Design and implementation of new services for smart cities in Android

January 22, 2014

Author: Adrian Latorre Crespo
Director: Mónica Aguilar Igartua
Agradecimientos

Primero de todo me gustaría agradecer a Mónica y a su equipo por su total implicación con el proyecto, aportando ideas, sugerencias y contenido que me ha sido de gran ayuda. A continuación me gustaría agradecer a mis padres, mi hermano y abuelos. A mis padres por haberme ayudado en todo momento, dándome una excelente educación y apoyándome incondicionalmente a lo largo de toda esta larga carrera. A mi hermano por estar siempre de mi parte y ayudarme con las traducciones. Y a mis abuelos por sus sabios consejos. Me gustaría agradecer también a una persona muy especial en mi vida, que en ningún momento ha dejado de creer en mí y cuyo apoyo constante y alegría han sido vitales en todos estos años. Gracias Anna. También me gustaría agradecer a Dani, porque con nuestras conversaciones hemos aprendido mucho los dos y a Álvaro, porque aunque estés lejos ahora, las hemos pasado de todos los colores juntos. A todos mis compañeros del trabajo, gracias, por no dejar de animarme. Y a todo el resto que no puedo incluir por falta de espacio. Por último me gustaría agradecer a la Universitat Politècnica de Catalunya, por haberme enseñado tantas cosas a lo largos de estos años que voy apreciando cada vez más. No ha sido un camino fácil, pero desde luego ha valido la pena. A todos los nombrados anteriormente sólo puede decirles una cosa: Muchas gracias, de corazón.
## Vehicular Ad hoc Networks (VANETs) and SmartCities

### 1 Introduction to MANETs and VANETs

1.1 Brief Introduction to Mobile Ad-hoc Networks (MANET)

1.1.1 Properties

1.1.2 Challenges

1.2 Brief Introduction to Vehicular Ad-hoc Networks (VANET)

1.2.1 Vehicular Ad hoc networks (VANET)

1.2.2 Properties and challenges

### 2 SmartCities

2.1 Introduction to SmartCities

2.2 Wireless Sensor Networks (WSN)

2.2.1 Brief introduction to WSNs

### 3 Android devices as wireless nodes

3.1 Android devices

3.1.1 What makes Android Special?

3.2 Interaction between Android, MANETs/VANETs and Smart Cities

3.2.1 Android devices as VANETs nodes

3.2.2 Applications with Smart Cities' WSNs

## Design and implementation of the EMRISCO application

### 1 Overview

1.1 Introduction

1.2 Licensing

### 2 Maps

2.1 Google Maps API

2.1.1 Download and configure Google Play Services

2.1.2 Obtaining an API Key

2.1.3 Define the specifications in the ApplicationManifest

2.2 Adding a Map

2.3 Managing Locations

### 3 Markers

3.1 Parsing a document

3.1.1 Parsers

3.1.2 XML document

3.1.3 JSON document

3.1.4 Parsed outcome

3.2 Manipulating the Parking list

3.2.1 Markers

3.2.2 ListViews

3.2.3 Personalization

3.2.4 Listeners

3.3 Synchronization Awareness
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wireless Vehicular Adhoc Network</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>Characteristics and factors of a smart city</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Wireless Sensor Networks</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Waze Map with real time information about the traffic</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>SHA-1 fingerprint of our application</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Simple map inserted</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Markers on Android</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>ListView + Map</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>Markers and ListView personalized</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>Google Cloud Messaging Architectural Overview</td>
<td>38</td>
</tr>
<tr>
<td>11</td>
<td>Design of the EMRISC application</td>
<td>39</td>
</tr>
<tr>
<td>12</td>
<td>GCM Connection Server differences by Google Android Developers</td>
<td>40</td>
</tr>
<tr>
<td>13</td>
<td>HTTP GCM Server</td>
<td>41</td>
</tr>
<tr>
<td>14</td>
<td>GCM Notification example</td>
<td>46</td>
</tr>
<tr>
<td>15</td>
<td>Sharing Notifications</td>
<td>48</td>
</tr>
<tr>
<td>16</td>
<td>A flow summary</td>
<td>50</td>
</tr>
</tbody>
</table>
## Nomenclature

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>app</td>
<td>application</td>
</tr>
<tr>
<td>EMRISCO</td>
<td>Emrisco rules</td>
</tr>
<tr>
<td>GCM</td>
<td>Google Cloud Messaging</td>
</tr>
<tr>
<td>GNU</td>
<td>Recursive acronym for 'GNU’s Not Unix'</td>
</tr>
<tr>
<td>ICT</td>
<td>Information and Communication Technologies</td>
</tr>
<tr>
<td>LTE</td>
<td>Long Term Evolution</td>
</tr>
<tr>
<td>MANET</td>
<td>Mobile Ad hoc Network</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-peer</td>
</tr>
<tr>
<td>VANET</td>
<td>Vehicular Access Network</td>
</tr>
<tr>
<td>Wi-Fi</td>
<td>Wireless Fidelity</td>
</tr>
<tr>
<td>WPA2</td>
<td>Wi-Fi Protected Access II</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
</tbody>
</table>
Abstract

Information and Communication Technologies have become more and more important over the last years and Smart Cities are nowadays one of the main fields of research in the world. Mobile ad hoc networks and wireless sensor networks presence is increasing and vehicular-oriented variations are progressively becoming more significant. But the lack of adapted vehicles slow down the development of these technologies and reduce their potential. On the other hand, the number of mobile devices is in constant growth and in 2013, more than 81% of the world’s smartphones run an Android operating system. One logical solution is to use these mobile devices to provide different kind of services for the Smart Cities.

In our project we have designed an Android application that offers two services for a Smart City: a real-time parking spot finder and an emergency instant notifier. Our application will act as a node of a VANET, and will connect to other devices using different types of communication such as Wi-Fi Direct or Google Cloud Messaging. We have also designed an architecture of notifications that, in case of emergency, sends a message via push to all the users of the application, notifying them of the event. This message, in turn, can be forwarded using any kind of messaging application. Moreover, as new potential services appear each day, we have also adapted the application so it will suit completely to future different purposes.
Resumen

Las Tecnologías de la Información y Comunicación han ido cobrando cada vez más importancia a lo largo de los últimos años y las Smart Cities (Ciudades Inteligentes) son hoy en día uno de los principales campos de investigación en el mundo. Así mismo, la presencia de Mobile ad hoc networks y de Wireless Sensor Networks es cada vez más significante. Pero la falta de vehículos adaptados ralentiza el desarrollo de estas tecnologías y reduce su potencial. Por otro lado, el número de dispositivos móviles está en constante crecimiento y en el año 2013, más del 81% de los smartphones del mundo utilizaban el sistema operativo Android. Una solución lógica es usar estos dispositivos móviles para ofrecer diferentes tipos de servicios a las Smart Cities.

En nuestro proyecto hemos diseñado una aplicación Android que ofrece dos servicios para una Smart City: un buscador en tiempo real de plazas de aparcamiento y un notificador instantáneo de emergencias. Nuestra aplicación actuará como nodo de una VANET y se conectará a otros dispositivos utilizando diferentes tipos de comunicación tales como el Wi-Fi Direct o Google Cloud Messaging. También hemos diseñado una arquitectura de notificaciones que, en caso de emergencia, envía un mensaje vía push a todos los usuarios de la aplicación, notificándolos del evento. Este mensaje puede, a su vez, ser reenviado utilizando diferentes tipos de aplicaciones de mensajería. Incluso, ya que aparecen cada día nuevos usos potenciales, hemos adaptado la aplicación para que se ajuste completamente a cualquier propuesta futura.
Resum

Les Tecnologies de la Informació i Comunicació han anat cobrant cada vegada més importància al llarg dels últims anys i les Smart Cities (Ciutats Intel·ligents) són avui dia un dels principals camps de recerca al món. Així mateix, la presència de Mobile ad hoc networks i de Wireless Sensor Networks és cada vegada més significant. Però la falta de vehicles adaptats ralenteix el desenvolupament d’aquestes tecnologies i rebateix el seu potencial. D’altra banda, el nombre de dispositius mòbils està en constant creixement i l’any 2013, més del 81% dels smartphones del món utilitzaven el sistema operatiu Android. Una solució lògica és usar aquests dispositius mòbils per oferir diferents tipus de serveis a les Smart Cities.

En el nostre projecte hem dissenyat una aplicació Android que ofereix dos serveis per una Smart City: un cercador en temps real de places d’aparcament i un generador de notificacions d’emergències. La nostra aplicació actuarà com a node d’una VANET i es connectarà a altres dispositius utilitzant diferents tipus de comunicació tals com el Wi-Fi Direct o Google Cloud Messaging. També hem dissenyat una arquitectura de notificacions que, en cas d’emergència, envia un missatge via push a tots els usuaris de l’aplicació, notificant-los de l’esdeveniment. Aquest missatge pot, al seu torn, ser reexpedit utilitzant diferents tipus d’aplicacions de missatgeria. Fins i tot, ja que apareixen cada dia nous usos potencials, hem adaptat l’aplicació perquè s’ajusti completament a qualsevol proposta futura.
Part I

Vehicular Ad hoc Networks (VANETs) and SmartCities

Before introducing our project, we want to introduce Vehicular Ad hoc Networks (VANETs) and their current interactions with Wireless Sensor Networks (WSNs), whose presence is becoming more and more important in Smart Cities.

1 Introduction to MANETs and VANETs

One of the most significant facts over the recent years is the growth of the mobile sector services (a constant growth of 14% over the years) and a total number of 4 billion of subscribers is expected by 2017. Mobile communications are applied everywhere, in our daily routines and realize an infinity of interactions with other elements. Usually, there are two main ways of communication between two wireless mobile devices:

- **Infrastructure or Centralize Network:** Wireless mobile networks have traditionally been based on the cellular concept, where all devices are connected to a central node which is the acting agent for all communications, and relied on good infrastructure support. Typical examples of this kind of wireless networks are GSM, UMTS, WLAN, etc.

- **Infrastructure-less Network:** This mobile wireless network is commonly known as a Mobile Ad hoc networks or MANETs. A MANET is a collection of wireless nodes that can dynamically be set up anywhere and anytime without using a pre-existing network infrastructure. It is an autonomous system in which mobile hosts connected by wireless links are free to move randomly and often act as routers at the same time.

During the last years, MANETs have been becoming more and more popular and nowadays is one of the most vibrant research fields. In this section, we want to realize a brief introduction to MANETs and VANETs, talk about their properties and discuss some possible challenges. Papers and articles from D. Djenouri[20], C. E. Perkins[37] or D.B Johnson[25] do a more deep study on these type of networks.

1.1 Brief Introduction to Mobile Ad-hoc Networks (MANET)

A Mobile Ad hoc Network or MANET is an assemblage of wireless devices that can form a dynamic network. This devices (or nodes) can change their position and are capable of reorganize themselves in order to make a communication succeed.

1.1.1 Properties

The MANETs have a very specific list of properties common to all of them:

- **Distributed Operation:** As there is no central or main node in the network, the control management of it is distributed among the devices.

- **Dynamic Network Topology:** We have said that the elements of the network can change their position, hence, the topology of it changes randomly with time. MANETs have to adapt to these changes assuring a minimum of communication quality.

- **Multi-Hop Routing:** In a source to destination communication, the packets have to do multiple hops between intermediate nodes. A high number of hops induce higher error rates as the conditions of the connection deteriorates.

- **Autonomy and Self-Organization:** In a MANET, each device is an autonomous node, functioning as a host and a router (receiving and forwarding packets). They are also responsible of discovering near nodes to connect to.

---

• Light terminals: Usually, the MANET nodes are mobile devices with limited characteristics: small memory, small processing capacity an low power storage. The communications need to be optimized in order to match the limitations.

1.1.2 Challenges

Apart from their properties, the MANETs have several advantages and disadvantages that must be studied.

Scalability Weakness A MANET can have a very large number of nodes that can move. To find a good route to a destination requires a very high rate of exchange of information among the nodes, and thus a high amount of traffic, that will augment with every new node added. We have to be careful with overdimensionating the MANET.

Security Security is one of the main concerns of MANETs. Normally, in a network the security is centralized in a reference node, which manages all the privileges. This way, the rest of the nodes can operate without worrying so much about security due to the fact that they don’t contain valuable information. In MANETs, as mentioned before, every node needs to be able to operate autonomously, hence, needs to have all the information. So, an inefficient authentication protocol in these Ad hoc networks can lead to the vulnerability of it.

Routing Routing in Ad hoc networks is different from normal or traditional IP routing, as we have the problem of facing to a dynamic network. Due to the autonomy of their nodes, each one of them needs to know at least the reachability information of its neighbors in order to determine a packet route. But these neighbors can move in a short time and thus, the route may not be available anymore. Finally, the routing solutions for Ad hoc networks are classified in two proposals: proactive and reactive routing. Proactive or table-driven protocols are inspired by IP routing protocols and consist in defining and maintaining a stable route to send data to any node of the network. In the other hand, reactive routing proposes algorithms that research viable routes between two nodes when a request is received. It tries to keep these routes as updated as possible in order to minimize errors. Finally, there are some hybrid proposals that include both protocols features.

Quality of Service (QoS) In a MANET we can have different kinds of traffic that requires certain characteristics such as a minimum of duration, bandwidth, stable end-to-end, number of interactions, etc. So, a uniform packet processing is not optimal, as each kind of traffic requires a different QoS. The Ad hoc network must adapt in order to assure these different values, taking in consideration the continuous changes of topology and the limited resources.

Interoperation A not very much researched topic is the interoperation: the fact of bringing together two different MANETs. This can cause several problems such as false neighbors detection, security problems or routing.

Energy-Constrained Operations As we said before, some or all the nodes in an Ad hoc network are mobile, and so, may rely on batteries/exhaustible resources. Therefore, the conservation of energy is a very important MANET research matter. This can be achieved by developing better batteries or reducing the energy consumption of the nodes.

1.2 Brief Introduction to Vehicular Ad-hoc Networks (VANET)

After a brief introduction to MANETs, let us now talk about traffic, and in particular, traffic management. When we talk about controlling traffic, we are picturing a centralized system: cameras and sensor installed everywhere transmitting data to a central station where it is processed. This system has proven good results but has an important lack: there is a time gap between the time of the event and the time of receiving the information. However, following the aforementioned growth of communications, a new architecture based on vehicle-to-vehicle communications has emerged.

---

1.2.1 Vehicular Ad hoc networks (VANET)

Vehicular Ad hoc networks, also called VANETs, are a variation of the previously defined Mobile Ad hoc Networks (MANETs) adapted to vehicles. The aim of the VANETs is to enable a communication between nearby cars and elements of the road, providing information about the traffic or safety advises as well as other services. For example:

- A broadcasted warning message informing about a near accident.
- A guidance system to inform about locations or new parking spots.
- Broadcasting multimedia services

But, as the MANETs, the VANETs also have different properties to take in consideration.

1.2.2 Properties and challenges

As we have defined, a VANET is a specific kind of MANET and so, share some of the properties of them. But no all the characteristics of a MANET are shared by the VANETs due to its characteristics: VANETs nodes move quicker, have not any limitation of energy sources and have more processing capabilities. But, they also have some constraints such as radio obstacles (buildings, multi-path, fading effects). Let us take a more careful look at those issues.

- Processing and Energy: As vehicles have their own power source, we can have more powerful processors and the limitation of energy is not a problem anymore.
- Environment: Usually, MANETs are designed for indoors. In VANETs we will face a lot of problems in consequence of the presence of buildings that will cause multi-path or fading problems.
- Network Topology: As vehicles move at higher speeds, a device can now quickly join or leave a network in a very short time, and so, the topology changes with a higher frequency. This will cause that we will have to update the routing table more often. Also, in VANETs if there isn’t a direct route, a packet is carried by a node until it could be forwarded to a node being closer to the destination.

In general, due to these characteristics, the Vehicular Ad hoc Networks are focused on broadcasting one message to different sources rather than establishing a long communication between two particular nodes.
2 SmartCities

2.1 Introduction to SmartCities

Over the last years, the importance of Information and Communication Technologies (ICTs) has been growing. The ICT is the result of integrating all the infrastructures of telecommunications, devices, technology and software in order to improve the user access to the information. ICTs have a big impact in three key areas: productivity and innovation, modernization of public services and advances in science and technology. And so, when we bring together these ICTs and a city, a new concept is born: a smart city.

A city becomes ‘smart’ when its aim is to integrate traditional and modern (ICT) infrastructures to improve the quality of life of its citizens. It does not respond to only a certain aspect, but more to a concept depending on the field. In business terms, a ‘smart city’ is a city with ‘smart’ industry that applies ICT. But we can also refer to the education field: a smart city has ‘smart’ inhabitants. We can even find concepts of ‘smart’ politics (e-governance, e-democracy) or ‘smart’ mobility. To sum up, we have six characteristics of a smart city:

- Smart Economy
- Smart People
- Smart Governance
- Smart Mobility
- Smart Environment
- Smart Living

And so, a ‘smart’ city is a city that is performing well in these six characteristics.

2.2 Wireless Sensor Networks (WSN)

One of the aforementioned characteristics of ‘smart’ cities is the concept of Smart Mobility that relies on the so-called Wireless Sensor Networks (WSN).

2.2.1 Brief introduction to WSNs

In recent years, advances in wireless networking, micro-fabrication and integration embedded microprocessors have unlocked a new possible technological concept: wireless sensor networks. WSN consist of a large number of sensor nodes which collect data and interoperate each other to carry out functions involving some kind of tracking, monitoring or controlling. WSNs are considered as a special case of a MANET with reduced or no mobility.

Its operation is very simple: a node is a group of sensors that gather and process data. Once the information is collected, it is sent to the other nodes, traveling through all the network using wireless links and multi hop routing. Usually, sensor nodes are organized in clusters to reduce power consumption because the amount of energy used to transmit a message from a sensor to the clustering node is lower than the one that would be necessary to transmit the same message from the sensor nodes to the base station in a single hop. Thus, most common WSNs use the sequence presented in Figure 3 on page 18 to process data obtained by its sensor.

A WSN has a few properties that we have to take into consideration:

- Failure tolerant: In WSNs, when a node stops working it doesn’t affect the rest of the network because each node works independently and collects very similar data to the adjacent nodes.
- Deployment: A WSN can be deployed without a planning or a previous configuration, reducing costs and time.
- Networking and security: Nodes have very limited resources of energy and processing capabilities, thus they will run very simple versions of routing protocols and security implementations.

3 Android devices as wireless nodes

Now that we have defined what are the Smart Cities, the WSNs and the MANETs/VANETs, let us inquire about their interactions and define our objective.

3.1 Android devices

In spring of 2009, a new open source operating system for mobile devices is created by Google and the Open Handset Alliance: Android by Google. Nowadays, Android’s share of the global smartphone market has reached 81% and has become the most popular operating system in the world.

3.1.1 What makes Android Special?

There are many mobile platforms on the market today, including iOS, Windows Phone, Blackberry or Symbian, but Android has certain characteristics that makes it special. Android is the first OS that combines:

- A truly open, free development platform based on Linux: Developers like it because it is a platform based on something that they know and can adapt. Configurations and applications that work on Linux will work as well on Android. But also, as it is a free and open source

---

\[4\] Android by Google http://android.com

operating system, any handset maker can install it in their product. Some interesting independent handset makers as Geeksphone\textsuperscript{6}, Meizu\textsuperscript{7} or Oppo\textsuperscript{8} have emerged, providing very low cost devices.

- **Built-in Services**: Android comes with Google Apps preinstalled, a collection of built-in services that work out-of-the-box. These services provide us readings of different sensors (dual axis gyroscope, altimeter, accelerometer, compass) as well as location related logs (GPS, network triangulation).

- **High end devices**: Latests Android devices are very powerful machines. As a reference, the last device of Google, the Nexus \textsuperscript{9} has a quad-core 2.26GHz processor built-in and a battery of 2300 mAh (standby time up to 300 hours). They are also very compact devices: 5 inches and 130g.

- **New technologies**: Android newer devices come with the latest technological equipments. Android supports Near Field Communications\textsuperscript{10} (NFC), Dual-Band Wi-Fi (2.4/5 GHz) 802.11 a/b/g/n/ac, Wireless Charging, 4G LTE communications, Bluetooth 4 and Wi-Fi Direct communications.

- **Millions of applications and devices**: As May 2013, 48 billion\textsuperscript{11} apps have been installed from the Google Play. As September 2013, 1 billion Android devices have been activated\textsuperscript{12}. These numbers offer a big marketplace for every application and a high number of potential users.

3.2 Interaction between Android, MANETs/VANETs and Smart Cities

In the previous introduction to VANETs we pointed out that the main purpose of those vehicular wireless networks is to provide instant information to the drivers. However, one of the main problems VANETs have to face is that there is not enough cars with VANET-enabled protocols installed.

\textsuperscript{6}Geeksphone http://geeksphone.com
\textsuperscript{7}Meizu http://en.meizu.com
\textsuperscript{8}Oppo http://en.oppo.com
\textsuperscript{9}Google Nexus 5 http://google.com/nexus/5
\textsuperscript{10}Near Field Communications http://nfc-forum.org
\textsuperscript{11}The Verge http://theverge.com/2013/5/15/4333584/total-android-activations-900-million
\textsuperscript{12}Vic Gundotra, Senior Vice President for Google http://plus.google.com/+VicGundotra/posts/8CVJ79nPQwN
3.2.1 Android devices as VANETs nodes

One straightforward solution to the problem is to use our Android devices as a node of the VANET. Placing a device in a car connected to its power source, creates an intelligent and fully capable of communicating node. It supports every kind of communication type (Wi-Fi (2.4/5 GHz) a/b/g/n/ac, Bluetooth 4.0, NFC, Wi-Fi Direct, LTE, GSM) and has high processing capabilities. In fact, starting in 2014, Google has formed the Open Automotive Alliance (OAA), a global alliance of technology and auto industry leaders committed to bringing the Android platform to cars.

3.2.2 Applications with Smart Cities’ WSNs

With a potential network of millions of nodes, we can start to think of multiple applications that will use all the information of the Smart Cities’ WSNs.

Parking spots One of the main problems nowadays in a city is to find a parking spot. If our smart city has sensors on each spot, we can easily inform the drivers about free nearby places where to park. More and more cities are starting to roll out smart parking sensors and are making this data open and free to access. Applications as Apparcar or Parker use this open data to guide you.

Information about traffic Monitoring the speed of the cars, we can easily detect the traffic status of a particular region. Waze, purchased by Google, collects this kind of data and creates dynamic maps of the traffic status (Figure 4).

Figure 4: Waze Map with real time information about the traffic

---

13Open Automotive Alliance http://openautoalliance.net
15Apparcar App http://aparcar.com
16Parker by Streetline http://theparkerapp.com
17Waze App http://waze.com
Intelligent traffic lights  Information about an approaching ambulance could be transmitted to an intelligent traffic light and it could switch so the emergency vehicle doesn’t have to stop.

Accident detection  A device in a car could detect a sudden change of speed and interpret it as an accident. Then, it would broadcast the information to all the nearby cars, providing instant awareness of it and preventing collateral damages. The message would propagate through all the VANET, informing far vehicles of the event and offering the possibility to change their route.

The aim of our project is to design an Android application that can provide services (parking spots and accidents notification) to vehicles in a smart city environment. The application will be part of a VANET and we will simulate the protocols used in a vehicular ad hoc network with some of the actual technologies that an Android phone has.

19 Chris Thompson and Jules White and Brian Dougherty and Adam Albright and Douglas C. Schmidt, "Using Smartphones to Detect Car Accidents and Provide Situational Awareness to Emergency Responders" (2011).
Part II
Design and implementation of the EMRISCO application

1 Overview

1.1 Introduction

The main objective of the EMRISCO application is to provide an example of possible new services for smartphones in smart cities. In our application we are going to implement the following:

- A localization of free parking spots with GPS guidance to them. The android device fetches a document (XML or JSON) with all the available parkings, parses it and places in a map and a list all of them ordered by type and distance. This service includes the personalization and filtering of the results by different criteria. When the user picks a parking spot, a GPS application is opened to help the user to reach the selected destination.

- An emergency notifications system where the users generate content that is distributed via push to all the users. A smart user witnesses an event (accident, traffic jam, ...) and notifies it to our EMRISCO server. The event is shared via push notification to all the EMRISCO users situated in a defined radius. The final user can, in turn, share the message via different ways (social networks, mail, phone, sms) to enhance its diffusion.

Both of these services can be personalized to fulfill different purposes and have been designed to work in Android smartphones since version 2.3 Gingerbread (90% of all of them).

1.2 Licensing

The EMRISCO application is open sourced and is subjected to the GNU General Public License\(^\text{20}\). This means that you can distribute it, modify it, use it privately, with commercial purposes or apply to a patent grant. The software author cannot be held liable for damages and you may not grant a sublicense to modify and distribute this software to third parties not included in the license. In every case, it is required that the source code is made available when the software is distributed, a copy of the license and copyright must be in the code and the significant changes made to the code must be indicated.

We have chosen this license due to the fact that we are using services, which in turn are subjected to CreativeCommons licenses.

2 Maps

In our application we are going to use Google Maps to place markers on the free parking spots. In order to be able to use the Google Maps in an Android application, we are obliged to use their API. In this section, we are going to explain how to configure the Google Maps API and how to add a map to our application. Finally, we are going to retrieve our location and place it on the map.

2.1 Google Maps API

The steps that we are going to follow are:

- Download and configure Google Play Services
- Obtain an API key
- Define the specifications in the ApplicationManifest
- Add a map to our application

\(^{20}\)GNU General Public License v2 http://www.gnu.org/licenses/gpl-2.0.html.
2.1.1 Download and configure Google Play Services

After downloading the Android SDK\footnote{Google Android SDK: https://developer.android.com/sdk/index.html}, we will launch the SDK Manager to download both, an Android API superior to Android 4.2.2 and the Google Play services package. We are doing this because the Google Maps API is distributed as part of the Google Play Services, and therefore, we need both of them to be installed to be able to continue. Once we have downloaded everything, we will import it as a library to our project.

2.1.2 Obtaining an API Key

When we call an API from Google, we are required to provide a unique project identifier. This enables Google to tie the requests to difference projects in order to monitor traffic or handle billing. Hence, to be able to use the Google Maps API in our app, we need to add a simple API key to our application. This key will be linked exclusively to our app. In order to obtain an API key we need to provide the SHA1-fingerprint of our application. It can be obtained easily in Eclipse->Window->Preferences->Android->Build (Figure 5).

![Figure 5: SHA-1 fingerprint of our application](image)

Once we have our SHA-1 fingerprint, in the Google APIs console\footnote{Google API console: https://cloud.google.com/console} we introduce the fingerprint and the name of our application and we will obtain the new API key.

2.1.3 Define the specifications in the ApplicationManifest

Now that we have our API key, we need to insert it in our application. There are three blocks that need to be modified.

1. In the AndroidManifest.xml we have to add some code as a child of the `<application>` block:

   1. `<meta-data`
   2. `android:name="com.google.android.maps.v2.API_KEY"
   3. `android:value="API_KEY"/>`

   where ‘API_KEY’ is our new API key.

2. The following permissions are needed:

   (a) `android.permission.INTERNET`: Used by the API to download the maps from the servers.
(b) `android.permission.ACCESS_NETWORK_STATE`: Allows the API to check the network status to download data.

(c) `com.google.android.providers.gsf.permission.READ_GSERVICES`: Allows the API to access to the Google servers.

(d) `android.permission.WRITE_EXTERNAL_STORAGE`: Allows the API to save cached data in the external storage unit (sd-card).

(e) `android.permission.ACCESS_COARSE_LOCATION`: Allows the API to use WiFi or mobile data (or both) to obtain our location.

(f) `android.permission.ACCESS_FINE_LOCATION`: Allows the API to use the GPS to obtain our location with precision.

3. Finally, Google Maps uses OpenGL ES v2 to render the maps. If we don’t have it installed, the map will not be displayed and so, it is recommended to add the condition:

```xml
<uses-feature
    android:glEsVersion="0x00020000"
    android:required="true"/>
```

We can now start to use Google Maps in our application.

### 2.2 Adding a Map

Adding a map in our application is very simple. We just have to add this code in the main `onCreate()` method.

```java
mapFragment = ((SupportMapFragment) getSupportFragmentManager().findFragmentById(R.id.map));
map = mapFragment.getMap();
```

And the application will show us a fullscreen map as shown in Fig: 6. Once we have our map, what we want to do now is to center it on our location.

![Simple map inserted](image)

Figure 6: Simple map inserted

### 2.3 Managing Locations

There are a few ways to obtain our location in Android and also Google suggests us what is the best strategy on how to use them. We need to keep in mind one very important thing: the battery consumption. When an application that uses the user’s location is developed, a decision on how this position is obtained needs to be done. We can use the GPS and obtain a coarse location in

---

exchange of using more battery or just triangulate the location using the WiFi network or the 3G network.

Following Google’s suggestions, we have developed a well-balanced solution to obtain the user’s location and then, centered the map’s position on it. All the code can be consulted on the Annex: [VI on page 57](#). Let’s analyze what we have coded (all the code is located in the MainActivity):

- First we add a few variables that we will use: the SupportMapFragment that we have talked about in the last subsection, a GoogleMap object and a LocationClient object. Also, we define a maximum time that we will wait until we decide that the connection has failed (CONNECTION_FAILURE_RESOLUTION_REQUEST). The block [public static class ErrorDialogFragment](#) shows us a message informing us that the connection could not be established.

- In the onCreate method, we instantiate the LocationClient object and activate the localization on the map. When the Activity resumes from a paused status, we call the onStart method and we check if GooglePlay services are available. In the same way, when we exit our application we call the onStop method to disconnect us from everything.

- In the onActivityResult, after receiving an OK status from the onStart method, we initiate the connection to GooglePlay services.

- Finally, in the onConnected method we define what we have to do once we are connected. In our case, we show a toast on screen telling that the connection has finished successfully and then we center the map in our position using the following code:

```java
Location location = mLocationClient.getLastLocation();
LatLng latLng = new LatLng(location.getLatitude(), location.getLongitude());
CameraUpdate cameraUpdate = CameraUpdateFactory.newLatLngZoom(latLng, 17);
map.animateCamera(cameraUpdate);
```

This code obtains the last known location, extracts the latitude and longitude and sets a CameraUpdate object to that position with a determined zoom level. Finally, we tell the map to show the view of that CameraUpdate object. It is very interesting to take a moment and investigate all the possibilities and movements one can do with the Camera in Google Maps: it is possible to change the position, the zoom level, the orientation and even the angle. There is a lot more information in the Changing View section.

Now that we have our map, the next step is to place markers in it.

### 3 Markers

One of the purposes of this application is to show on a map different kinds of parking spots. I am going to describe two ways of doing this in Android and in the Future Implementations’ section discuss a couple of ways to improve them. In this section we are going to:

- Parse a file (XML or JSON) to create a list of parkings that can be manipulated
- Sort this list and display it in two ways: graphically and textually
- Personalize the elements (color, distance)
- Adding responsiveness with Action Listeners
- Optimize the code to avoid synchronization problems

#### 3.1 Parsing a document

In this subsection, we are going to explain what ‘parsing’ a document means and why do we do it. Also, we are going to parse two different kinds of documents, an XML document and a JSON document. Finally, we are going to analyze of the outcome.

---

24Google Developers, Changing the View of the Camera, [https://developers.google.com/maps/documentation/android/views](https://developers.google.com/maps/documentation/android/views)
3.1.1 Parsers

The parsing of a document is the process of analyzing a document or a string, detecting its structure and transforming it to another language or structure following a determined set of rules. As Aho and Ullman explain in their book\(^{25}\) when we talk about parsing we are referring to the computational method of analyzing a group of related tokens \(<\text{words}>\) as an arithmetic expression in order to convert it into structured representations such as expression trees.

In our case, our objective is to analyze a given document and extract four main attributes: Name of the parking, Latitude, Longitude and Status (free/full). In order to do this, we have to define a new class for the ParkingMarker object (Annex VI on page 61).

3.1.2 XML document

We are receiving an XML document (Annex VI on page 60) constituted of nine blocks of parkings. Each block of parking has the following structure:

```
<parking>
  <name>Parking UPC</name>
  <lat>41.387179</lat>
  <lng>2.1125849</lng>
  <free>0</free>
</parking>
```

In Android we can parse an XML document in a lot of ways: XML Pull Parser\(^{26}\), DOM\(^{27}\) or SAX\(^{28}\) are the most popular amongst all. After evaluating which one is more suitable for our project, we have decided to use the SAX method.

Parser Handler

As we start parsing our a document, certain events take place. For example, when we have reached the end of a tag or the end of the document. The five main events that we can face are:

- `startElement()`: start of an XML tag.
- `endElement()`: end of an XML tag.
- `characters()`: text fragment.

In our code defined in Annex VI on page 62 we handle each one of the situations. In `startDocument()` we define a new array of ParkingMarker objects, in `startElement()` we precise that if the tag equals to `<parking>`, then a new object is created. And finally, in the `endElement` we assign the values:

- If I was in the `<name>` tag, set this value to the .name field of my ParkingMarker object.
- If I was in the `<lat>` or `<lng>` tags, set the corresponding fields of the ParkingMarker object.
- If I was in the `<free>` field, if its value is “1” set a TRUE statement, if its value is “0” a FALSE one and if its none of these, define a parse error.

```java
public void endElement(String uri, String localName, String name)
    throws SAXException {
    super.endElement(uri, localName, name);
    double value;
    if (this.parkingActual != null) {
        if (localName.equals("name")) {
            parkingActual.setName(sbText.toString().trim);
        } else if (localName.equals("lat")) {
            value = Double.parseDouble(sbText.toString().trim);
            parkingActual.setLat(value);
```
```java
if (localName.equals("lng")) {
    value = Double.parseDouble(sbText.toString().trim);
    parkingActual.setLng(value);
} else if (localName.equals("free")) {
    value = Double.parseDouble(sbText.toString().trim);
    if (value==1) parkingActual.setFree(Boolean.TRUE);
    else if (value==0) parkingActual.setFree(Boolean.FALSE);
    else {
        // NOT CORRECT STRUCTURE
        parsingError = true;
    }
} else if (localName.equals("parking")) {
    parkings.add(parkingActual);
}
```

The Parser

Finally, the parser is a standard SAXParser with our personalized handler. The most important part of the code is these few lines of code where we define how it has to behave:

```java
SAXParser parser = factory.newSAXParser();
ParkingHandler handler = new ParkingHandler();
parser.parse(this.getInputStream(), handler);
return handler getParkings();
```

with parser.parse(this.getInputStream(), handler); we are telling the parser to parse the file that it is receiving using our handler and then with handler.getParkings we are starting the parsing process.

3.1.3 JSON document

Another type of document that we can receive is a JSON document. While XML works well in many scenarios it has its disadvantages in others, for example in Ajax-style web. And even if most browsers can send and parse XML files, JSON provides a lighter and more standardized way to exchange data, ideal for mobile applications.

A `<parking>` object in JSON will be as following:

```
{
    name: "Parking UPC,
    lat: 41.387179,
    lng: 2.1125849,
    free: True
}
```

Obtain a JSON document

To get the JSON document in Android, we can use a simple HTTP request:

```java
DefaultHttpClient httpclient = new DefaultHttpClient(new BasicHttpParams());
HttpPost httppost = new HttpPost("http://someJSONUrl/jsonWebService");
// Depends on your web service
httppost.setHeader("Content-type", "application/json");
InputStream inputStream = null; String result = null;
try {
    HttpResponse response = httpclient.execute(httppost);
    HttpEntity entity = response.getEntity();
    inputStream = entity.getContent();
    // json is UTF-8 by default
    BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream, "UTF-8"), 8);
    StringBuilder sb = new StringBuilder();
    String line = null;
    while ((line = reader.readLine()) != null) {
        sb.append(line);
    }
    result = sb.toString();
```

Dave Romero, JSON (Javascript Object Notation).
Microsoft Developer Network, "An Introduction to JavaScript Object Notation (JSON)".
```java
    sb.append(line + "\n");
    }
    result = sb.toString();
}
catch (Exception e) {
    // Oops
}
finally {
    try{
        if(inputStream != null) inputStream.close();
    }catch(Exception squish){}
}
```
3.2.1 Markers

Adding a marker in the map is very simple. We just have to add the following detailed code:

```java
// Go through each item on the list
Iterator<ParkingMarker> i = parkings.iterator();
// for (ParkingMarker parking : parkings) {
while (i.hasNext()) {
    ParkingMarker parking = i.next();
    aLat = parking.getLat();
    aLng = parking.getLng();
}
// adding a new marker on the map
Marker marker = mMap.addMarker(
    new MarkerOptions()
        .position(new LatLng(aLat, aLng))
        .title(parking.getName())
        .icon(BitmapDescriptorFactory.defaultMarker(markerColor)));
// adding event to HashMap
eventMarkerMap.put(marker, parking);
```

After running our application with the new changes, we obtain the Figure 7.

![Figure 7: Markers on Android](image)

3.2.2 ListViews

The other option is to obtain those parkings in a list form. We decide to divide the screen in half to display both views at the same time. We have followed the instructions of Androideity and Mbii to distribute the weights of the layers evenly in order to have a top and a bottom sections and everything is defined in the final source code.

**Defining a List Adapter** To be able to display a list in Android, we need an adapter. This is due to the fact that Android doesn’t know what kind of objects we have in our array. We have to implement a new class that extends ArrayAdapter to define it (Annex VI on page 63). The interesting part is the getView() method. Here is where we are going to define the kinds of content that we are going to insert and where do we want to put them.

---

32 Condesa, Ul flúidas y la propiedad weight en Android, Androideity (2012).
33 Ugía, Jose Luis, Android, LinearLayout Distribution Explained (weight and sizes).
We assign a position of the list (R.id.LblTitle) to a variable (lblTit).

```java
// We assign a position of the list (R.id.LblTitle) to a variable (lblTit)
TextView lblTit = (TextView)item.findViewById(R.id.LblTitle);
// We set in this position the data from our array
lblTit.setText(parkings_data.get(position).getName());
```

We repeat this process with all the variables of our ParkingMarker object.

**Adding the code** To put everything together, we need to do two extra steps.

1. We create three private variables that can be accessed from every method of the class: lstOptions, adapter and parkings_data.

```java
private ListView lstOptions;
private List<ParkingMarker> parkings;
private List<ParkingMarker> parkings_data;
ParkingAdapter adapter;
```

2. In the onCreate() method we assign the adapter defined in the previous section

```java
// We create the parking list object
parkings_data = new ArrayList<ParkingMarker>();
// We copy all the parks obtained from the parsing
parkings_data = parkings;
// We set the adapter and the options
adapter = new ParkingAdapter(MainActivity.this, parkings_data);
lstOptions = (ListView) findViewById(R.id.LstParkings);
lstOptions.setAdapter(adapter);
```

We have now our list of parkings as shown in Figure 8.

![Figure 8: ListView + Map](image)

### 3.2.3 Personalization

Now that we have everything working, let’s personalize a bit the markers and elements of the list.
Distance  We have decided to add a distance field to each parking corresponding to the distance between the user and the parking itself. To calculate the distance between two points on Earth’s surface we can use the Haversine formula:

\[
d = 2r \arcsin \left( \sqrt{\sin^2 \left( \frac{\Delta \varphi}{2} \right) + \cos(\varphi_1) \cos(\varphi_2) \sin^2 \left( \frac{\Delta \lambda}{2} \right)} \right)
\]

(1)

However, for small distances and as the accuracy is not very important (it doesn’t matter that the parking is situated a few meters nearer), we can use Pythagora’s theorem on an equirectangular projection:

\[
\begin{align*}
x &= \Delta \lambda \cos(\varphi) \\
y &= \Delta \varphi \\
d &= r \sqrt{x^2 + y^2}
\end{align*}
\]

This uses only one trigonometric and one sqrt function to obtain the same results. We now have to adapt the equation to Java programming language and in particular, to our variables. We define the distance function:

```java
public double distance(double lat1, double lng1, double lat2, double lng2) {
    double distance = 0;
    // Converting to radians
    lat1 = Math.toRadians(lat1);
    lat2 = Math.toRadians(lat2);
    lng1 = Math.toRadians(lng1);
    lng2 = Math.toRadians(lng2);
    // Applying the formula
    double x = (lng2 - lng1) * Math.cos((lat1 + lat2) / 2);
    double y = (lat2 - lat1);
    distance = Math.sqrt(x * x + y * y) * EARTH_RADIUS;
    return distance;
}
```

Sorting  Now that we have calculated the distance separating the user’s position to the parking, we have to sort our list by distance. In order to enhance the user’s experience we are going to filter our parkings list also by availability: if the parking is free or not.

```java
// Go through each item on the list
Iterator<ParkingMarker> i = parkings.iterator();
while (i.hasNext()) {
    ParkingMarker parking = i.next();
    aLat = parking.getLat();
    aLng = parking.getLng();
    // Calculating the distance separating it from my position
    double dista = distance(aLat, aLng, mLat, mLng);
    parking.setDist(dista);
    // Filtering: if dist > filter_radius, delete
    if (dista > RADIUS_DIST && FILTERING_RADIUS) {
        i.remove();
    }
    // Defining a comparator to sort by distance
    Comparator<ParkingMarker> comparatorDistantType = new Comparator<ParkingMarker>() {
        @Override
        public int compare(ParkingMarker o1, ParkingMarker o2) {
            String p1, p2;
```
```java
int result;

// Comparing the type of parking
if (o1.getFree()) p1="a";
else p1 = "b";
if (o2.getFree()) p2="a";
else p2 = "b";
result =p1.compareTo(p2);

// If they are of the same type, compare the distance
if (result == 0) {
    result = Double.compare(o1.getDist(), o2.getDist());
}
return result;

// Sorting
Collections.sort(free_parkings, comparatorDistanceAndType);
```

**Color**  Finally, depending on the status of the parking, we are going to show one color or another. We have also added also the possibility for the user to define in the Settings menu some default colors. Therefore, the first thing is to retrieve these preferences and store them in two variables:

```java
// Retrieving Color Preferences
int color_free_int = Integer.parseInt(sh.getString("pref_markercolorFREE", getString(R.string.pref_markercolorFREEdefault)));
int color_occupied_int = Integer.parseInt(sh.getString("pref_markercolorOCCUPIED", getString(R.string.pref_markercolorOCCUPIEDdefault)));
float markerColor_free, markerColor_occupied, markerColor;
markerColor_free = retrieveColor(color_free_int);
markerColor_occupied = retrieveColor(color_occupied_int);
```

One we have our base colors, we are going to set them depending on the status:

```java
// change the color depending on the status
if (parking.getFree())
    markerColor = markerColor_free;
else{
    markerColor = markerColor_occupied;
}
```

We have now personalized our markers and list, adding distances, sorting them and changing their colors. We can see the new modifications in the Figure 9.

### 3.2.4 Listeners

We have correctly displayed all the elements displayed correctly in our application, but if we press any of them nothing will happen. This is due to the fact that we must set a few listeners in our app in order to interact with it. An event listener[^36] or simply listener, is a method or a set of actions waiting that an event takes place to be executed. This event can be a mouse click, a finger touch or a shaking of the phone for example. We have a couple of elements in our application for which we want things to happen:

- The buttons in the ActionBar (the event listeners come by default).
- When we press a marker on the map, the GPS app will launch and guide us to the destination.
- Also, when we pick an element on the list, also the GPS will launch.

We now provide the explanation of the set up of only one of the listeners. The whole code implementing the rest of them can be found in the Annex VI.

When we pick an item of the list  We are going to use the OnItemClickListener. Because I don’t want to launch the GPS app if the parking is full, the first thing to do is to retrieve the object and check this condition. To do this, first we have to call the adapter and pick the item in the position that we have pressed.

```java
// Obtaining the status
boolean status_selected = ((ParkingMarker) adapter.getAdapter().getItem(position)).getFree();
```

And if the status is free, then call an Intent to launch the GPS application

```java
if (status_selected) {
    // Launch GPS Navigation
    Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse("google.navigation:q=" + parkings.get(position).getLat() +"," + parkings.get(position).getLng()));
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    startActivity(intent);
}
```

### 3.3 Synchronization Awareness

A very important point in Android to have in mind is the synchronization and concurrency. Android usually modifies and handles events from one single thread, also called main thread, MainActivity. It also collects all events in a queue and process them with the Looper class.

If the program does not use any concurrency and all the code runs in the main thread, every operation is executed after each other. So, if a task takes a long time, until it finishes, the application will remain blocked. To provide a good user experience and prevent potential slowdowns, we have to run certain tasks asynchronously.

**Note:** Since Android 4.0, to prevent slow running operations, we cannot do network, file and database operations in the main thread. All these operations must be done asynchronously.

Following these rules, we have to adapt our application in such way as it retrieves the XML/JSON document asynchronously.

In the main thread (MainActivity)

---

38 Android Google Developers, "AdapterView.OnItemClickLis tener", Google Developers (2013)  
// We call to an auxiliary Async method to retrieve all the information following the Google criteria
RetrieveFeed task = new RetrieveFeed();
Toast.makeText(this, "Parsing the XML file", Toast.LENGTH_SHORT).show();
Log.d("PARSER", "Ejecuta el Async");

// We execute the task catching the possible exceptions
try {
  Log.d("PARSER", "TaskGet");
task.get(5, TimeUnit.SECONDS);
  Log.d("PARSER", "Fin del TaskGet");
  /**
  * Adding extra options
  */
} catch (InterruptedException e) {
e.printStackTrace();
} catch (ExecutionException e) {
e.printStackTrace();
} catch (TimeoutException e) {
e.printStackTrace();
}

In an asynchronous thread

private class RetrieveFeed extends AsyncTask<String, Integer, Boolean> {

protected Boolean doInBackground(String... params) {
  // Getting the parkings
  ParkingParser parkingParser = new ParkingParser(params[0]);
  parkings = parkingParser.parse();
  */
  /* ... sorting ... */
  return true;

protected void onPostExecute(Boolean result) {
  // Here we add the colors and create the markers and the list
  ....
}

At this point, the design of our application is finally completed. The next step is to research about possible architectures for the notifications system and implement them.
Part III

Architecture of notifications

In our application we will use two main different technologies: Wi-Fi Direct and Google Cloud Messaging. In this section we are going to thoroughly describe both of them with their benefits and their flaws. Moreover, we are also going to define which solution we are going to use and how to implement it.

1 Wi-Fi Direct

1.1 Introduction

The standard Wi-Fi Peer-to-Peer or Wi-Fi Direct is a Wi-Fi standard that allows to establish a connection between two or more devices without requiring an access point. One of the main benefits of Wi-Fi Direct is that the technology behind Wi-Fi P2P is built on the Wi-Fi strengths such as performance, security or ease of use. In devices with Wi-Fi 802.11n support, you can reach download speeds of 54 Mbps, have a range of action between 1 and 100 meters and activate WPA2 security.

1.1.1 How Wi-Fi Direct works

In order to establish a Wi-Fi Direct connection we need at the very least two or more Wi-Fi P2P Certified devices. The next step is to create a new connection or join an existing one.

- If you create a new connection, the device will turn into an Access Point and will be the new group owner. As a group owner, you can accept or reject connections from other devices, but you cannot transfer files across the network.

- If you join an existing connection, the device will try to connect to the group owner as a normal Wi-Fi connection. Once the connection is established, we can share data and messages to the group owner device (one-to-one) or to a group of devices also connected (one-to-many).

- Any peer can leave a group but only the group owner can remove the group and end all connections.

1.1.2 Why Wi-Fi Direct?

Amongst other technologies, we have studied Wi-Fi Direct because of its similarity to the possible ways of connection in a SmartCity or a VANET. We are talking about average ranges of 50 meters and high download speeds. So, as we cannot test in real conditions with real technologies (the VANET-Android connection is not implemented yet), we think that a good way to simulate it is to connect two devices by Wi-Fi Direct and then, send one file from one to the other.

1.2 Implementation

Android provides an official support for Wi-Fi Direct since Android 4.0 Ice Cream Sandwich version. The WifiP2PManager class provides us different methods to interact with the hardware (Table 1 on the following page).

---

35
Table 1: Wi-Fi P2P API Overview

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize()</td>
<td>Registers the application with the Wi-Fi framework. This</td>
</tr>
<tr>
<td></td>
<td>must be called before calling any other Wi-Fi P2P method.</td>
</tr>
<tr>
<td>connect()</td>
<td>Starts a peer-to-peer connection with a device with the</td>
</tr>
<tr>
<td></td>
<td>specified configuration.</td>
</tr>
<tr>
<td>cancelConnect()</td>
<td>Cancels any ongoing peer-to-peer group negotiation.</td>
</tr>
<tr>
<td>requestConnectInfo()</td>
<td>Requests a device’s connection information.</td>
</tr>
<tr>
<td>createGroup()</td>
<td>Creates a peer-to-peer group with the current device as the</td>
</tr>
<tr>
<td></td>
<td>group owner.</td>
</tr>
<tr>
<td>removeGroup()</td>
<td>Removes the current peer-to-peer group.</td>
</tr>
<tr>
<td>requestGroupInfo()</td>
<td>Requests peer-to-peer group information.</td>
</tr>
<tr>
<td>discoverPeers()</td>
<td>Initiates peer discovery</td>
</tr>
<tr>
<td>requestPeers()</td>
<td>Requests the current list of discovered peers.</td>
</tr>
</tbody>
</table>

1.2.1 Creating a Broadcast Receiver

The first thing that we have to implement is a broadcast receiver. It allows our application to respond to the events that we are interested in. The most important part is the onReceive() method where we specify the actions that take place depending on the received intent. For example, if we receive the WIFI_P2P_PEERS_CHANGED_ACTION intent, it means that there are new devices to connect to. So, we will retrieve a list of them using the requestPeers() method:

```java
if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {
    // Call WifiP2pManager.requestPeers() to get a list of current peers
    if (mManager != null) {
        mManager.requestPeers(mChannel, (PeerListListener)mActivity.getFragmentManager().findFragmentById(R.id.deviceList));
        return;
    }
}
```

The whole implementation of the Broadcast Receiver can be found in the Annex VI on page 65.

1.2.2 Connecting to Peers

To connect two devices, two main actions are needed:

1. Discover peers. To discover other peers in the network we can call the method discoverPeers() provided by the Android API. We have to call this function in an asynchronous way (more information in the Synchronization Awareness in section 3.3 on page 32) and collect the result with onSuccess() or onFailure(). If some peers are discovered, a WIFI_P2P_PEERS_CHANGED_ACTION is broadcasted. This intent is caught in our BroadcastReceiver defined in the previous section and a full list of the peers is obtained.

2. After we have obtained the list of peers, we can select one and call the connect() method to connect to the device.

1.2.3 Transferring data

When two or more devices are connected we can start to transfer data between them using sockets.

1. Firstly we have to create a ServerSocket. This socket is waiting a connection on a defined port. This tasks must be done in an asynchronous thread.

2. Secondly we have to create a ClientSocket. Using the IP address and port of the ServerSocket, it establishes the connection.

3. Then, when the client is connected to the server we can start to transfer data with byte streams.

4. Meanwhile, the server socket detects the connection and starts to receive the data.
With these three main actions, implementing a BroadcastReceiver, connecting to peers and transferring data, we have a basic Wi-Fi P2P android application that can be used for testing purposes.

1.3 Conclusions

We have tested our Wi-Fi Direct over two different devices and the results are not as satisfactory as we would like. Here are the main issues we encountered when dealing with a Wi-Fi Direct communication:

**Connection Establishment** Initially we thought that establishing a connection between two devices over Wi-Fi P2P would be immediate. We have, however, discovered that different actions are needed instead. To connect two devices we need to activate the Wi-Fi Direct module in both phones, search for peers, select a peer, send a connection request, accept it, and only then we have established the connection. In our application we want to receive the file or notification instantly, without having to manually establish the connection before.

**Application presence** We also discovered that in order to have a working Wi-Fi Direct connection, both devices must have the application opened. If one device closes the app, the connection stops to work or cannot be established. In our application, we want to receive a notification, whenever we have or not our application opened.

**Transfer limitations** The fact that the file transfer can only be made in one way (from user to group owner) is a very important limitation as we want in our application to have a bidirectional communication.

Taking all this points in consideration, we have decided to investigate and choose another architecture for our notification system.

The whole application uses five Android classes that we have uploaded to a GitHub repository. Everything can be consulted and used under a GPL license.

**2 Google Cloud Messaging (GCM)**

In 2011, Google implemented an evolution of an existent service, Android Cloud to Device Messaging, called Google Cloud Messaging (GCM) for Android. Since then GCM has experimented a 400% grow in every 10 months and 60% of the most popular Android applications utilize it. Let us explain what is Google Cloud Messaging and how it works.

**2.1 GCM Introduction**

The Google Cloud Messaging is a free service whose purpose is to help us send a message from our servers to all the users devices. For example, if we want to notify all of our users that there is a new update of their data (send_to_sync) or directly send that data (send_data), instead of sending multiple devices we only need to send one message to the GCM server and it takes care of the rest. The main characteristics of Google Cloud Messaging are the following:

- It allows third-party servers to send message to their Android devices
- It also allows the servers to receive upstream messages from the users using the XMPP protocol.
- An Android application doesn’t need to be running a determined application in order to receive the message. Automatically, the system will wake up when a message is received.
- GCM passes raw message data received from the application server without adding anything.
• It requires Google Play Services and an Android 2.2 firmware version installed in the device.

All of these characteristics fit perfectly for our application and so, we now proceed to its implementation.

2.1.1 Architectural Overview

To implement GCM in our application we need three things: an application server, and android application and a connection to the GCM servers. All these components interact as shown in Figure 10.

![Google Cloud Messaging Architectural Overview](image)

Figure 10: Google Cloud Messaging Architectural Overview

In a more detailed way:

1. If the devices sends a <hello> message to the Google Cloud Messaging server
2. If the application is not registered yet, the server stores it and then sends a new unique RegID.
3. The application sends to its server the RegID received. The whole setup is now completed.
4. When the application server wants to send a new message, it sends a simple message to the GCM server...
5. ... whom it broadcasts it to all of the registered devices.

2.2 Application Design

The figure on the facing page illustrates the flow of information in our application. There are four main aspects involved in the whole process:

• T1: Registration of devices in the Google Cloud Messaging network and our servers
• T2: Emergency event/user-generated content and its posting to our servers.
• T3: Launching a push notification to all of our application users
• T4: From the device, re-share the content and access to different resources

We are now going to explain them a little bit more in detail.

When a new user opens for the first time our application, an automatic (and invisible to the user) registration to the GCM servers is done. Following this process, the device tells our emrisco server its new GCM iD. Now, the device is registered in both places, the GCM servers and our main servers. We will explain further the reason of this registration in the T3 state.

Once the device is registered, it is able to generate content. This content can be of any kind:

• Alert message: the most common. It can be generated automatically once a collision or a traffic jam is detected.

• Media content: For performance reasons, it is not optimal to send a whole video or image through our notifications system. In order to do so, a first step is required. Using the Youtube API or the Imgur API, the user can easily upload any kind of media content and retrieve an URL or code that identifies it. We will send then this identifier through our notification service.

A request is made to our servers, passing the message needed to share. Our emrisco server communicates with the GCM servers, asking to generate a notification to all our registered users. And so, GCM generates a push notification that will appear automatically in all the devices with our message.

Once the device has received the message, the user can consume the content: the picture or video will automatically appear in the application. Another content example that we have developed is the ability to display in a map all the near available parking spots. The user can filter them by radius, sort them according to different parameters or launch a GPS navigator to reach them.

Finally, the user can re-share the notification using all the options that a mobile phone can offer (twitter, Facebook, Whatsapp, messaging, mail ...). This way, the message will spread easily and will reach a larger audience. An interesting option is the possibility of sending the notification to an EMRISCO AccessPoint which will be able to re-share the message using different protocols as VANET.
2.3 GCM Server

The server side of Google Cloud Messaging is divided in two elements: our third-party server EMRISCO Server and the Google-provided GCM Connection Servers. These servers take the messages sent by our EMRISCO server and forward them to our Android application.

2.3.1 Deciding the type of server

The GCM connection server (our EMRISCO server) can be implemented in two ways: HTTP and CSS (XMPP\(^{48}\)). Both ways can be used separately or in tandem, with the following differences:

- **Upstream/Downstream messages**
  - GCM HTTP: Downstream only: cloud-to-device.
  - CCS: Upstream and downstream (device-to-cloud, cloud-to-device).

- **Asynchronous messaging**
  - GCM HTTP: 3rd-party app servers send messages as HTTP POST requests and wait for a response. This mechanism is synchronous and causes the sender to block before sending another message.
  - CCS: 3rd-party app servers connect to Google infrastructure using a persistent XMPP connection and send/receive messages to/from all their devices at full line speed. CCS sends acknowledgment or failure notifications (in the form of special ACK and NACK JSON-encoded XMPP messages) asynchronously.

- **JSON**
  - GCM HTTP: JSON messages sent as HTTP POST.
  - CCS: JSON messages encapsulated in XMPP messages.

Figure 12: GCM Connection Server differences by Google Android Developers

Basically, its role is to be able to communicate with both, the GCM server and my client, forwarding correctly the messages from one to the other. In order to do that, it needs to handle requests, store API keys and registration IDs and finally, it has to be able to generate messages.

2.3.2 HTTP Connection server

Taking into account the actual uses of our application and the complexity of programming a new server, we have decided to choose an HTTP server as our EMRISCO server.

**Authorization and Sending a message** To send a message, our server has to issue a POST request to the url provided by Google\(^{49}\) for this purpose. This message is divided in 2 parts, the HTTP header and the HTTP body. The HTTP header must contain the authorization field and another field defining the type of message. The HTTP body depends on the type of message (JSON or plain text). An example of message can be:

```plaintext
1 Content-Type: application/json
2 Authorization: key=API_key
3 {
4   "ids" : ["1, 2, 5"],
5   "data" : { "TEST MESSAGE"
6   },
7   "time" : 9871986387162
8 }
```

Where the ids field would be the devices ids to which we want to send the message.

\(^{48}\)Moffitt, Jack, Professional XMPP Programming with JavaScript and jQuery (Wrox, 2010).

\(^{49}\)URL for HTTP servers' POST inquiries: http://android.googleapis.com/gcm/send
Responses When we try to send a message, multiple outcomes can take place. We have summarized them in the Table 2.

<table>
<thead>
<tr>
<th>Response</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 (Success)</td>
<td>The response body contains a JSON object with different fields: the ID of the multicast message, the number of messages processed without an error, the number of messages that could not be processed, the number of results that contain a canonical registration ID and an array of objects representing the status of the message processed.</td>
</tr>
<tr>
<td>400</td>
<td>Only applies for JSON requests and indicates that the request could not be parsed as JSON or it contained invalid fields.</td>
</tr>
<tr>
<td>401</td>
<td>There was an error authenticating the sender account.</td>
</tr>
<tr>
<td>5XX</td>
<td>Internal error in the GCM server while processing the request. Also, the server is temporarily unavailable.</td>
</tr>
</tbody>
</table>

Table 2: Google Cloud Messaging HTTP POST responses

2.3.3 Implementation of the GCM Server

The server is set up using a standard servlet web server. All the java classes are uploaded to a Github repository and can be consulted anytime, but we are going to describe a few aspects of some of them.

- ApiKeyInitializer.java: In this class, we tell the server that the API key needed to communicate with the GCM server is located inside the api.key file in the root directory of the server. If the file cannot be accessed an exception is thrown.
- BaseServlet.java: This is an standard class defining the main methods of a servlet web server (doGet, getParameter, setSuccess,...).

Figure 13: HTTP GCM Server

50 Hunter, Jason and Crawford, William, Java Servlet Programming (O’Reilly, 2001).
• Datastore.java: Where all the methods to register and unregister users are stored.

• HomeServlet.java: The HelloWorld page where the fields of the message are settled and then passed to SendAllMessagesServlet.java. We can see a screenshot of the interface in Figure 13 on the preceding page.

• RegisterServlet/UnregisterServlet.java: Called when a new user is registered. Passes the new ID to Datastore.java. Opposite effect using the second version.

• SendAllMessagesServlet.java: The main class. In the beginning we retrieve the API key from ApiKeyInitializer.java and the parameters from HomeServlet.java adding some logic.

```java
// Retrieving data
String eventType = req.getParameter("eventType");
String eventLocation = req.getParameter("eventLocation");
if (eventLocation.equals("Diagonal")) {
    eventLat = "41.3856347";
    eventLng = "2.116170799999635";
} else if (eventLocation.equals("Campus Nord")) {
    eventLat = "41.38902840000001";
    eventLng = "2.112550699999929";
} else if (eventLocation.equals("Paris")) {
    eventLat = "48.848062999999999";
    eventLng = "2.3473043999999845";
}
```

Then it checks if we have devices subscribed (IDs on the database). If we have only one device, it directly sends the message. But if we have multiple devices, we send a multicast message using JSON splitting in chunks of 1000 devices, the GCM limit.

```java
// send a multicast message using JSON
// must split in chunks of 1000 devices (GCM limit)
List<String> partialDevices = new ArrayList<String>(total);
int counter = 0;
for (String device : devices) {
    counter++;
    partialDevices.add(device);
    int partialSize = partialDevices.size();
    if (partialSize == MULTICAST_SIZE || counter == total) {
        // function to build the message and send it
        asyncSend(partialDevices, eventType, eventLat, eventLng, extras_content, extras);
        partialDevices.clear();
        tasks++;
    }
}
```

We build the message using the parameters retrieved from the HomeServlet and then we send it to the GCM servers.

```java
String registrationId = devices.get(0);
Builder builder = new Message.Builder();
    builder.addData("eventType", eventType);
    builder.addData("eventLat", eventLat);
    builder.addData("eventLng", eventLng);
    builder.addData("eventExtras", extras_content);
    builder.addData("eventExtras_type", extras);
    Message message = builder.build();
    Result result = sender.send(message, registrationId, 5);
```

The full class is in Annex VI on page 67.

The whole server application uses seven Java classes that we have uploaded to a GitHub repository. Everything can be consulted and used under a GPL license.

[^EMRISCO HTTP GCM Servlet Server]: Adrian Latorre
http://Github.com/willylatorre/GCM-servlet-server-CSS.
2.4 GCM Client

The GCM Client is the Android side of Google Cloud Messaging. It has to be included in our app and its function is to register the device for GCM, send and also receive messages. Of course, the client can be a lot more complex, but these are the minimum functions to be able to function properly. We are going to describe how to implement the different parts of code in our application and what are their functions.

2.4.1 Permissions

In order to have all the parts of the GCM client functioning, we need to set some permissions on the Android Application’s Manifest. This manifests presents essential information about the app to the Android system needed to run the code. There is a lot more information about the App Manifest in the Google Developer’s manual.

We need to add the following permissions to our manifest:

- **com.google.android.c2dm.permission.RECEIVE**: to register and receive messages
- **android.permission.INTERNET**: to send the registration ID to the 3rd party server.
- **android.permission.GET_ACCOUNTS**: as GCM requires a Google account.
- **android.permission.WAKE_LOCK**: to keep the processor from sleeping when a message is received
- **XXXXX.permission.C2D_MESSAGE**: to prevent other Android applications from registering and receiving the Android application’s messages.
- **com.google.android.c2dm.SEND**: so that only the receiver can get a message from the GCM Framework.
- **Service (IntentService)**: where the WakefulBroadCastReceiver relies on handling the GCM message.

2.4.2 Implementation of the GCM Client

There are three main sections in the GCM Client application: registering, sending (only in XMPP) and receiving messages. Let us give a quick look to each one of them and understand how have they been implemented.

**Registering a device**  As defined in the introduction section, the first thing the Android app needs to do is register itself with the GCM servers. When it successfully registers, it stores the value for future use. So, in our onCreate() method we will call the getRegistrationId to check if there is an existing registrationID stored.

```java
private String getRegistrationId(Context context) {
    final SharedPreferences prefs = getGCMPreferences(context);
    String registrationId = prefs.getString(PROPERTY_REG_ID, "")
    if (registrationId.isEmpty()) {
        Log.i(TAG, "Registration not found.");
        return "";
    } // Check if app was updated; if so, it must clear the registration ID
    // since the existing regID is not guaranteed to work with the new
    // app version.
    int registeredVersion = prefs.getInt(PROPERTY_APP_VERSION, Integer.MIN_VALUE);
    int currentVersion = getAppVersion(context);
    if (registeredVersion != currentVersion) {
        Log.i(TAG, "App version changed.");
        return "";
    } // Check if app was updated; if so, it must clear the registration ID
    return registrationId;
}
```

If the registrationID doesn’t exist, the returned string is empty and the application will call the registerInBackground() method. The registration is done in an asynchronous task for the same reasons defined in section 6.3 Synchronization Awareness.

```java
private void registerInBackground() {
    new AsyncTask() {
        @Override
        protected String doInBackground(Void... params) {
            String msg = "";
            try {
                if (gcm == null) {
                    gcm = GoogleCloudMessaging.getInstance(context);
                }
                regid = gcm.register(SENDER_ID);
                msg = "Device registered, registration ID=" + regid;
                //sending the registrationID to our EMRISCO server
                sendRegistrationIdToBackend();
                //Save the registrationID for the next time
                storeRegistrationId(context, regid);
            } catch (IOException ex) {
                msg = "Error:" + ex.getMessage();
                // If there is an error, don’t just keep trying to register.
            }
            return msg;
        }
    }.execute(null, null, null);
}
```

Now, we have registered our device. As we don’t send messages with this device, the next step is to implement the message reception.

**Receiving a message** The application includes a WakefulBroadcastReceiver, a mechanism used by GCM to deliver the messages. This Receiver is a special type of broadcast receiver whose function is to take care of creating and managing a partial wake lock for our application. This way, even if the device is locked, it can still receive messages.

The BroadcastReceiver passes the task of processing the message to the GCMIntentService.java and returns to its main function described before.

```java
public void onReceive(Context context, Intent intent) {
    // Explicitly specify that GcmIntentService will handle the intent.
    ComponentName comp = new ComponentName(context.getPackageName(),
        GcmIntentService.class.getName());
    // Start the service, keeping the device awake while it is launching.
    startWakefulService(context, (intent.setComponent(comp)));
    setResultCode(Activity.RESULT_OK);
}
```

However, we have to be careful as we might receive another message while passing the first one to the intent service. A lock to keep the device awake while it is launching is therefore needed to prevent these situations.

**Processing the message** This snippet of code shows us how to handle the message and retrieve the content. Finally, we will tell the device to make the notification icon appear:

```java
protected void onMessage(Context context, Intent intent) {
    Log.i(TAG, "Received message");
    //Retrieving Notification Preferences
    SharedPreferences sh = PreferenceManager.getDefaultSharedPreferences(this);
    int distance_max = Integer.parseInt(sh.getString("pref_notifradius", getString(R.
        string.pref_notifradiusDefault)));
    //Retrieving variables
    gcm_type=intent.getStringExtra("eventType");
    //HARDCODE Timestamp
```
The most interesting and challenging part is at the end, when we have filtered the messages so the users won’t be bothered by thousands of messages. Also, at this point we can filter the messages by any attributes or characteristics of our choice: type of vehicle, position, type of content, etc...

Generating a notification  Finally, we need to generate the notification that will appear on the status bar. This is done using the notification Manager.

```java
private static void generateNotification(Context context, String message) {
    int icon = R.drawable.ic_stat_gcm;
    long when = System.currentTimeMillis();
    NotificationManager notificationManager = (NotificationManager) context.getSystemService(Context.NOTIFICATION_SERVICE);
    Intent notificationIntent = new Intent(context, ShareNotification.class);
    //adding parameters
    notificationIntent.putExtra("type", gcm_type);
    notificationIntent.putExtra("time", gcm_time);
    notificationIntent.putExtra("lat", gcm_lat);
    notificationIntent.putExtra("lng", gcm_lng);
    notificationIntent.putExtra("extras", gcm_extras);
    notificationIntent.putExtra("extras_type", gcm_extras_type);
    // set intent so it does not start a new activity
    notificationIntent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_SINGLE_TOP);
    PendingIntent pendingNotificationIntent = PendingIntent.getActivity(context, 0, notificationIntent, PendingIntent.FLAG_UPDATE_CURRENT);
    Notification notification = new Notification.Builder(context)
            .setContentIntent(pendingNotificationIntent)
            .setContentTitle("OJO!! Ha pasado algo")
            .setContentText(message)
            .setSmallIcon(R.drawable.ic_gcm_noti)
            .setLargeIcon(BitmapFactory.decodeResource(res, R.drawable.ic_gcm_noti))
            .setAutoCancel(true)
            .build();
    notificationManager.notify(0, notification);
}
```

Basically, this method puts in the notificationIntent all the extra content received in the message, puts some flags to prevent unwanted actions and finally builds the notification with a personalized message and icon. We can see the notification example in Figure 14 on the following page.
3 Sharing Notifications

We have now a notification icon appearing in our status bar. But what do we do, now? We need to develop a class that will be called when the notification is selected in order to start some action. ShareNotifications.java will read the message received and the extra content (if there is any), display it on screen and then offer to the user the possibility to call or share it with other people.

3.1 Retrieving the message

Firstly we have to retrieve the message embedded in the notification. This message has been passed in the ‘extras’ field of an Intent, and we are going to read its content using:

```java
gcm_type = extras.getString("type");
```

**Date** A more complex field to retrieve is the date. We have passed the Timestamp of the time of the event. The Unix Timestamp is a way to track the time as the total number of seconds that have passed since January 1st, 1970. It is used usually to track times and positions because it doesn’t depend of our position on the globe. To convert a timestamp to a date format in order to place it in the message, we have to do a simple conversion:

```java
//Retrieving the timestamp
Timestamp tstamp = Timestamp.valueOf(gcm_time);
Date d = new Date();

//Check if we have received correctly the timestamp to prevent parsing errors
if(tstamp != null){
    d.setTime(tstamp.getTime());
    //Pass the date to a string
    gcm_time_parsed = d.toString();
} else {
```

52Unix Timestamp [http://www.unixtimestamp.com](http://www.unixtimestamp.com)
3.2 Reverse Geocatching

From the message we have a latitude and a longitude, but instead of showing to the user its values, we are going to perform a reverse geocaching to obtain the address associated to them. We are going to use the Geocoder class from Google and manipulate in an efficient way the results.

```java
Geocoder geocoder;
List<Address> addresses = new ArrayList<Address>();//
try {
geocoder = new Geocoder(this, Locale.getDefault());
addresses = geocoder.getFromLocation(lat, lng, 1);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
String address = addresses.get(0).getAddressLine(0);
String city = addresses.get(0).getAddressLine(1);
String country = addresses.get(0).getAddressLine(2);
String place = address + "," + city;
```

3.3 Extra Content

We have the option to add as extra content in our message a Youtube video or an image, so we need to implement a way to show this media. The first thing to do is to detect if there is extra content or there isn’t. If we don’t have any, we will make the button disappear:

```java
if (gcm_extras_type.equals("none")) {
extrasButton.setVisibility(View.GONE);
}
```

If there is some extra content, we will launch an Intent with the media url.

```java
extrasButton.setVisibility(View.VISIBLE);
extrasButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
        Intent i = new Intent(Intent.ACTION_VIEW);
i.setData(Uri.parse(gcm_extras));
startActivity(i);
    }
});
```

3.4 Sharing the message

Finally, we need to compose the final message and then launch an ACTION_SEND Intent. This way, any application that we have installed in our device and that is susceptible of sending a message will offer itself to realize the action.

```java
// Message sharing
String message_build = "";
if (!gcm_extras.isEmpty()) {
    message_build = "Atención: " + gcm_type + " en " + place + ".
Contenido extra: " + gcm_extras;
} else {
    message_build = "Atención: " + gcm_type + " en " + place;
}
```
final String message = message_build;
final EditText message_text = (EditText) findViewById(R.id.shareMessage);
message_text.setText(message);

// Share button
ImageButton shareButton = (ImageButton) findViewById(R.id.share);
shareButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
        String final_message = message_text.getText().toString();
        Intent sendIntent = new Intent();
        sendIntent.setAction(Intent.ACTION_SEND);
        sendIntent.putExtra(Intent.EXTRA_TEXT, final_message);
        sendIntent.setType("text/plain");
        startActivity(Intent.createChooser(sendIntent,
            getResources().getText(R.string.send_to)));
    }
});

We have finished our sharing notifications class and therefore, our application.

![Sharing Notifications](image)

Figure 15: Sharing Notifications

4 Flow summary

At this point, our application is finally completed. As a reminder, we are going to review the whole application flow, from the start of the service until the end, the notifications system.

1. In the beginning, when the user starts the application a Map and a List containers are created. These will be the places where we will store all the information. Also, the Google Play services are initiated and the user’s location is gathered.

2. The next step is to activate the GoogleCloudMessaging notifications. This is done using the method register() to register the device on the GCM servers and obtain a GCM_id.
3. Then, in an asynchronous thread, we launch the background tasks: parsing the parking document, sorting, filtering and personalizing. The markers are added to the Map container, the List is filled with items and the listeners are placed on those elements.

4. In the meanwhile, a user has generated content and has sent a notification to the GCM server which sends us a message indicating that a new content is available.

5. After reading this message with the method onMessage(), the application will parse the data passed within the message and will display a notification in the taskbar with all the relevant information.

6. Finally, when the user clicks on the notification, the SharingActivity will be displayed offering the user the possibility to share, in different ways, the message received.

We can see a visual representation of this flow in the Figure 16 on the next page.
Figure 16: A flow summary
Part IV
Conclusions

1 Results interpretation

In the process of creating this application we have obtained a lot of insight and encountered many problems that, if we were to redo the app from scratch, we would choose differently:

- **XML vs JSON**: On a first instance, we thought that parsing an XML would be the most logical and easier solution to retrieve data. After doing some research, we have learned that the most optimal way to retrieve information is to use JSON documents. JSON files are much lighter, and thus consumes less data, and are more versatile. Also, parsing them is much easier. It is, however, alright to do it both ways as most of the open data information provided by cities is still in XML format.

- **Synchronization awareness**: Until we completely understood the asynchronous aspects of an Android application, we couldn’t make it work. Before starting to develop an application, it is a good practice to plan carefully the whole flow of it to take into consideration all these points.

- **Wi-Fi Direct**: In a first appreciation, we thought that a Wi-Fi Direct communication was the best way to simulate a VANET exchange between two nodes. It has proven that nowadays, Wi-Fi Direct is still not ready for complex applications and its use is limited to certain cases. Also, having to pre-establish a connection and to have always the application open in order to work, was not the scenario that we were looking for. It has been a good decision to stop further development on this path and switch to a completely different technology.

Nevertheless, we are pleased with the results obtained with our application.

2 Future implementations and applications

If we had been granted a bit more time, there is a list of upgrades that we think would be great for our application and would add interesting functionalities:

- **Local Database**: Instead of downloading each time the parking spots file and parsing it, it would be more optimal to save all the data in a local database. When the user refreshes the application, a query is launched asking to the server if the data has been modified since the last check. If not, our device will not download anything. If, however, there is a variation (a parking has changed its status), we will only download the information about this concrete element. This change will save a lot of data traffic for the application.

- **Add properties to the vehicles**: We can add different properties such as size of the vehicle or paying vehicles in order to refine our filtering criteria.

- **Web version**: It would be interesting to develop a web version to check from a computer all the data.

- **Tablet layout**: An interesting change is to add a tablet layout so in a horizontal stand, the parking list will appear as a side bar and the map will have a bigger space dedicated.

- **Multiple notifications**: We need to implement a solution for the scenario where multiple notifications of the same kind are received in the same area (multiple citizens informing about an accident for example).

Also, our application is completely adaptable to different kind of services. If we want to show bus stops instead of parking spots, with the right data organization, the application will automatically work without further changes.
Part V

References

References


Part VI
Annexes

Annex 1 Simple application with a map centered on the user’s location

```java
public class MainActivity extends FragmentActivity implements
GooglePlayServicesClient.ConnectionCallbacks,
GooglePlayServicesClient.OnConnectionFailedListener {

private SupportMapFragment mapFragment;
private GoogleMap map;
private LocationClient mLocationClient;

/*
* Define a request code to send to Google Play services
* This code is returned in Activity.onActivityResult
*/
private final static int CONNECTION_FAILURE_RESOLUTION_REQUEST = 9000;

// Define a DialogFragment that displays the error dialog
public static class ErrorDialogFragment extends DialogFragment {

    // Global field to contain the error dialog
    private Dialog mDialog;

    // Default constructor. Sets the dialog field to null
    public ErrorDialogFragment() {
        super();
        mDialog = null;
    }

    // Set the dialog to display
    public void setDialog(Dialog dialog) {
        mDialog = dialog;
    }

    // Return a Dialog to the DialogFragment.
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        return mDialog;
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_activity);
        mLocationClient = new LocationClient(this, this, this);
        mapFragment = ((SupportMapFragment) getSupportFragmentManager().findFragmentById(R.id.map));
        map = mapFragment.getMap();
        map.setMyLocationEnabled(true);
    }

    /*
    * Called when the Activity becomes visible.
    */
    @Override
    protected void onStart() {
        super.onStart();
        // Connect the client.
        if(isGooglePlayServicesAvailable()){
            mLocationClient.connect();
        }
    }

    /*
    * Called when the Activity is no longer visible.
    */
    @Override
    protected void onStop() {
        // Disconnecting the client invalidates it.
        mLocationClient.disconnect();
        super.onStop();
    }
```

57
Override protected void onActivityResult(
    int requestCode, int resultCode, Intent data) {

    // Decide what to do based on the original request code
    switch (requestCode) {
        case CONNECTION_FAILURE_RESOLUTION_REQUEST:
            /*
             * If the result code is Activity.RESULT_OK, try
             * to connect again
             */
            switch (resultCode) {
                case Activity.RESULT_OK:
                    mLocationClient.connect();
                    break;
            }
            break;
    }
}

private boolean isGooglePlayServicesAvailable() {

    int resultCode = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);

    // If Google Play services is available
    if (ConnectionResult.SUCCESS == resultCode) {
        // In debug mode, log the status
        Log.d("Location Updates", "Google Play services is available.");
        return true;
    } else {
        // Get the error dialog from Google Play services
        Dialog errorDialog = GooglePlayServicesUtil.getErrorDialog(
            resultCode, this, CONNECTION_FAILURE_RESOLUTION_REQUEST);

        // If Google Play services can provide an error dialog
        if (errorDialog != null) {
            // Create a new Fragment for the error dialog
            ErrorDialogFragment errorFragment = new ErrorDialogFragment();
            errorFragment.setDialog(errorDialog);
            errorFragment.show(getSupportFragmentManager(), "Location Updates");

            return false;
        }
    }

    /*
     * Called by Location Services when the request to connect the
     * client finishes successfully. At this point, you can
     * request the current location or start periodic updates
     */
}

@Override public void onConnected(Bundle dataBundle) {

    // Display the connection status
    Toast.makeText(this, "Connected", Toast.LENGTH_SHORT).show();
    Location location = mLocationClient.getLastLocation();
    LatLng latLng = new LatLng(location.getLatitude(), location.getLongitude());
    CameraUpdate cameraUpdate = CameraUpdateFactory.newLatLngZoom(latLng, 17);
    map.animateCamera(cameraUpdate);
}

    /*
     * Called by Location Services if the connection to the
     * location client drops because of an error.
     */
    @Override public void onDisconnected() {

        // Display the connection status
        Toast.makeText(this, "Disconnected. Please re-connect.",
            Toast.LENGTH_SHORT).show();
    }

    /*
     * Called by Location Services if the attempt to
     * Location Services fails.
     */
}

@Override public void onConnectionFailed(ConnectionResult connectionResult) {

    /*
     * Handle results returned to the FragmentActivity
     * by Google Play services
     */
}
Google Play services can resolve some errors it detects.
* If the error has a resolution, try sending an Intent to
  start a Google Play services activity that can resolve
  the error.
*/
if (connectionResult.hasResolution()) {
  try {
    // Start an Activity that tries to resolve the error
    connectionResult.startResolutionForResult(
        this,
        CONNECTION_FAILURE_RESOLUTION_REQUEST);
    /*
     * Thrown if Google Play services canceled the original
     * PendingIntent
     */
    } catch (IntentSender.SendIntentException e) {
      // Log the error
      e.printStackTrace();
    }
  }
else {
  Toast.makeText(getApplicationContext(), "Sorry. Location services
  not available to you", Toast.LENGTH_LONG).show();
}
Annex 2: Full parking list (XML)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ParkingPositions>
  <parking>
    <name>Parking UPC</name>
    <lat>41.387179</lat>
    <lng>2.1125849</lng>
    <free>0</free>
  </parking>
  <parking>
    <name>Parking Casa Adri</name>
    <lat>41.3845864</lat>
    <lng>2.1392143</lng>
    <free>1</free>
  </parking>
  <parking>
    <name>Parking Palau Reial</name>
    <lat>41.38575629197152</lat>
    <lng>2.1175289154052734</lng>
    <free>0</free>
  </parking>
  <parking>
    <name>Parking Liceo Frances</name>
    <lat>41.392059</lat>
    <lng>2.117139</lng>
    <free>0</free>
  </parking>
  <parking>
    <name>Parking Sant Estació</name>
    <lat>41.3796349</lat>
    <lng>2.139746000000059</lng>
    <free>1</free>
  </parking>
  <parking>
    <name>Parking Camp Nou</name>
    <lat>41.3808989</lat>
    <lng>2.122818599999996</lng>
    <free>0</free>
  </parking>
  <parking>
    <name>Parking Barcelona Activa</name>
    <lat>41.406358999999999</lat>
    <lng>2.1922075999999606</lng>
    <free>1</free>
  </parking>
  <parking>
    <name>Parking Polytechnique Paris</name>
    <lat>48.848006299999999</lat>
    <lng>2.3473043999999845</lng>
    <free>1</free>
  </parking>
  <parking>
    <name>Parking Francesc Macià</name>
    <lat>41.392206</lat>
    <lng>2.143175000000042</lng>
    <free>1</free>
  </parking>
</ParkingPositions>
```
package com.example.pfc_alpha;

public class ParkingMarker {
    private String name;
    private double lat;
    private double lng;
    private boolean free;
    public ParkingMarker(String name, double lat, double lng, boolean free) {
        super();
        this.name = name;
        this.lat = lat;
        this.lng = lng;
        this.free = free;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getLat() {
        return lat;
    }
    public void setLat(double lat) {
        this.lat = lat;
    }
    public double getLng() {
        return lng;
    }
    public void setLng(double lng) {
        this.lng = lng;
    }
    public boolean getFree() {
        return free;
    }
    public void setFree(boolean free) {
        this.free = free;
    }
}
Annex 4: Parser Handler

```java
package com.example.pfc_alpha1;
import java.util.ArrayList;
import java.util.List;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
public class ParkingHandler extends DefaultHandler {
    private List<ParkingMarker> parkings;
    private ParkingMarker parkingActual;
    private boolean parsingError = false;
    public List<ParkingMarker> getParkings() {
        return parkings;
    }

    @Override
    public void startDocument() throws SAXException {
        super.startDocument();
        parkings = new ArrayList<ParkingMarker>();
        sbText = new StringBuilder();
    }

    @Override
    public void startElement(String uri, String localName, String name, Attributes attributes) throws SAXException {
        super.startElement(uri, localName, name, attributes);
        if (localName.equals("parking")) {
            parkingActual = new ParkingMarker(name, 0, 0, true);
        }
    }

    @Override
    public void characters(char[] ch, int start, int length) throws SAXException {
        if (this.parkingActual != null) {
            sbText.append(ch, start, length);
        }
    }

    @Override
    public void endElement(String uri, String localName, String name) throws SAXException {
        super.endElement(uri, localName, name);
        double value;
        if (this.parkingActual != null) {
            if (localName.equals("name")) {
                parkingActual.setName(sbText.toString().trim());
            } else if (localName.equals("lat")) {
                value = Double.parseDouble(sbText.toString().trim());
                parkingActual.setLat(value);
            } else if (localName.equals("lng")) {
                value = Double.parseDouble(sbText.toString().trim());
                parkingActual.setLng(value);
            } else if (localName.equals("free")) {
                value = Double.parseDouble(sbText.toString().trim());
                if (value==1)parkingActual.setFree(Boolean.TRUE);
                else if (value==0)parkingActual.setFree(Boolean.FALSE);
                else {
                    // NOT CORRECT STRUCTURE
                    parsingError = true;
                }
            }
            parkingActual.setName(sbText.toString().trim());
            parkings.add(parkingActual);
        }
        sbText.setLength(0);
    }
}
```
Annex 5 ArrayAdapter<ParkingMarker> to display a ListView

class ParkingAdapter extends ArrayAdapter<ParkingMarker> {
    Activity context;
    List<ParkingMarker> parkings_data;
    public ParkingAdapter(Activity context, List<ParkingMarker> parkings_data) {
        super(context, R.layout.list_details, parkings_data);
        this.context = context;
        this.parkings_data = parkings_data;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = context.getLayoutInflater();
        View item = inflater.inflate(R.layout.list_details, null);

        // We retrieve the name of the parking
        TextView lblTit = (TextView)item.findViewById(R.id.LblTitle);
        lblTit.setText(parkings_data.get(position).getName());

        // We retrieve the status of the parking
        TextView lblStat = (TextView)item.findViewById(R.id.LblStatus);
        boolean free;
        free = parkings_data.get(position).getFree();
        // Depending on the status, we change the color
        if (free) {
            lblStat.setText("Free");
            lblStat.setTextColor(Color.GREEN);
        } else {
            lblStat.setText("Occupied");
            lblStat.setTextColor(Color.RED);
        }
        return(item);
    }
}

private void addListeners() {
    // Preparing error message
    Context context = getApplicationContext();
    CharSequence text = "This parking is full!";
    int duration = Toast.LENGTH_SHORT;

    final Toast toast = Toast.makeText(context, text, duration);

    // On click Listeners for the List
    lstOptions.setOnItemClickListener(new OnItemClickListener {
        @Override
        public void onItemClick(AdapterView<?> adapter, View v, int position, long id) {
            // Obtaining the status
            boolean status_selected = ((ParkingMarker) adapter.getAdapter().getItem(position)).getFree();
            if (status_selected) {
                // Launch Navigation
                Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse("google.navigation:q=+parkings.get(position).getLat()+","+parkings.get(position).getLng()"));
                intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                startActivity(intent);
            } else {
                toast.show();
            }
        }
    });

    // On click Listeners for the markers
    mMap.setOnInfoWindowClickListener(new OnInfoWindowClickListener {
        @Override
        public void onInfoWindowClick(Marker marker) {
            // Retrieve the parkingMarker object associated
            ParkingMarker parking_info = eventMarkerMap.get(marker);
            if (parking_info.getFree()) {
                // Launch Navigation
                Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse("google.navigation:q=+parking_info.getLat()+","+parking_info.getLng()"));
                intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                startActivity(intent);
            } else {
                toast.show();
            }
        }
    });
}
Annex 7 Broadband receiver for P2P Intents

```java
package com.example.pfc_wifi_p2p;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.net.NetworkInfo;
import android.net.wifi.p2p.WifiP2pDevice;
import android.net.wifi.p2p.WifiP2pManager;
import android.net.wifi.p2p.WifiP2pManager.Channel;
import android.net.wifi.p2p.WifiP2pManager.PeerListListener;
import android.view.View;

public class WifiDirectBroadcastReceiver extends BroadcastReceiver {
    private WifiP2pManager mManager;
    private Channel mChannel;
    private MainActivity mActivity;

    public WifiDirectBroadcastReceiver(WifiP2pManager manager, Channel channel,
            MainActivity activity) {
        super();
        this.mManager = manager;
        this.mChannel = channel;
        this.mActivity = activity;
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals(action)) {
            // Check to see if Wi-Fi is enabled and notify appropriate activity
            int state = intent.getIntExtra(WifiP2pManager.EXTRA_WIFI_STATE, -1);
            if (state == WifiP2pManager.WIFI_P2P_STATE_ENABLED) {
                // Wifi P2P is enabled
                mActivity.setIsWifiP2pEnabled(true);
            } else {
                // Wi-Fi P2P is not enabled
                mActivity.setIsWifiP2pEnabled(false);
                mActivity.resetData();
            }
        } else if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {
            if (mManager == null) {
                mManager.requestPeers(mChannel, (PeerListListener)mActivity.
                        getFragmentManager().findFragmentById(R.id.devicelist));
                return;
            } else if (WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action)) {
                // Call WifiP2pManager.requestConnectionInfo() to get a list of current peers
                if (mManager != null) {
                    mManager.requestConnectionInfo(mChannel, (PeerListListener)mActivity.
                            getFragmentManager().findFragmentById(R.id.devicelist));
                    return;
                } else if (WifiP2pManager.WIFI_P2P_DISCONNECTED_ACTION.equals(action)) {
                    // Respond to new connection or disconnections
                    if (mManager == null) {
                        return;
                    }
                    NetworkInfo networkInfo = (NetworkInfo) intent
                            .getParcelableExtra(WifiP2pManager.EXTRA_NETWORK_INFO);
                    if (networkInfo.isConnected()) {
                        // we are connected with the other device, request connection info to find group owner IP
                        final DeviceListFragment listFragment = (DeviceListFragment) mActivity.
                                getFragmentManager().findFragmentById(R.id.devicelist);
                        listFragment.getView().setVisibility(View.GONE);
                        DeviceDetailFragment detailFragment = (DeviceDetailFragment) mActivity.
                                getFragmentManager().findFragmentById(R.id.devicedetail);
                        //fragment.showDetails(device);
                        mManager.requestConnectionInfo(mChannel, fragment);
                    } else {
                        // It’s a disconnect, back to main menu
                        final DeviceListFragment listFragment = (DeviceListFragment) mActivity.
                                getFragmentManager().findFragmentById(R.id.devicelist);
                    }
        }
    }
}
```
listFragment.getView().setVisibility(View.VISIBLE);

final DeviceDetailFragment fragment = (DeviceDetailFragment)
mActivity.getActivityFragmentManager()
    .findFragmentById(R.id.device_detail);

fragment.blockDetail();

return;

} else if (WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION.equals(action)) {
    // Respond to this device's wifi state changing
    mActivity.updateThisDevice((WifiP2pDevice)intent.getParcelableExtra(
        WifiP2pManager.EXTRA_WIFI_P2P_DEVICE));

    return;

}
Annex 8 GCM Send messages servlet

```java
/**
 * Servlet that adds a new message to all registered devices.
 */
@SuppressWarnings("serial")
public class SendAllMessagesServlet extends BaseServlet {

private static final int MULTICAST_SIZE = 1000;
private Sender sender;
private static final Executor threadPool = Executors.newFixedThreadPool(5);

@Override
public void init(ServletConfig config) throws ServletException {
    super.init(config);
sender = newSender(config);
}

protected Sender newSender(ServletConfig config) {
    String key = (String) config.getServletContext().getAttribute(ApiKeyInitializer.ATTRIBUTE_ACCESS_KEY);
    return new Sender(key);
}

/**
 * Processes the request to add a new message.
 */
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws IOException, ServletException {
List<String> devices = Datastore.getDevices();
String status;

// Retrieving data
String eventType = req.getParameter("eventType".);
String eventLat=null, eventLng=null;
String eventLocation = req.getParameter("eventLocation*");
if(eventLocation.equals("Diagonal")){
    eventLat = "41.3856347";
    eventLng = "2.116170799999635";
}
else if(eventLocation.equals("Campus Nord")){
    eventLat = "41.3890284000001";
    eventLng = "2.112550699999929";
}
else if(eventLocation.equals("Paris")){
    eventLat = "48.84800629999999";
    eventLng = "2.347304399999845";
}
String extras = req.getParameter("eventExtras");
String extras_content="";
if (!extras.equals("None")){
    if (extras.equals("youtube") ) extras_content = req.getParameter("youtube link");
    else if(extras.equals("img") ) extras_content = req.getParameter("img_link");
    else extras_content = "";
}
if ( devices.isEmpty()) {
    status = "Message ignored as there is no device registered!";
} else {
    // NOTE: check below is for demonstration purposes; a real application
    // could always send a multicast, even for just one recipient
    if ( devices.size() == 1 ) {
        // send a single message using plain post
        String registrationId = devices.get(0);
        Builder builder = new Message.Builder();
        builder.addData("eventType", eventType );
        builder.addData("eventLat", eventLat);
        builder.addData("eventLng", eventLng);
        builder.addData("eventExtras", extras_content);
        builder.addData("eventExtras_type", extras);
        Message message = builder.build();
        Result result = sender.send(message, registrationId, 5);
        status = "Sent message to one device: 0 + result;"
    } else {
        // send a multicast message using JSON
    }
}
```

// must split in chunks of 1000 devices (GCM limit)
int total = devices.size();
List<String> partialDevices = new ArrayList<String>(total);
int tasks = 0;
for (String device : devices) {
    counter ++;
    partialDevices.add(device);
    int partialSize = partialDevices.size();
    if (partialSize == MULTICAST_SIZE || counter == total) {
        asyncSend(partialDevices, eventType, eventLat, eventLng, extras_content, extras);
        partialDevices.clear();
        tasks ++;
    }
}
status = "Asynchronously sending " + tasks + " multicast messages to " + total + " devices";
req.setAttribute(HomeServlet.ATTRIBUTE_STATUS, status.toString());
getServletContext().getRequestDispatcher("/home").forward(req, resp);

private void asyncSend(List<String> partialDevices, final String eventType, final String eventLat, final String eventLng, final String extras_content, final String extras) {
    // make a copy
    final List<String> devices = new ArrayList<String>(partialDevices);
    threadPool.execute(new Runnable() {
        public void run() {
            try {
                multicastResult = sender.send(message, devices, 5);
            } catch (IOException e) {
                logger.log(Level.SEVERE, "Error posting messages", e);
                return;
            }
            List<Result> results = multicastResult.getResults();
            // analyze the results
            for (int i = 0; i < devices.size(); i++) {
                String regId = devices.get(i);
                Result result = results.get(i);
                String messageId = result.getMessageId();
                if (messageId != null) {
                    logger.info("Successfully sent message to device: " + regId + "; messageId = " + messageId);
                    String canonicalRegId = result.getCanonicalRegistrationId();
                    if (canonicalRegId != null) {
                        // same device has more than one registration id; update it
                        logger.info("canonicalRegId " + canonicalRegId);
                        Datastore.updateRegistration(regId, canonicalRegId);
                    } else {
                        String error = result.getErrorCodeName();
                        if (error.equals(Errors.ERROR_NOT_REGISTERED)) {
                            // application has been removed from device - unregister it
                            logger.info("Unregistered device: " + regId);
                            Datastore.unregister(regId);
                        } else {
                            logger.severe("Error sending message to " + regId + "; " + error);
                        }
                    }
                }
            }
        }
    });
}