objects: the sensor in our project. For each unique device or object that we web-enable, we create a unique Thng.

This can be done with the following command:

```
curl -X POST "https://api.evrythng.com/thngs?project=$PROJECT_ID"

\-H "Authorization: OUR_EVRYTHNG_API_KEY"

\-H "Content-Type: application/json"

\-d '{ "name": "IIT-1", "product":"'OUR_PRODUCT_ID'"}'
```

You can see that we're also sending the product ID in this request.

We can easily generate a Thng API key that allows your device to see and edit only itself. Send a POST (using your operator API key) to the end point https://api.evrythng.com/auth/evrythng/thngs with the ID of your Thng, as follows:

```
curl -X POST "https://api.evrythng.com/auth/evrythng/thngs"

\-H "Authorization: OUR_EVRYTHNG_API_KEY"

\-H "Content-Type: application/json"

\-d '{ "thngId": "'$THNG_ID'" }'
```

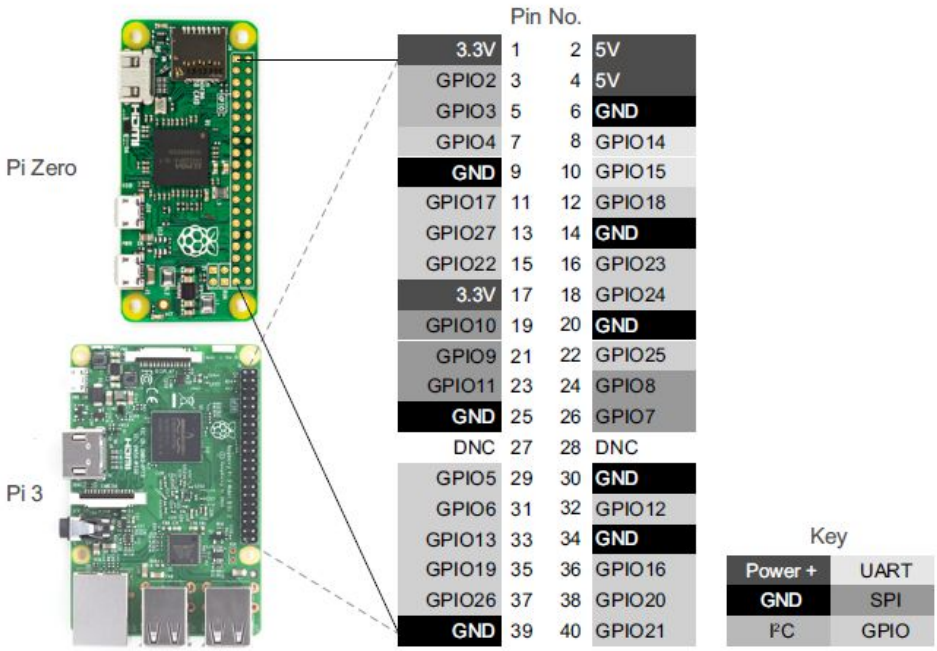
This request generates the following response:

{"thngId": "UCE7qfbK8VKwdt8kAfqtbwmd","thngApiKey": "M1ST3RP0TAT0H3ADROCKS"}

The thngApiKey field contains an API key that allows the device to see and update itself. We store these thngId and ThngApiKey in our database corresponding to unique spots.

4.2 Hardware

Each IR sensor has 4 pins: 5V Input, Ground, Digital output and Analog output. All connections are currently implemented on the breadboard. There are 40 pins on the Raspberry Pi and they are classified according to their purpose: like Power, GND, GPIO and DNC. The pin corresponding to 5V output and ground of the Raspberry pi is connected 5V input and ground of every sensor. Since we need the value of the sensor status in terms of 0 and 1, we make use of only digital output. So the gpio pins are connected to digital output of the sensors.



A file containing spot ids (_id) of the spots monitored by the connected sensors is maintained on the Pi. This file is an input to a script continuously running on the Pi since it's start up. This script does 3 tasks: For each id in the file, respective thng id and thng key is searched from the database. Once the thng credentials are obtained a connection is made from Pi to Evrythng server via MQTT protocol. Finally, after all connections are established it keeps reading the sensor values to detect change in the value and if change is detected the new value is instantly updated on the Evrythng cloud.

4.3 Summary

This is how our smart parking system has been implemented. The source code is available at github. We now describe our testing on the system.

Chapter 5

Testing and results

The testing was done based on various scenarios and cases. The maximum number of users while testing multiple users case were 3. Simply hand was placed over the sensor as a sign of user reaching the parking spot.

1. **Access to only authorized users:**
    a. Test: GET and POST requests were sent to the server through POSTMAN without any token in headers and expired id tokens.
    b. Result: JSON object {“message” : Unauthorized} was received.
2. **Nearby parking area detection:**
    a. Test: Tried booking when at locations with and without parking areas nearby (within 2 km range).
    b. Result: User location was rightly read and nearby parking area was displayed in case it was in the vicinity of the user.
3. **Sequential bookings**: One user has booked the spot, reached in time and parking time is on.
    a. Test: Other users request the free spots and book.
    b. Result: The booked spots are not provided in the list of free spots. All bookings work smoothly in an independent manner by receiving notifications in time based on whether they reach on time or not and on vacating the spot with correct total parking time.
4. **Three ongoing bookings:** One user books spot 1 of a certain parking area, second user books spot 2 of same parking area and third user books spot 3 of different parking area.
    a. Test:
        i. No user reaches in time.
        ii. One user reaches in time other two don’t.
        iii. Two users reach in time one doesn’t.
        iv. All users reach in time.
    b. Result: Appropriate notifications received by the users in all cases.

Chapter 6

Limitations and Edge Cases

Though we have tried to come up with an efficient and robust design and implementation of the smart parking problem a few challenges still exist within this project. In this section we introduce these problems and wherever possible we mention a solution that can be worked on to improve this project.

6.1 Limitations

A problem with any kind of large scale system is the failure of one of its components. Our system is no different.

6.1.1 Sensor Failure

A very important and likely scenario for failure in our system are sensors. Though usually very robust, sensors are exposed to a wide variety of harsh conditions. This can lead to some sensor not functioning correctly or stop working completely.

To tackle the problem of sensor failure we can come up with multiple solutions that are given below:

A. Assign multiple sensors to spots and treat the data coming from these sensors as one single entity. This approach can function nicely since we don't have to alter our system much just instead of the pi reading data from one sensor it reads data from multiple sensors and writes data to a single entity in the evrythng cloud. The pi can therefore detect failure of some sensor when sensors assigned to the same spot are giving different readings. This he can directly report to the central server where we would have to create a endpoint to log failure requests. Then a maintenance team can manually go and inspect the particular parking spot.
B. Additionally add a image recognition device (camera) to the pi and consult data from both the sensor and the camera when updating status of the evrthng cloud entity. The pi can run a vehicle recognition program when the sensor detects an obstacle on top of it. This program if identifies the vehicle will update the status of the evrythng cloud entity.
C. Provide scheduled maintenance checks for sensors in areas to reduce the possibility of failures.

6.1.2 Evrythng cloud failure

Another point of failure though less likely is our middle man between backend server and hardware i.e. the evrythng cloud. Though its is very less likely that a cloud service fail, it has its scheduled downtimes and maintenance issues. Due to such causes the evrythng cloud is a point of failure.

To tackle the problem of the evrythng cloud engine failing, following solutions are described:

A. Instead of keeping a middleman cloud service directly connect the pi to the internet. However this is a time consuming and cost-ineffective task and can also lead to problems since the network design in different parking areas maybe different (for e.g. there maybe proxy servers, NATs, bizarre firewalls etc in between the Pi and the Internet).
B. Setup our own dedicated cloud engine for this project. This can help but will have to be researched thoroughly.

6.2 Edge cases

As with any large scale system design a complete solution for all problems is impossible. Our system is no different. Though we have tried as much as possible to remove any kinks that may arise there are still some evident problems that require future addressing.

6.2.1 Unreachable Spot

An unaddressed problem that exists in a system is that the buffer time allocated for reaching parking spots in case of a booking. This time is set to a default value currently but ideally this time might not be enough since the parking spot might not be practically reachable in that particular amount of time.

This problem can be addressed by applying rigorous mathematical treatment to generate a formula for calculating a buffer time.

Another solution to this problem can be generated by applying sophisticated machine learning algorithms on the spots to analyze the time it takes to reach a spot and then define a optimal buffer time window.

6.2.2 Incorrect Sensor Data due to Random Natural Causes

Since our sensors will be embedded into the ground and they are not highly specialized to detect only vehicles on top of them, it might be possible that a scenario of the following kind occurs: User books a spot and before he reaches the spot a stray animal (say cat) reaches the spot and sits on top of the sensor. This might lead to the user being sent a notification that his parking is now started which is incorrect. A similar scenario can occur wherein the user books one spot but parks on another spot. This can also lead to unexpected behaviour.

To tackle problems like these some of the following methods could be used:

A. Using multiple sensors for a spot
B. Providing navigation instructions to the user when reaching a parking spot
C. Using image processing devices to correctly identify that a booked spot has been parked upon
D. Using high quality metal detection sensors

Chapter 7

Conclusions

To conclude this report we would like to say that this has been a thorough learning experience. We have been exposed to a fine grained understanding of the workings of the web. We have learnt deeply the difficulties involved in developing a large scale system. We understood the process of developing RESTful APIs and providing APIs as a Service to enable building of complex applications and extend the existing applications functionalities.

Through the development of this app we have been able to come up with an algorithmic process to develop new and complex IoT applications. We state the steps below:

- Connect physical devices to the internet by connecting these devices to a gateway that is connected to their unique virtual entity at all points in time.
- Always keep on executing a publish/subscribe code on the gateway to publish real-time changes in the physical world to their corresponding virtual entity. This ensures that the data sent into the virtual world is always up-to-date.
- Since now we have the real-time data for a physical entity, generate a middleware service that interprets this data and gives it meaning so that it can be fruitfully utilized by end level applications.
- Create an end-level applications that uses this real-time data by subscribing to the middleware service for updates of particular virtual entities.

So to conclude, we have tried to provide users with the parking data all over the city via the Internet and a provision to book or reserve the spot of their choice. The updates are real time and hence, reliable. We have also provided a design of the parking architecture that would facilitate the provisions provided in a smooth way.

Chapter 8

Future work

As mentioned in the sections on edge cases and limitations, the prototype of the proposed system have lot of areas that need to be looked upon to improve the overall performance of the system while avoiding the limitations.

To start with, the sensors are prone to failures and therefore a comprehensive mechanism to tackle failures on sensors can be worked upon. To suggest one such possible method, a spot can be monitored and defined by multiple sensors. So there will not be a single point of failure that will result in incorrect status of the parking spot. In case one sensor fails, others will still provide the right data. This will guarantee the right value even if one of the sensor fails, since in that case majority of other sensors will provide a different and right value.

Another area of work that can be done is to store and provide navigation to the parking spots inside parking areas on the client UI. This can lead to elimination of human error in the form of parking at the wrong spot. Moreover, it will help the users to find the required spot without any hassle and hence, it will lead to a systematic organisation of the parking process as a whole.

A significant scope exists for making the sensor connection to the pi wireless. That way a lot of cost reduction can occur since cost of fitting and maintenance of wires will be reduced.

A machine learning strategy should be implemented for calculating buffer times. This would lead to optimal buffer time calculation which would further lead to server getting communications in a much more streamlined manner from the sensors. As mentioned before, we are assigning a fixed buffer time without considering any special cases. But if previous data is considered while assigning the buffer time, it would be much more practical. For example, there is heavy traffic at a particular time on particular day in a particular area. So every time a user makes a booking in that area, the time from his location to the parking is stored, studied and the buffer time values are assigned based on this.

Another area of work can be developement of a script to be run on the pi that automatically generates a virtual entity for a new sensor and adds its values to the database adds a key in the spots list of the particular area where it is being used and adds the spot id into the list of spots for which the program has to run at startup of the pi. This would automate the process of adding new spots or sensors to the parking.

References

1. Gubbi, J., Buyya, R., Marusic, S. and Palaniswami, M., 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. Future Generation Computer Systems, 29(7), pp.1645-1660.
2. Guinard, D., Trifa, V. and Wilde, E., 2010, November. A resource oriented architecture for the web of things. In Internet of Things (IOT), 2010 (pp. 1-8). IEEE.
3. Guinard, D., Trifa, V., Mattern, F. and Wilde, E., 2011. From the internet of things to the web of things: Resource-oriented architecture and best practices. In Architecting the Internet of Things (pp. 97-129). Springer Berlin Heidelberg.
4. Guinard, D., Trifa, V., Pham, T. and Liechti, O., 2009, June. Towards physical mashups in the web of things. In Networked Sensing Systems (INSS), 2009 Sixth International Conference on (pp. 1-4). IEEE.