Smart Parking using Magnetometers and Mobile Applications

Fredrik Bernspång

Master of Science in Engineering Technology
Computer Science and Engineering

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Abstract

People are using cars for transportation every day all over the world. This leads to a constant demand for parking. In public areas there are usually designated places for parking that many people share. The shared parking spots with the best location and cheapest fee thereby become attractive. Some motorists cruise around hoping to find an attractive spot that is not occupied. This cruising is generating a lot of unnecessary Carbon Dioxide as well as causing other problems on the streets. This thesis proposes a solution to these problems by providing motorists with real-time information about the availability of spots in parking houses through a smartphone application. In order to reach as many potential users as possible the smartphone application was developed with PhoneGap and standard web technologies. To achieve scalability on the back end Google App Engine was used. By focusing on multi-story car parks and parking lots the cost for deploying the system is inexpensive due to the low demand for sensors compared to those that require a sensor for each parking spot.
Preface

This thesis constitutes my final course as an undergraduate student at Luleå University of Technology. The work was carried out at Neava’s office in Luleå between November 2010 and May 2011.

First I would like to thank my advisor at Neava Tore Johnsson for all his engagement throughout this project. Thanks also to Kåre Synnes, my advisor at Luleå University of Technology. Finally I would like to thank Emily Granath for all her support during this time.
## Table of Contents

Chapter 1 – Introduction ........................................................................................................ 5  
1.1 Background .................................................................................................................. 5  
1.2 The problem .................................................................................................................. 5  
1.3 Purpose .......................................................................................................................... 6  
1.4 Limitations .................................................................................................................... 6  
1.5 Related work .................................................................................................................. 7  
  1.5.1 SFpark ....................................................................................................................... 7  
  1.5.2 ParkNet ..................................................................................................................... 8  
  1.5.3 Open Spot & Roadify ............................................................................................... 9  
  1.5.4 Others ..................................................................................................................... 9  
Chapter 2 – System design .................................................................................................. 10  
  2.1 Key components .......................................................................................................... 10  
  2.2 Back end ..................................................................................................................... 10  
    2.2.1 The database server ............................................................................................... 10  
    2.2.2 Cloud computing .................................................................................................. 11  
    2.2.3 Choosing cloud provider ...................................................................................... 11  
  2.3 Client application ......................................................................................................... 12  
    2.3.1 Know your audience ............................................................................................ 12  
    2.3.2 A cross-platform alternative ............................................................................... 12  
  2.4 The sensor ................................................................................................................... 13  
    2.4.1 Reflection based .................................................................................................. 14  
    2.4.2 Magnetometer ...................................................................................................... 15  
    2.4.3 Sensor data .......................................................................................................... 15  
  2.5 Data interchange format ............................................................................................... 15  
  2.6 Revised system design ................................................................................................. 16  
Chapter 3 – Implementation ............................................................................................... 18  
  3.1 Back end ..................................................................................................................... 18  
    3.1.1 Data models .......................................................................................................... 18  
    3.1.2 Channels ............................................................................................................... 18  
    3.1.3 Add a parking house ............................................................................................ 19  
  3.2 Client application ........................................................................................................ 21  
    3.2.1 Resources ............................................................................................................ 21  
    3.2.2 User experience .................................................................................................... 22  
  3.3 Searching for parking ................................................................................................. 22  
  3.4 Simulated sensors ........................................................................................................ 24
Chapter 1 – Introduction

1.1 Background

Studies [1] show that a car is parked 95% of its lifetime. It is also known that most car trips start and end on a parking lot. Therefore parking is naturally an important matter for car owners.

Probably, most people have asked themselves at some occasion; where can I find a parking spot? Later, rephrasing the question; where can I find a free parking spot?

Today there are numerous navigation solutions based on GPS-capable devices that can answer the first question. The second question requires additional information and dynamic updates as parking spots become available or occupied.

1.2 The problem

When it comes to private parking spots, such as a driveway in front of a house, the occupation of it is usually known to the people frequently parking there. The problem is when larger groups of people, with no relation to each other, share the same parking spots. This phenomenon is very common in public places such as business areas etc.

Depending on location, time of the day and price some parking spots are more popular than others which are more likely to be available. Because the demand is higher than the supply motorists cruise around hoping one of these attractive spots becomes available.

Drivers cruising around looking for parking spots are wasting a lot of their time and money, at the same time tons of unnecessary Carbon Dioxide is generated. As drivers get more and more frustrated they tend to bend more on the parking rules, for example, it has been shown that the relationship between occupancy of the attractive spots and illegal parking (for example double parking and obstruction of crosswalks, bus stops etc.) is exponential [4]. This in turn creates problems for delivery trucks, buses and everyone else on the streets.

A study [2] conducted in 2007 showed that the average cruising time in a 15-block business district in Los Angeles was 3.3 minutes. Over the course of a year, the traffic caused by cruising vehicles in this district is equivalent to 38 trips around the earth.

In a similar study [3] from New York the average cruising time was 3.25 minutes. In this case, the distance traveled by cruising vehicles in the studied district is in a year longer than a one-way trip to the moon.

In [5] the authors examine the future of personal transportation in the following 15 megacities (over 10 million inhabitants): Chicago, New York, London, Moscow, Paris, Buenos Aires, Mexico City, Rio de Janeiro, São Paulo, Bangalore, Calcutta, Delhi, Mumbai, Hong Kong and Shanghai. They consider current and expected future values of the following factors: population, wealth, level of motorization, public transportation and modal split. Based on this analysis and some selected urban transportation plans and strategies, they make projections through 2025 for changes in ownership of personal vehicles; distance traveled per capita by personal vehicle within inner core, for commuting, and for leisure; and for number of road fatalities per capita for each city. Due to already existing problems with congested traffic and limited space within the inner core of large metropolitan areas the projections don’t show any
significant increase in traffic within this area. Substantial increases in personal vehicle ownership in the four Indian cities and Shanghai are predicted. No substantial decrease in the reliance of personal vehicles is foreseen in any of the examined cities, instead an increase is forecasted in Brazil, India and China.

Based on [5] it is assumed that the previous examples from Los Angeles and New York are not specific to these cities only, rather it is believed that this problem exists in most big cities. From the projections of [5] these problems are not expected to disappear within the next 15 years.

1.3 Purpose
The purpose of this master’s thesis is to describe parts of a system that if fully implemented could help frustrated drivers find available parking spots quicker and thereby reduce pollution and unnecessary fuel consumption.

1.4 Limitations
Suitable properties for sensors will be discussed but no real-life trial will be performed, only simulations.

Instead of monitoring the number of cars that arrive and leave with a sensor based solution it might be possible to accomplish this task with image analysis or other techniques, although this will not be further investigated in this thesis.

As pointed out by [6] the sole availability of parking spots generates traffic, if there were nowhere to park in a city citizens would have to use other ways of transportation. The balance between the total amount of traffic and the number of parking spots will not be considered in this thesis.

Normally a car is parked in a multi-story car park, in a slot on the curb or in a parking lot. Multi-story car parks and parking lots share some properties; they usually have a limited number of entry and exit points and a fixed maximum capacity. If the total capacity is known, it is possible to establish how many spots are available at any time, by monitoring the number of cars that arrive and leave. With curbside spots it’s different; each spot is independent from the other. Therefore the thesis focus will be on multi-story car parks and parking lots.

The availability data could be distributed in many ways, for example thru variable message signs (VMS), see picture 1. However, the only distribution method that will be considered in this thesis is by a smartphone application.
1.5 Related work

Today there are a couple of ongoing projects that investigate the problems associated with parking. Below some of these will be discussed.

1.5.1 SFpark

SFpark is a very ambitious project both feature wise and in size. The aim of this project is to reduce some of the traffic problems in the city of San Francisco. Currently the project is running in a two year pilot phase and involves 6000 metered parking spots and 15 multi-story car parks [7]. The whole project is based on the principle that if parking prices are set just right, at least one parking spot will be available on every block [8]. Data about the parking spots is collected with wireless sensors [7, 8]. Every single parking spot must therefore be equipped with one of these sensors. This data is processed monthly [8] to determine where and when prices should be adjusted in order to achieve the ideal (around 20% availability [8]) balance of available parking spots. As part of the project parking meters have been replaced in order to offer more payment options and to make it easy to adjust parking rates [7].

San Francisco is one of the first cities in the world with a complete smart parking management solution [8]. The project enables drivers to get real-time data of available parking spots both through smartphones and variable message signs to make it easier to park [7]. The city benefits from reduced congestion and less pollution.
SFpark looks like a very promising project that will be very interesting to follow in the future. It comes at a price though; the price tag is around $25 million [8].

1.5.2 ParkNet

ParkNet is a mobile parking monitoring application. Instead of installing sensors at each parking spot mobile sensors are used. This significantly lowers the total number of required sensors. The sensor used relies on ultrasonic measurements together with GPS readings that are corrected through environmental fingerprinting [10].

![Early version of the ParkNet sensor mounted on the side of a car.](image)

When driving around in a car equipped with a sensor, it will detect cars parked at one side (the right side if driving in right-hand traffic and vice versa) of the street. With the algorithm proposed in [10] the authors claim that they reach approximately 95% accuracy in terms of obtaining parking counts.

Remember that this sensor is mounted on a moving vehicle. To be able to say anything about where the free parking spot is located a GPS unit is used. Given that a parking spot is about 7m in length the GPS unit needs to be very accurate to be able to tell one parking spot from another. To acquire a higher degree of accuracy for the GPS readings the authors use something called environmental fingerprinting; select a couple of fixed positions with known positions and compare the GPS reading with the known value each time the car passes by the fixed position. This makes it possible to compensate for the error of the GPS unit and thereby improve the accuracy.

Mobile sensors could be installed in cabs or municipal vehicles, who already circle a lot of areas every day [11]. A negative effect from using mobile sensors instead of stationary ones is that the data is not updated in real-time. It is not suggested that cabs should make extra trips to collect data, they only collect data as they drive around as usual.

Compared to SFpark, ParkNet is significantly cheaper. The authors of [10] estimate the cost saving factor to 12.5, or greater. This cost benefit comes from the fact that ParkNet provides a non-guaranteed, random sampling of the parking process, whereas fixed sensors provide continuous monitoring wherever they are installed [10].
1.5.3 Open Spot & Roadify

Open Spot and Roadify are two applications designed for smartphones. The concept of both applications are pretty much the same; when you are about to leave your parking spot you report this with your phone so that someone who is currently looking for a spot can find it [12, 13]. Open Spot is currently only available in the US, Canada, and the Netherlands [12]. Roadify is currently only available in selected areas of New York [13].

Both applications feature a karma point system to encourage people to report when they leave a spot and prevent users from only finding spots.

The concept relies completely on users instead of sensors so it’s very important to find enthusiastic users. This can of course be positive and negative depending on the situation; negative when there are none or few active users in a city and positive if there are many active users.

1.5.4 Others

For some multi-story car parks it’s possible to see real-time vacancy information online, for example at the airport in Zurich [14]. There are a number of applications for smartphones that gets data by parsing these websites, for example SwissParking [15] and Vacant Parking [16].

Unfortunately it is very few car parks that publish this information online so the benefits of these applications are very limited.
Chapter 2 – System design

2.1 Key components

Providing users with parking availability is the core of the whole system. Therefore a back end of some kind is required where this data can be stored and distributed from. Parking availability can change very quickly, rendering the data useless. To keep track of these changes and continuously update the back end with fresh data sensors are introduced into the system. Finally this data needs to be accessible to users, specifically users driving around looking for parking. A smartphone application that users can use anywhere they like seems like a suitable solution.

Picture 6: First draft of the system design.

Using the client application users should be able to search for a free parking spot close to their location or close to a specific address (their final destination). When the back end receives the search query it responds with a result, meaning there needs to be two-way communication between the back end and client applications (see picture 6). As mentioned parking availability can change quickly, therefore it would make sense to provide users with real-time updates of results. The sensors should be kept as simple as possible and not have to deal with incoming messages, just send updates to the back end.

2.2 Back end

2.2.1 The database server

Usually users don’t really care what type of back end is used, as long as it stays online and responds quickly to queries. From the developers perspective it can make a huge difference though. Both these perspectives are important to consider when choosing back end.

A popular way of storing data is using databases with support for relationships. Many of these databases are based on Structured Query Language (SQL), which can be used to access and modify the database. There is a wide selection of products and the technique is very established. With the help of a server-side scripting language it’s possible to generate results in a suitable format that clients can access through a regular URL. On the client side the result can be parsed and presented to the user. For the system discussed in section 2.1 this setup could certainly provide the necessary functionality.

There are several distributions of the operating system Linux which even makes it possible to set this up for free. Obviously it’s still necessary to pay for the actual hardware.

Now imagine that the application supported by this database becomes very popular. Eventually the number of users wanting to read data from the database is going to exceed the
available resources of the server hosting the database. Of course it would be possible to invest in a more powerful server but that is just a temporary solution, eventually the new server would also become too weak. One option could be to deny users access during peak time, but that may result in losing popularity and revenue. It is also likely that the popularity of the application will vary over time, making a powerful server excessive during nonpeak hours.

2.2.2 Cloud computing

It would be beneficial not to rely on a single server and only pay for the actual resources that the application consumes. There are multiple providers that offer the possibility to deploy applications on their cloud infrastructure. Basically that means the provider virtualizes the application on multiple servers. This can also take away a lot of the burdens that comes with setting up infrastructure for the programmer, enabling the programmer to focus more on the application itself.

The provider simply charge for the amount of resources that the application consumes. If the application becomes popular it will consume more resources but the infrastructure to handle the added popularity is already in place. The cloud infrastructure can thereby provide scalability and a solution to the problem discussed in section 2.2.1.

Something worth considering before outsourcing infrastructure is of course what happens if the provider suddenly goes bankrupt or changes strategy. Avoiding provider specific features as much as possible and having a backup strategy in case something like this happens could prove very valuable, as pointed out by [17]:

“Just as large ISPs use multiple network providers so that failure by a single company will not take them off the air, we believe the only plausible solution to very high availability is multiple cloud computing providers.”

2.2.3 Choosing cloud provider

Since December 2010 Amazon offers new users to try their cloud service for free during the first year but at the time when a provider was chosen for this thesis they did not [26]. Microsoft’s Windows Azure does offer a free trial but after 90 days they start charging for usage of the database [18]. The model used by Google App Engine is based on how much resources that are consumed, and all resources up to a certain level are free without any time limit [27]. This model seemed best suited for this project because of the low risk; free hosting as long as there are few users and if the application becomes popular it’s hopefully bringing in enough revenue to at least cover the hosting expenses.

Many providers give the user the opportunity to freely install nearly any *NIX compatible software. App Engine supports a number of APIs and frameworks and requires developers to use either Python or Java. There are of course both positive and negative aspects of this. A positive thing is that many of the administration challenges that come with building scalable infrastructure are removed. At the same time it’s not possible to use any other programming language than the two supported ones which of course can be negative if the developer is more comfortable using another language.

App Engine lacks support for a regular relational database, instead it offers a none-relational datastore [28]. The datastore has a SQL-like syntax called “GQL”. Any common SQL functionality that doesn’t scale efficiently is intentionally not supported by GQL. This can be
somewhat cumbersome for developers at first but in the end the scalability is worth a great deal. Unfortunately this also means that a lot of the great resources out there that are based on SQL usually don’t work on App Engine without modifications. In response to this a number of open source projects have been created. Thanks to one of these projects called Django-nonrel [24] it’s possible to deploy Django [25] based applications on App Engine. Django is a very feature rich framework for web development that enables rapid development.

2.3 Client application

2.3.1 Know your audience

Potential users of this application are all car owners that also own a smartphone of some kind. It is very difficult to say anything more specific about these potential users. For instance it’s impossible to guess what operating system their phones use, it’s probably safe to assume that in a group such as this one many different operating systems are represented. In table 1 below the market shares of operating systems for smartphones are presented.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>2010 share</th>
<th>change</th>
<th>2009 share</th>
<th>change</th>
<th>2008 share</th>
<th>change</th>
<th>2007 share</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbian</td>
<td>38%</td>
<td>-8</td>
<td>46%</td>
<td>-8</td>
<td>54%</td>
<td>-10</td>
<td>64%</td>
</tr>
<tr>
<td>Android</td>
<td>22%</td>
<td>+18</td>
<td>4%</td>
<td>+3</td>
<td>1%</td>
<td>+1</td>
<td>-</td>
</tr>
<tr>
<td>BlackBerry</td>
<td>16%</td>
<td>-5</td>
<td>21%</td>
<td>+5</td>
<td>16%</td>
<td>+6</td>
<td>10%</td>
</tr>
<tr>
<td>iOS</td>
<td>16%</td>
<td>+2</td>
<td>14%</td>
<td>+7</td>
<td>7%</td>
<td>+4</td>
<td>3%</td>
</tr>
<tr>
<td>Windows Mobile</td>
<td>4%</td>
<td>-2</td>
<td>6%</td>
<td>-6</td>
<td>12%</td>
<td>0</td>
<td>12%</td>
</tr>
<tr>
<td>Bada</td>
<td>2%</td>
<td>+2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Windows Phone 7</td>
<td>1%</td>
<td>+1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Others</td>
<td>1%</td>
<td>-8</td>
<td>9%</td>
<td>-1</td>
<td>10%</td>
<td>-1</td>
<td>11%</td>
</tr>
</tbody>
</table>

*Table 1: Market shares of operating systems for smartphones. [19]*

Based on the figures in table 1 it’s clear that the application needs to support more than a single operating system to reach out to more than 38% of the potential users. Symbian stands out as the biggest but looking at the trend it seems like its losing ground to rapidly growing Android and iOS. Furthermore Nokia has declared that they are moving from Symbian to Windows Phone which most likely means the trend for Symbian is not going to change in the near future [20].

By developing two versions of the application; one for Android and one for iOS 38% of the market is reached. This might be considered good enough for some. It is however also important to note that maintaining multiple versions means a lot of extra work compared to a single version. This problem has been declared the “inherent engineering challenge” by Bret Taylor who works as Chief Technology Officer at Facebook [21]. Facebook currently have 200 million people accessing through mobile devices, spread over seven different platforms [21]. Meaning there is a high demand for continuously added features. Deploying new features on seven platforms obviously causes a great deal of extra work compared to if it was only one.

2.3.2 A cross-platform alternative

Instead of developing and maintaining multiple versions of the same application for different operating systems it would be easier to just have one version that works on many different
operating systems. This can be achieved thanks to an open source project called PhoneGap [29].

PhoneGap is a web platform that exposes native mobile device APIs and data to JavaScript. Meaning, developers can write applications in HTML/CSS/JavaScript that runs in the phone’s built-in browser and still make use of the device APIs, see picture 7.

![Diagram of HTML5, CSS3, and JavaScript with PhoneGap](image)

*Picture 7: PhoneGap exposes the device APIs and data to JavaScript.*

PhoneGap supports the following operating systems: Android, BlackBerry, iOS, Symbian and webOS. Together these operating systems make up for more than 90% of the total market, see table 1. In the future this might grow, the PhoneGap web page says that they are working on support for Windows Phone in addition to already supported operating systems.

Applications developed with PhoneGap runs as native applications and can therefore be deployed through normal distribution channels such as Android Market, App Store etc. The creators of PhoneGap have also launched a build service that greatly simplifies compilation for multiple operating systems [22].

### 2.4 The sensor

The system needs to keep track of the number of available parking spots in the parking houses to be able to provide users with this data. Obviously this will not work for all parking houses, only the ones involved in the project. When a new parking house is added to the database the total capacity needs to be registered. Then by installing sensors at all entrances and exits it’s possible to keep track of the number of available spots. So a sensor of some kind that is capable of sending messages to the back end is needed.
2.4.1 Reflection based

A photocell sensor is fairly cheap and can be used for many different applications. The principle is rather simple; a light source emits a light onto a reflective surface that reflects the light back onto a photoresistor, which can detect if the light beam is cut off, see picture 8.

![Picture 8: The principle of a reflection based sensor.](image)

Imagine a sensor placed at the entrance of a parking house. When the sensor detects a passing car the total number of available spots in that parking house is decreased with one. However, if someone walks by the sensor this will also be recognized as a decrease in available spots, resulting in miscalculation. This problem can be solved by using two sensors mounted with some distance between them, see picture 9.

![Picture 9: Detecting a passing car with two reflection based sensors.](image)

The distance between the sensors would have to be shorter than a small car but still longer than a wide person. That should not really be a problem, but if the person also pushes a baby carriage this could incorrectly be recognized as a car.

It is important that the sensor solution is easy to install and replace when needed. Not all parking houses looks the same; in some there might not be any suitable place for installing two sensors close to the entrance/exit with reflective material on the opposite side.
2.4.2 Magnetometer

It would be beneficial with a sensor that is more independent of the environment in the parking house, easy to install/replace and can tell a car from a person with a baby carriage.

A magnetometer that can detect passing vehicles by changes in the magnetic field could be used. Instead of requiring two sensors and two reflective surfaces only one unit would be needed. This would simplify both installation and maintenance of sensors. Furthermore it would also be possible to tell a car from a person with a baby carriage. A sensor that fulfills these requirements is the road marking unit used in the iRoad project, see [30], [31] and [32].

2.4.3 Sensor data

Regardless of type of sensor used the data needs to be communicated to the back end. Some parking houses might already have a connection to the Internet but not all of them. Even if it would be possible to use an existing connection it’s probably better to rely on a solution that is independent of the parking house, something that works everywhere. The sensor installation would thereby be very similar in all parking houses. A small gateway to which the sensors could be connected with the ability to communicate over the mobile network could provide this. To further simplify installation and maintenance it would be great if the sensor and gateway could use wireless communication. A gateway that could provide wireless communication with sensors and also communicate with the back end over the mobile network is the Telit GG863-SR [33].

2.5 Data interchange format

As discussed in section 2.1 the system needs communication to be meaningful; sensors need to send data to the back end and client applications require two-way communication with the back end. Client applications run on smartphones that normally communicate over the mobile network. If the gateway solution discussed in section 2.4.3 is used sensor data will also be transferred through the mobile network. When dealing with the mobile network it’s important to keep the amount of traffic down as much as possible. The speed on the network may be slow and data transfer often cost money.

JavaScript Object Notation (JSON) and Extensible Markup Language (XML) are two data interchange formats currently used in industry applications. Both languages are designed to be human readable and easy for computers to parse and use, see picture 10. It is important that computers can parse the interchange format quickly without consuming a lot of resources.

When choosing a data interchange format it is important to consider both size and parser-friendliness.
Picture 10: Example of a search result in XML (left) and JSON (right) viewed in Notepad++.

Counting the number of characters in the example above, disregarding all white space characters, the total number of characters is 466 for XML and 283 for JSON. For this particular example the JSON syntax is approximately 39% smaller than the XML syntax.

In [23] the authors compare differences between XML and JSON with respect to performance and resource utilization. Their results indicate that JSON is both faster and uses fewer resources than XML.

2.6 Revised system design

At the start of this chapter key components where identified and a first draft of the system design was established. After a more in depth discussion about each component a revised system design has emerged, see picture 11.

Picture 11: Revised system design.
To clarify picture 11 it should be noted that it’s only the communication from App Engine to the smartphone application that is JSON-encoded. The communication from sensors/clients to App Engine is handled via URLs, meaning the only data sent to App Engine is in the actual URL. For example when a sensor communicates to the back end about a car passing by it opens this URL: http://xjneava.appspot.com/update/<Sensor ID>/, where <Sensor ID> is an integer that uniquely identifies the sensor.
Chapter 3 – Implementation

3.1 Back end

The back end for this project was implemented in Python and runs on Google App Engine. To speed up the development process the Django framework was used in combination with Django-nonrel.

3.1.1 Data models

Django models are used to describe stored data, in general each model maps to a single database table [34]. In App Engine an application creates entities, with data values stored as properties of an entity instead of database tables [28].

To keep track of the parking houses in this system four models were implemented:

1. ParkingHouse
2. ParkingFloor
3. Sensor
4. OpeninghoursPrice

The ParkingHouse model is used to store general information about a parking house, for example: name, address, contact person and coordinates.

In some cases the owner of the parking house might be interested in knowing how many spots are available on each floor. The ParkingFloor model contains data about capacity and availability for each individual floor to cover this. Of course this requires additional sensors to be installed between floors.

Each sensor is represented by an instance of the sensor model. The sensor is tied to a ParkingFloor and is of type: entrance, exit, up or down. For some parking houses it might be enough with a sensor by the entrance and another at the exit. Depending on the situation more sensors might be required; for example there could be multiple entrances spread across multiple floors. The system allows multiple sensors of any type for each parking floor without any need for extra configuration. At minimum one sensor is required at each entrance and exit though, otherwise the system will soon miscalculate. Each sensor also needs to be within communication distance to a gateway, either wirelessly or connected with a cable.

Finally the OpeninghoursPrice model is implemented to enable parking houses to define different prices depending on day and time. This model is also used to check if parking houses are open or closed.

3.1.2 Channels

As previously mentioned it would be desirable with real-time updates of parking availability. Thanks to the Channel API [36] provided by App Engine this can be achieved without generating more bandwidth then necessary for clients. After the client application receives a search result from the back end a channel is established between these two. This channel is then used by the back end to push possible updates to the client. The client has thereby subscribed to real-time updates from the parking houses included in the original search result. The back end keeps a record of which clients are interested in which parking houses with hash maps in a memory cache [43].
3.1.3 Add a parking house

One of many benefits with Django is the automatic admin interface. The administration interface requires all users to login and can be accessed in a regular web browser from any location. Thanks to this it’s easy to add new parking houses to the system and modify existing ones. The administrator simply enters some information about the parking house in a form and then presses save, see picture 12.

![Add parking house form](image)

*Picture 12: Adding a new parking house to the system through Django’s admin interface.*

The next step is to create a floor in the house with information about capacity and availability, see picture 13. Every parking house is required to add at least one floor. In the case where availability on each floor is not considered the added floor can represent the entire house. If at some point it becomes needed to manually reset the number of available spots this is done with the same interface.
Adding a floor with a capacity of 100 cars to the recently created parking house.

Adding a sensor to the previously created floor is done in a similar manner; the administrator selects a parking floor and assigns a type for the sensor. Once the sensor is added in the system it is given a unique id, see picture 14. Information about sensors is intentionally stored on the back end to achieve flexibility; a sensor thereby only needs to know its own id number.

Picture 14: A list over all sensors, the newly created sensor was assigned id 64005.

When one of the sensors in the system is passed by a car it sends a message with its id to the gateway which in turn notifies the back end by loading the following URL: http://xjneava.appspot.com/update/64005/. Since the back end knows which house and floor the id corresponds to it adjusts the availability accordingly. Thanks to this design a sensor can be moved from one parking house to another without any need of modifying the software on
the sensor, simply modifying the sensor’s settings on the back end will suffice. If a sensor breaks it can quickly be replaced by a second sensor configured to monitor the same area. However, in that scenario the broken sensor should be removed or disabled to avoid counting cars twice at that location.

A parking house will only appear in search results if it’s open, information about when the parking house is open is therefore required, see picture 15.

![Xjneava administration](image)

**Picture 15: The parking house is always open on Mondays and the hourly rate is set to 10.**

It is possible to define multiple intervals for the same day with different prices. The intention with this is to make the system flexible and hopefully cover many existing real-life situations. Again, this data can be modified when needed.

### 3.2 Client application

To reach out to as many potential users as possible, without the added workload of developing and maintaining multiple OS-specific versions, the client application was developed with standard web technologies and PhoneGap.

#### 3.2.1 Resources

Apart from PhoneGap some other resources are used in the prototype application.

Search results are by default displayed on a map, this is accomplished with Google Maps JavaScript API V3 [40]. To support search by address the Google Geocoding API [44] is used to convert the address into geographic coordinates.

If a user prefers, results can be viewed in a list instead of on a map. This list is created with iScroll 4 [42] to make it scrollable in case the results don’t fit on the screen. The lite edition was chosen over the standard edition because of the smaller file size.

To simplify JavaScript coding the xui [41] library is used. Xui is very small, intended for HTML5 mobile web applications and offers great functionality.
3.2.2 User experience

One important matter for every smartphone application is of course whether or not the users actually like the application once they start using it. If not there is a great risk that the application will struggle. One thing that could have a great impact on the experience is the design itself. One user might think the GUI is ugly and another user might think it’s gorgeous, that’s what makes design so difficult. However, this problem exists in all applications with a GUI regardless of what technology is used.

During the development of the client application for this thesis there was a problem with creating user interface components, such as buttons, that felt responsive enough to deliver a good user experience. The companies behind the different operating systems for smartphones have created ready-to-use GUI components so this is not a problem when developing native applications. The problem exist in web applications because mobile browsers wait ~300ms before firing the click event after a button is tapped to be able to interpret a double tap. If the button isn’t intended to have any double click functionality this will make the button feel slow. This problem was solved by applying a method described in [35] that listens for touchstart instead of click.

Another thing that can badly influence the user experience is when users are left waiting for data transfers. It is therefore important to keep transfers small and limited to a minimum. That’s why the prototype application uses the compact JSON format to transfer bigger chunks of data (search results). As discussed in section 3.1.2 channels are used to provide users with real-time updates. Another approach to basically achieve real-time updates involves setting a timer in the client application that refreshes the result every X seconds. However, that would generate a lot more bandwidth than the channel approach.

3.3 Searching for parking

When the smartphone application is started a start screen is presented, see picture 16A. There are two alternative ways to search for parking; the nearby button and the address field.

When the user starts typing in the address field the application starts suggesting possible matches with Google Geocoding API [44], see picture 16B. Every time the user enters
another letter in the field this is sent to Google servers who then return a JSON-encoded list of possible addresses. The result list also contains geographic coordinates for each address, see picture 17.

```
formatted_address: "Bånvägen, Luleå, Sweden",
+ address_components: [ _ ],
  geometry: {
    location: {
      lat: 65.6113045,
      lng: 22.1162119
    }
  }
```

*Picture 17: JSON-encoded result from Google Geocode.*

When the user selects one of the addresses the coordinates for that address is sent to the back end by loading a URL, such as: http://xjneava.appspot.com/search/65.6113045/22.1162119/. These coordinates are then used by the back end to filter out parking houses that are relevant for the user. Parking houses that are currently closed and those that are further than 5 kilometers away from the coordinates are filtered out. Once the filtering is complete a JSON-encoded list of the remaining parking houses is returned to the client application.

If a user presses the “Nearby” button instead of entering an address the application tries to get the user’s current location from the device (by means of GPS/triangulation). This location is then sent to the back end by loading a URL exactly as described above.

*Picture 18: Information about the parking house Kulturens hus.*

When the client application receives the result list it is parsed and parking houses are plotted on a map, see picture 16C. It’s possible to press the icons in the map and get information about the parking houses, see picture 18. There are five different icons that can appear on the map:

1.  This icon represents the location of the address, when searching for a specific address.
2.  This icon represents the location of the user when searching with the “Nearby” button.
3.  This icon represents a parking house with 50 or more available spots.
4.  This icon represents a parking house with 10 or more but less than 50 available spots.
5.  This icon represents a parking house with less than 10 available spots.
The same result can also be viewed as a list, see picture 16D. Here the results are sorted by distance with the closest parking house at the top. The same colors are used to color-code the “Spots” column. The purpose with the color-coding is to quickly give users an idea of the availability.

Beyond the parking houses the JSON-encoded result from the back end also contains a token. This token is used by the client application to establish a channel with the back end. As discussed in section 3.1.2 this channel is then used by the back end to push real-time updates to the client application which in turn updates the user interface.

### 3.4 Simulated sensors

Due to the limited time frame of this project there was not enough time to implement the sensor/gateway part of the system. Instead a web interface was created to simulate this part to enable testing of the other parts.

The interface consists of a list over parking houses and their floors, see picture 19. The minus and plus buttons on the right corresponds to sensors tied to the relevant floor. The highlighted minus button at the bottom is the sensor added in section 3.1.3. This particular sensor has id 64005 which also matches the URL at the bottom left corner in picture 19. In picture 14 it’s possible to see that this sensor was placed at the entrance of the parking house which is why it’s represented with a minus button here, pressing the button will decrease the available spots with one.

When a sensor triggers an update the back end first adjusts the availability and then queues a task to notify currently subscribed clients of the new availability. The task queue [45] improves the responsiveness of the simulation interface. Another benefit is that the task will automatically be retried if it fails on the first attempt. When the task is executed the array with
currently subscribed clients is retrieved from memory cache and the updated availability data is pushed to each channel.
Chapter 4 – Evaluation

4.1 Back end

There are many benefits with using Google App Engine instead of hosting the back end in-house. Maybe the most powerful benefit is the scalability that is otherwise difficult to achieve. The power of App Engine’s scalability was recently demonstrated; the Official Royal Wedding web site for the marriage between Prince William and Catherine Middleton generated over 15 million page views from 5.6 million visitors on the day of the wedding, despite the heavy load the web site (hosted on App Engine) ran smoothly [50].

Once an application is deployed on App Engine it’s possible to login to the admin console (not to be confused with the Django administration interface discussed earlier) and see how the application is doing. The admin console contains logs and graphs over resource usage, task queues, datastore etc.

It would be possible to create a scalable platform with an admin interface that provides similar features as that of the admin console in App Engine. But how much time would be required to setup this platform? Everyone knows that time is money; how much would it cost to setup this platform? In the end it is likely more cost- and time effective to use App Engine.

It should be noted that when dealing with very sensitive data the added cost of setting up the platform might be necessary.

In this project the following Google App Engine specific APIs where used: Channel, Memcache and Task Queue. All these APIs offer great functionality that significantly improves the application. At the same time they also make the application less portable. As discussed in section 2.2.2 it could be dangerous to become locked into a single provider’s technology. Keeping this in mind, it becomes difficult to decide whether these provider specific APIs actually improve the application or not. The bottom line is that developers have to be aware of this problem and consider backups beforehand.

For the application developed in this project no such backup was implemented or tested. However, as previously mentioned, Django-nonrel is used in this Django based application and the creators of Django-nonrel provide a template for writing non-relational Django back ends [46]. Meaning they have simplified the process of writing alternatives to App Engine’s datastore. Today the following alternatives exist: Cassandra [47], MongoDB [48] and SimpleDB [49]. Albeit these alternatives are in early development stages they do offer portability for Django applications.

Another thing to consider when using App Engine specific APIs are quotas. The most critical quota for this system is likely the number of channels created with the Channel API. The quota for free applications is 8640 channels created daily at a maximum rate of 6 creations/minute. With billing enabled the quota is 86 400 channels created daily at a maximum rate of 60 creations/minute.

This system connects parking houses with the Internet and then via the client application to a wide range of people. When designing this system a lot of effort went into making the system as flexible as possible when it actually is deployed in parking houses. The idea was to create the system independent of whatever conditions might exist at different parking houses. Furthermore the number of components was kept to a minimum to make installation and
maintenance easy.

4.2 Client application

By harnessing standard web technologies and PhoneGap it’s possible to rapidly develop cross-platform applications. Thereby the cost for development and maintenance can be drastically reduced compared to serving multiple native applications.

Another positive thing with this technique is that it is constantly evolving: HTML5 and CSS3 keeps adding more powerful functionality. Besides, there is already a great deal of talented web developers out there who can utilize their already existing skills instead of investing time learning new languages. There are a couple of web frameworks for smartphones that developers can use, for example: Sencha Touch [37] and jQuery Mobile [38]. The community around this technology seems to be steadily growing and with it the number of frameworks and other resources. Adobe® recently released an updated version of their web authoring software Dreamweaver® with built-in integration for PhoneGap and jQuery Mobile [51].

The target audience for this application is broad and includes users with all kinds of phones and phone subscriptions. It is therefore likely that a lot of these users don’t have unlimited data plans and thereby pays for the amount of data used. The application was developed with this in mind and efforts were made to minimize the amount of data.

The GUI of the smartphone application was designed to be simple, responsive and require little interaction from the user. Searching nearby the current position only requires one interaction after the application has started.

When developing complex games with intensive graphics it’s better to create native applications instead of cross-platform web applications. This is simply because native applications have better performance and for this type of applications performance is very important. Most other types of applications can with advantage be developed as web applications, to benefit from cross-platform deployment and simpler maintenance.
Chapter 5 – Discussion

When the development of the smartphone application started different frameworks that could simplify the development process where considered. Sencha Touch was quickly rejected because it appeared to have a rather steep learning curve. Someone with more previous experience of JavaScript coding could probably make great use of Sencha Touch though. After rejecting Sencha Touch the attention was switched to jQuery Mobile. Compared to Sencha Touch jQuery Mobile use more html elements and less JavaScript which felt less frightening. After investing a few weeks on jQuery Mobile, failing to create the desired layout without buggy behavior, this too was rejected. It should be noted that the framework at this time was still in alpha phase. In the end the application was developed with xui, a small library that handles cross-platform JavaScript issues, and pure CSS. From a learning perspective this was great, even though the development time could possibly be shortened by using a framework. With all the experience gained during this project Sencha Touch could be given another chance in the future.

In this thesis a smartphone application was chosen as the frontend for the system. This was a rather natural decision since a person who is searching for a parking spot is normally out and a lot of people always carry their phone. However it doesn’t necessary need to be the only frontend. As discussed in section 1.4 a variable message sign could also be a frontend that receives data from the same back end. In fact any kind of device that is capable of communicating over the Internet could act as frontend. That could include GPS-devices, car’s own built-in info systems and even a modern fridge. For the parking application discussed in this thesis all of these might not make sense but the possibility is there none the less. When considering devices one important thing is the fact that the application should not distract the driver. Therefor an audio-based complement to the visual frontend could be very valuable.

A lot of parking houses today are equipped with toll bars that are used in the payment process. These might enable usage of a simpler sensor than the one suggested previously, or even make additional hardware totally unnecessary. Another piece of equipment often found in parking houses is surveillance cameras. As discussed in section 1.4 it might be possible to use image analysis techniques instead of a sensor to count cars. Depending on the surveillance system it could even be possible to use that as data. Despite the fact that these are potential money-savers when deploying they were ignored because focus was on versatility. Meaning, the system should be compatible with any parking house regardless of existing equipment.
Chapter 6 – Conclusions

The combination of a cross-platform smartphone application and a scalable back end constitutes a very solid foundation for many kinds of applications. As a developer it is very motivating to know that the project you are working on is not restricted by otherwise common bottlenecks. The added motivation can also contribute to improving other aspects of the application. Even for a developer without previous experience it might be worth investing the time required to create this foundation for the motivation alone.

Throughout this thesis a solution to problems related to parking has been discussed. It was shown that many motorists spend a lot of time cruising around looking for available spots. Frustration grows when no available spot is found, leading to an increased disregard for traffic rules. It is assumed that these problems can be alleviated if motorists could get access to real-time information about where there are available spots.

In order to provide users with this information the following necessary components were identified: a back end for storing and distributing data, sensors to keep track of availability and a client application to make the data accessible to users.

Different back end solutions were discussed and their pros and cons compared. The scalability of the back end was assessed to be very important in order to avoid future problems if the application would become popular. In the end Google App Engine was deemed the most suitable option and thereby also used for the implementation. A flexible design that is independent of whatever conditions might exist at different parking houses was developed and successfully implemented.

An attempt to identify which operating system potential users have on their smartphones was made. No single operating system could be identified which is why a cross-platform approach was chosen. The cross-platform approach is based on an open source project called PhoneGap and web technologies. The implemented prototype application has been successfully tested on Android and iOS. The prototype can also be used on regular computers through a web browser. The prototype developed during this thesis and the 100’s of other applications developed with PhoneGap [39] confirm that it’s possible to create a user experience that is equal to that of native applications with web technologies.

Two different types of sensors where evaluated. Some desirable properties of the sensor were identified and finally a sensor that provides these is suggested.

To enable testing of the back end and the client application an interface for simulating sensors was also implemented. Tests show that the system works and can indeed provide real-time information about availability in parking houses. The system thereby fulfills the purpose of this thesis.
Chapter 7 – Future work

If more time would be invested in this system the first thing to do would be to replace the simulation with real sensors. To enable successful calibration and testing of sensors a pilot project with real cars should be setup. During the pilot project the system can receive input from users and be improved. The sensor and gateway suggested in this thesis should provide the necessary functionality to launch a pilot project.

The prototype application developed in this thesis is quite simple in terms of number of features. Many useful features could be added to further improve the application:

- Save a history of addresses and enable users to mark specific addresses as favorites to speed up searching by address.
- Enable users to get driving instructions to parking houses through Google Maps API.
- When searching for parking houses nearby a user’s current location the search could be automatically updated when the user has moved a specified distance from the original position. This could be implemented by monitoring the user’s position and calculating the distance to the original position.
- Enable customization of the application through settings. For example it would be useful to be able to configure the color-coded intervals. Currently the back end filters out all parking houses that are further away than 5 kilometers. It would make sense to let users choose this distance themselves via a setting.
- For those parking houses where the system keeps track of availability on each floor the application could show this data when the user approaches the parking house.
- Integrate payment solutions within the application to enable users to pay for the parking directly in the application.
References


