An Android application for crowdsourcing 3G user experience

Ingeniería en Informática
Universitat Politècnica de València

Author:
Miquel Martínez Raga

Supervisor:
Dr. Juan Carlos Ruiz García

September, 2011

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Abstract

This report is composed by a project splited in two parts, a practical part and a research one. The first part of the project was done in Valencia (Spain) under the supervision of the company NUBESIS. We developed an Android application that is the mobile application for a website developed by the same company.

We describe the problems we had while developing the application and the way we solved them. We will explain the different processes that take place in the application and how these processes are integrated in the application's functionality, we also talk about the user's interaction with the different screens and their behavior.

The second part of the project is planned as a research project to improve the connectivity problem that can appear in the first application. This part was done in Sydney (Australia) in cooperation with the University of New South Wales (UNSW) and under the supervision of Professor Mahbub Hassan.

In this part we discuss the design and implementation of Android based 3G/HSDPA network bandwidth measurement mobile application. This application acts as a mobile sensor in a crowd sourcing system.

We use a network link bandwidth estimation technique called packet pair probing, which can easily be implemented on a mobile platform and we also justify why we have chose the specific methodology after reviewing the related literature. We also propose a measurement initiation process with the Measurement Server which allows the packet pair probing technique to reflect an accurate download bandwidth on the measurement.

We have calibrated and fine-tune the measurement tool so it can contribute optimally to the crowd sourcing system by addressing issues such as usability, data consumption and power consumption. We include geo tags in each

measurement we take and discuss the implementation issues addressed in the project. Finally, we introduce an algorithm which measures the download bandwidth in a timely fashion.

We study the behaviour of the measurements by changing parameters such as packet size and packet train length. The results obtained were evaluated by comparing them to a reliable commercial bandwidth estimation tool under the same environment. Given these results we conducted a number of hypothesis tests where we used the T-statistic as the test statistic under the null hypothesis.

Acknowledgments

It has been a great experience for me to write part of my final project at the Universidad Politécnica de Valencia (UPV) while doing a work placement for the company NUBESIS, and of course it has been amazing to write the other part at the School of Computer Science and Engineering (CSE) at the University of New Sout Wales (UNSW). I would not have been able to complete this project without the support and encouragement of many people. I would like to take this chance to thank all of them.

I would like to thank in first place the company NUBESIS for giving me such a great opportunity of working with them, specially to Javier de la Cueva Orts for all the meetings and all the support and comprehension that he offered me since I know him. I think he is a great person and a really hard worker.

In second place I would like express my gratitude to Juan Carlos Ruiz García, my supervisor at the UPV. Juan Carlos has been very supportive and he inspired me to go to Australia to do part of my final project. He is an incredible mentor and an extraordinary person, without him none of this would have ever been possible, he is a role model to me.

I would like to thank Professor Mahbub Hassan too, he gave me the opportunity to collaborate in his project and also for all the resources he provided me. Mahbub has given me insightful advice, constant support, patient guidance and inspiring ideas on the way to proceed in my project. He is a great man and an even greater mentor. His influence has been key to seeing out the successful completion of this project.

And of course, I would like to thank my co-supervisor Dr. Salil S. Kanhere for being so helpful and patient with me. Salil offered me good guidance and support through insightful suggestions and at times contructive critism that opened new avenues to explore and streamlining my research throught. He

has shared with me his knowledge and ideas and for this I am greatful. Salil is a great person and I'm very happy to have had the chance to colaborate with him.

I would like to thank also to my colleague Thilanka Panthie who is colaborating with me on this project. Thilanka has been very supportive and patient with me during the duration of this project. His knowledge and willingness to help have proved to be instrumental in the timely completion of this project.

Last but not least, I am very thankful to my family for their support and their unconditional love. Without their constant support and encouragement on many a night when the goal did not seem attainable, I would not have been able to complete this project. This report is dedicated to my parents, Miguel Martínez and Elvira Raga, they have been very helpful and they have always beleived in me. Thank you very much. I also would like to thank my girlfriend, Miriam Panero, who has made such tremendous efforts during these months and has given me a lot of strength to keep working in hard times. Miriam, I'm very lucky to have you in my life.

Contents

Chapter 1

Introduction

1.1 Report Description

This report is structured basically in two parts, as mentioned in the Abstract this project has been made in two different places. The first part is the practicall part, while the second is more based on researching to improve the first one. In the first part we talk about the Android application we developed for the company NUBESIS, PideCita. In this part we describe application's web based features along a full dedicated chapter, we focus on the code and on the visual parts of the application. The most significant feature of the application is that is a cloud computing application, this feature is explained in more detail in chapter 2.

The issues found when developing a web based application made us think about the second part. In this second part we design and develop an Android application to collect mobile networks information and upload this information to a server, all the collected information stores the geolocation where the measurements where taken, letting the server stablish patterns and identify areas with lower signals. During the report we will show the relation between both parts.

In the next section we justify our decision of choosing Android over other operating systems and we introduce the Android operating system, so the reader won't get lost with some terminology used in this report. After that we explain how the report is organized chapter by chapter.

1.2 Android

1.2.1 Justifying the system

When thinking about which system to choose, we had to think about which are the most popular in order to secure a large customer base. Nowadays the three top mobile systems are: Android, iOS and Windows Phone. Although RIM still has a considerable presence, its market share has been plunging. Our application is very concrete, it has access to location information, device information and network information. We envisage this application being distributed primarly amoungst students, we also hope that it would generate sufficient interest outside of this group in the general population. This fact (distribution), was one of the most significant in the choice of Android over the other systems. We now give a brief outline on how non-market applications are distributed.

iOS In order to develop for iOS, you at least need a machine running Snow Leopard. The software and the iOS SDK are available to download from the website and developers can test their applications in an emulator. When it comes to running the applications for testing with real devices, you are required to become a member of the iOS Developer Program. This has a total cost of US$ 99 per year and it allows you to register the device you will test your application with. It also allows you to create groups of people for testing, but if you want other people to test your application, you need to register their devices into your accounts.

Windows Phone When developing for Windows Phone, it's necessary to use Visual Studio 2010 Express for Windows Phone. App Hub is the website where developers register for membership as a Windows Phone developer. It is a mandatory requirement for those wishing to develop Windows Phone applications and will be the starting point for developers. Developers can begin by signing up for a Windows Live ID. Next, they can sign up to obtain the Windows Phone Developer Tools and associated licensing materials for developing applications using Visual Studio and Expression Blend. To register with App Hub you can use one of three methods: as a company, as an individual or as a student. With the first two you have to pay a membership fee of US$ 99 per year. Students don't have to pay a fee but they have to register with Microsoft DreamSpark and they must submit identification documentation. If developers want to test their applications with real devices, they must register these devices into their account.

Android Android provides an open development platform. Android applications can be developed using different integrated development environments (IDEs). Through their official website they recommend the use of Eclipse, which everyone can be downloaded for free. Android developers don't need to pay a fee to register for developing applications. This means that they don't need to register any device to test their applications, developers can use any Android device to test their applications, the only thing they need to do is modify one option in the phone's settings (Settings ->Applications ->Development ->USB debugging). This option allows the developer to debug their applications using a real device.

As we can see, when it comes to distributing our application without uploading it to an applications market (which comes with an economical charge) or registering users' phones in a website, Android makes it easy as these are not required. The only thing we need to do is send our application to users, so that they can install it in their phones. Another fact is that we don't need to spend money to get a developer's license. This does not mean that crowd sourcing is not possible with the other systems, here we are simply referring to the distribution of the application without the need to uploading it to a market.

1.2.2 Android system

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. Android is an open source system. Consider figure 1.1 with technical features of the Android OS.

Applications framework Android provides an open development platform, which offers the developers the chance to build very rich and innovative applications. Developers have full access to the same framework APIs used by the core applications (all written in Java programming language), so they can take advantage of the device hardware, access location information, run background services, set alarms, add notifications to the status bar, etc. The application architecture is designed to simplify the reuse of components, and any application can publish its capabilities, so any other application may then make use of those capabilities (subject to security constraints enforced by the framework). This means that any application can allow other applications to access its data, which is not a common feature with commercial applications.

Libraries Android includes a set of C/C++ libraries used by various components of the Android system. These capabilities are exposed to developers through the Android application framework.

Android RunTime Android consists of Java applications running on a Java-based, object-oriented application framework on top of Java core libraries running on a Dalvik virtual machine (DVM). The DVM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the *.dex* (Dalvik Executable, which is optimized for minimal memory footprint) format by the included "dx" tool. It has been written so that the device can run multiple VMs efficiently. The DVM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

Linux Kernel Android relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack.

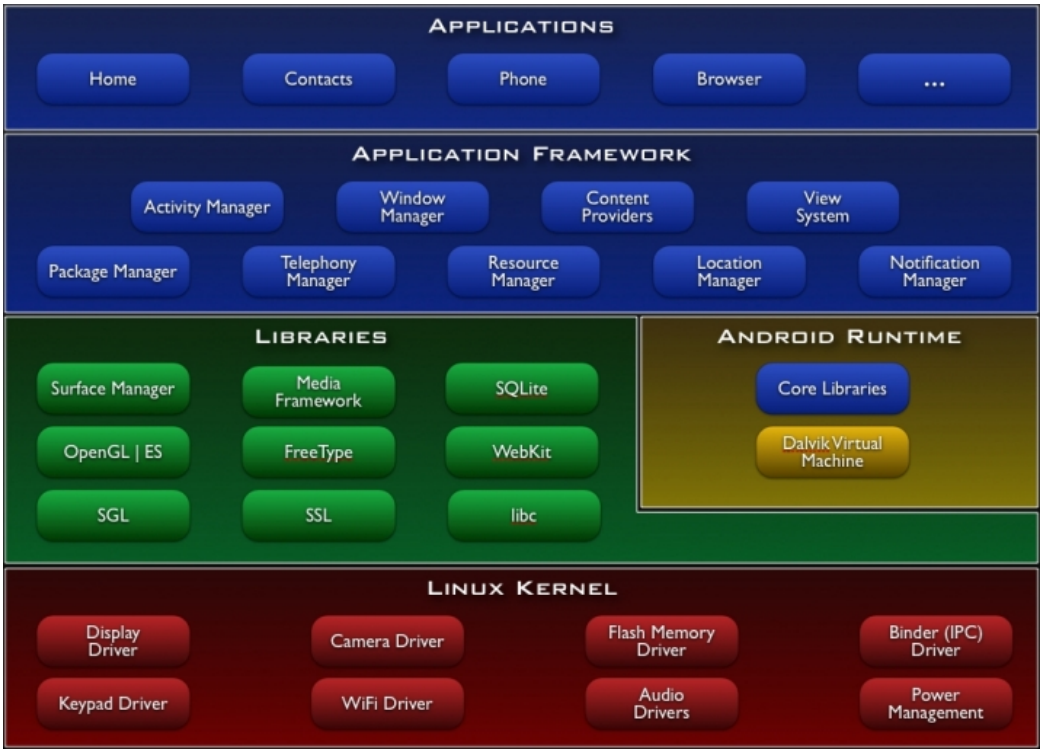


Figure 1.1: Android's system architecture

1.2.3 Android Activity

We introduce briefly the Android class because we will refer to it later in the project. For further information about this application's class the reader should refer to [1].

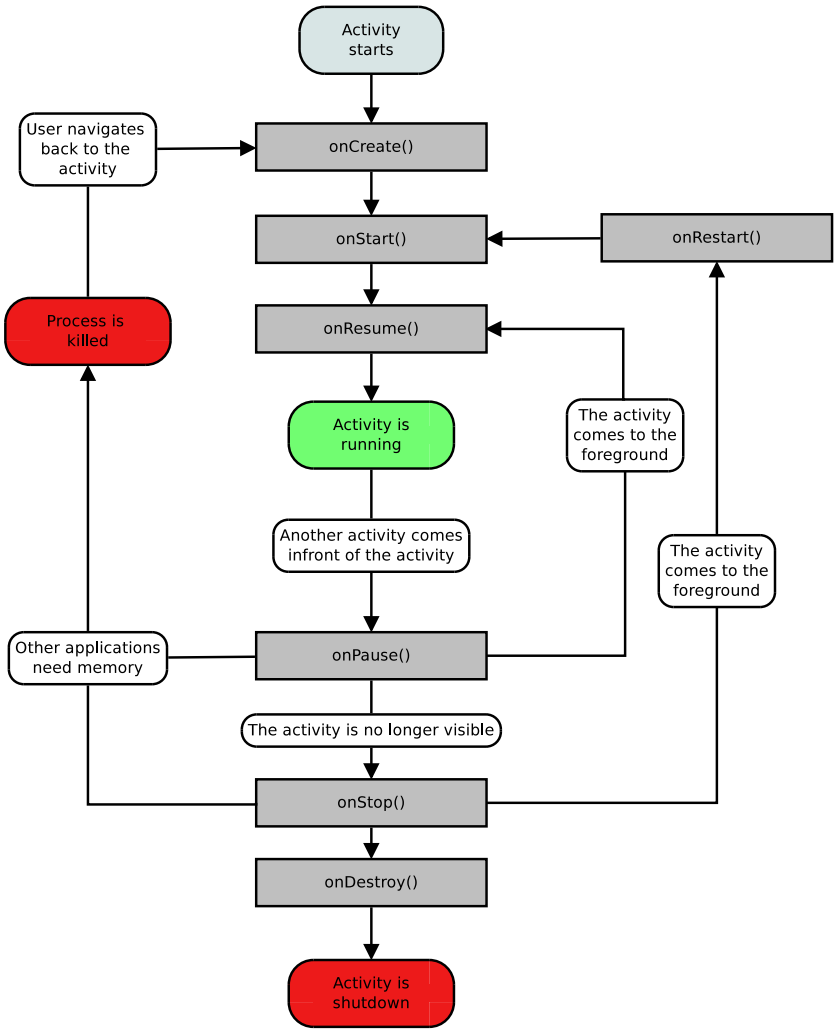


Figure 1.2: Activity lifecycle

An *Activity* is an application component that provides a screen with which users can interact in order to do something. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows. An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main"

activity, which is presented to the user when launching the application for the first time.

Activities' changes are notified through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state—whether the system is creating it, stopping it, resuming it, or destroying it—and each callback provides the developer the opportunity to perform specific work that's appropriate to that state change. This lifecycle is shown in figure 1.2, our main class in the application is a subclass (extends) of the *Activity* class.

1.3 Cloud Computing

1.3.1 Description

Cloud computing (CC) is a new way of computing that has its bases on the Internet.Through the Internet the shared resources are provided to computers and other devices as services. CC is a change in the programming paradigm where the information is stored permanently in servers and accessed by temporary clients through the Internet.

Cloud computing describes a new supplement, consumption, and delivery model for IT (information technologies) services based on Internet protocols, and it typically involves provisioning of dynamically scalable and often virtualized resources.

Cloud computing allows to develope applications based in services allocated externally, on the web. This way of computing provides ubicuity, the information can be accessed from anywhere with an Internet connection. The ubicuity of the information makes things easy for end-users, they can access the same information from any device such as a computer or a smartphone because it is centralized.

1.3.2 History

The term "cloud" is used as a metaphor for the Internet, based on the cloud drawing used in the past to represent the telephone network, and later to depict the Internet in computer network diagrams as an abstraction of the underlying infrastructure it represents.

The concept of CC started with big scale Internet services providers like: Google, Amazon AWS and others that build up their own infraestructure. Among them a new architecture appeared: a system with distributed horizontally resources and introduced as IT virtual services massivelly scalated and managed as combined and configured resources in a continuous way. Amazon played a key role in the development of cloud computing by modernizing their data centers, which like most computer networks, were using as little as 10% of their capacity at any one time, just to leave room for occasional spikes. Amazon initiated a new product development effort to provide cloud computing to external customers, and launched Amazon Web Service (AWS) on a utility computing basis in 2006.

This architecture model was inmortalized by George Gilder in his article for the *Wired* magazine in 2006, *The Information Factories* (available on line in here: `http://www.wired.com/wired/archive/14.10/cloudware.html`). The servers farm he wrote about had a similar architecture to the "grid" processing, but this new model based on the cloud was aplyed to the Internet services.

In early 2008, Eucalyptus became the first open-source, AWS API-compatible platform for deploying private clouds. In early 2008, OpenNebula, enhanced in the RESERVOIR European Commission-funded project, became the first open-source software for deploying private and hybrid clouds, and for the federation of clouds. In the same year, efforts were focused on providing QoS guarantees (as required by real-time interactive applications) to cloud-based infrastructures, in the framework of the IRMOS European Commission-funded project.

1.4 Report organization

In this section we present a chapter by chapter overview of the rest of the report.

In Chapter 2, we discuss the related work we used as a reference for our work. We discuss the studies based on techniques to estimate the connection bandwidth, which we used to develop our Bandwidth Measurement tool. We introduce a commercial tool used to tests our results and we conclude arguing the ideas we got from the related work.

Chapter 3 is related with the Android application we have developed in the first part of the project. In this chapter we talk in detail about the security problems we found when authenticating with the PideCita website. We explain the main methods and classes used in the application, we also discuss the integration of different APIs to our project, Google Maps and Facebook. And we conclude the chapter showing in detail the functionality of the application.

Chapter 4 introduce the problem found with the PideCita application that lead us to work on the second part. This chapter introduce the problem we found with the mobile networks and explain how this problem affects the developed application. The chapter conclude with a section that introduce the solution developed and studied during the second part of the project.

In Chapter 5, we discuss the measurement tool we developed for the second part of the project. We first give an introduction about the tool and its components. Then we talk about its different components, client and server, focusing on what they do and how they operate. At the end we present some performed tests and we discuss the obtained results.

Chapter 6 is related with the Android application we have developed in the second part. In this chapter we explain all the parts that compose the application. We start talking about the way we store the information obtained with the application. We explain the different components and functionalities we need to obtain from the device, and how we get them. We discuss about the issues we had to face while developing the application. We finish talking about the main functions that we developed to achieve our needs, focusing on their individual and group behavior.

Chapter 7 is an attempt to complete the jigsaw puzzle by connecting the dots between the techniques used in the second part of the project and the final outcome of the same, and also talking about the benefits of integrating these techniques into the first part of the project. We discuss the main aims of this whole project, briefly outlining why we choose Pair Packet probing over other techniques. We also make reference to our main findings and

finally conclude by remarking on the future direction of our research.

Chapter 2

Related work

2.1 Introduction

In this chapter we talk about the related work we make reference to in this report. We discuss studies about techniques to estimate the connection bandwidth which we used to develop our Bandwidth Measurement tool. We also introduce a commercial mobile application that has significant features related to our work. Packet pair probing techniques are introduced as network path capacity estimation methodologies ([3], [4]) using dispersion of two back to back packets [2]. These same techniques can be used to estimate the network narrowest link throughput which is the link between the base station and the network end user in our case.

2.1.1 Pathrate

Pathrate was introduced by Dovrolis et.al [3] as a tool which is based on packet pair dispersion techniques. Pathrate uses UDP probe packets to measure the narrowest link capacity. In order to initiate a measurement process it uses a TCP connection as the control channel. Authors have justified their choice of UDP probes as the probing packets instead of TCP based probes such as ICMP and TCP-FIN packets, saying that employing such packets will affect the bandwidth measurements because the reply probes are forwarded on the reverse path from receiver to sender. Successive packet pair probes used in the Pathrate tool have at least RTT difference between them to isolate each packet pair on the way from sender to receiver.

Accuracy of Pathrate compared to CapProbe is limited due to its inaccurate readings in high bandwidth paths with narrow links between 640Mbps to 1000Mbps due to dispersion measurement noise at the receiver. Authors

pinpoint that such noises occurs due to application layer time stamping when the bandwidth is much larger than mentioned above. And authors suggest that a machine with higher time resolution (much faster processor) or OS level time stamping could avoid such noises at the receiver by precision time stamps. Also, Pathrate provides inaccurate readings in highly congested or loaded paths because of the large probing packets it employs, it also encounters additional dispersion due to network cross traffic.

2.1.2 Cap Probe

CapProbe is a link capacity estimation tool for wired and wireless links. Introduced by Kapoort et.al [4], which is based on packet dispersion techniques as described in section 2.1.1, and combined with an error filtering technique/parameter called minimum delay sum. *Minimum delay sum* is the the minimum time dispersion between any packet pair out of all the packet pairs and the authors argued that such packet pair is not distorted by cross traffic induced queuing and reflects accurate narrowest link capacity estimates under the assumption that in a packet train there will be at least one such packet pair. Authors have used a modified version of the ping utility (IPUtils open source software) which can send a packet pair instead of one packet. During the capacity calculation stage the program waits for ICMP replies (packet pair) to calculate the narrowest link capacity.

Figure 2.1 shows the CapProbe's packet dispersion technique for calculating narrowest link capacity. It is based on the idea that if two consecutive packets are the same size then the transmission delays are the same for both packets within a link.

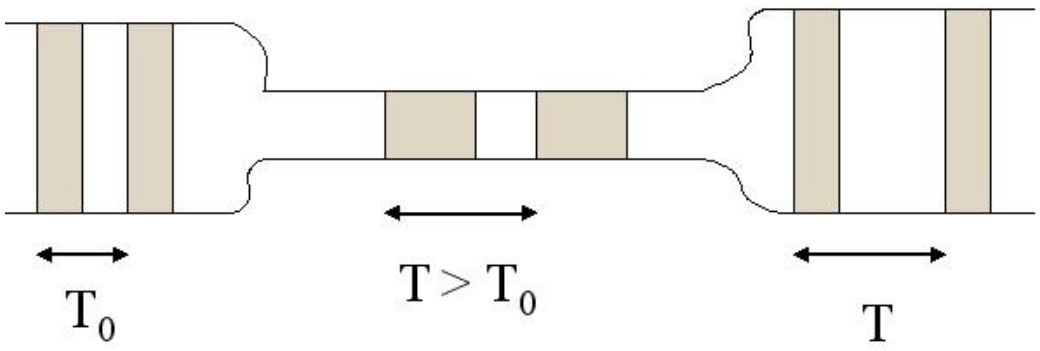


Figure 2.1: Packet Pair Dispersion Technique Used in CapProbe [4]

Authors argued that a dispersion of a packet pair from source to destination would be compressed or expanded due to cross traffic and hence they provide inaccurate capacity estimations (figure 2.2). The minimum delay sum from at least one packet pair would be enough to measure the capacity of the narrowest link.

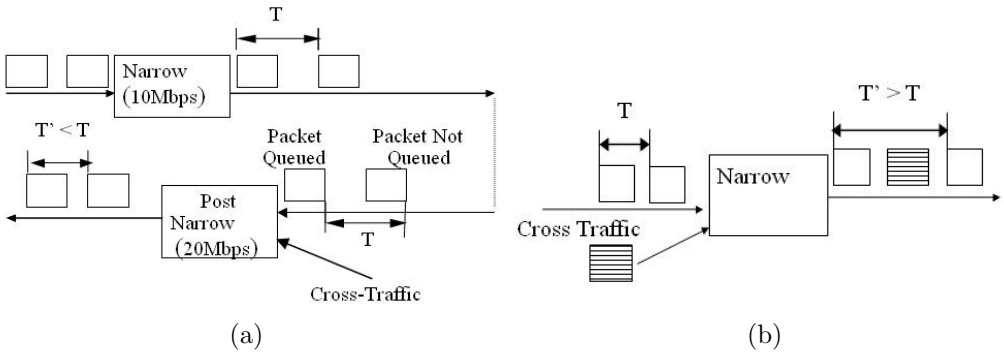


Figure 2.2: (a) Over estimation and (b) under estimation of capacity [5]

2.2 Mobile applications

2.2.1 Speedtest.net mobile

Speedtest.net mobile (ST) is a commercial application developed for Android and iOS devices. This application is the native Android version of a very popular broadband speed test on the internet [6]. It operates using a large global infrastructure to minimize the impact of internet congestion and latency. This application according to the Android Market has been installed by one to five millions users. It is one of Android's top free applications in the Tools section.

Once you begin a test with this application, it tries to get your GPS information via the location service on the device, in case it is unable, it uses the GeoIP information provided by $MaxMind$. $MaxMind$ provides its geo location technology through the GeoIP brand. They have a GeoIP database which can be accessed from different OS.

The tests performed in this application consists in three parts:

1. Latency test

2. Download bandwidth
3. Upload bandwidth

We only focus on the "Download bandwidth" part to compare it with our data. While doing a test, the user can visualize a real-time graphs of throughput during the tests. The data collected from the tests can be exported to CVS format and sent by email so users can use the results, which makes it very helpful to compare results.

2.3 What we have learn from the Related Work

Below we endeavour to explain why application such as ST or measurement tools like CapProbe and Pathrate can not be used for crowd sourcing.

The user should actively participate in order to take measurements from the ST application which consumes valuable time. From our observations a typical ST test will take between eight to twelve seconds depending on the network parameters at the time of the measurement.

CapProbe uses ICMP packets and such TCP based measurements would not reflect the same bandwidth measurements suitable for streaming media.

On the other hand *Pathrate* uses UDP packet sizes between 550 bytes and 1500 bytes with 500 to 1000 packet pairs for each individual test which could sink the mobile data plans of contributors demotivating them.

For example, ST provides us the service of network bandwidth measurement and users can contribute to the ST's measurements archive which ST uses for commercial purposes such as sharing bandwidth measurements with network providers and related services.

As discussed with regard to Pathrate, in order to timestamp received packets accurately we have to reduce the overhead at the receiver. During the Measurement tool development stage we have identified that our receiver program is burdened with high computational overhead leading to inaccurate timestamps and resulting in large variations of throughput measurements. The packet pair probing technique could underestimate or overestimate due to network path queuing. This may result in higher or lower measurements of the actual rate.

Chapter 3

PideCita Application

3.1 Introduction

This application is a cloud computing client developed for Android. This application is the mobile version of the cloud computing service offered in this website, *http://www.pidecita.com*. The application has been developed under the supervision of **NUBESIS S.L.** (owners and developers of the previous website). This website allows the users to apply for an appointment in any company registered in the system. The companies belong to different areas in the industry, the user's can choose from medical companies to hairdressers. This website allows users to book for an appointment without the need to contact the company because all companies provide their available timetables, so a user can choose between several companies the one that fits better to his/her schedule.

The aim of the application is to make easy and nice the interaction of the users with the system through their mobile phones. In this chapter we explain the different parts of the application and their interaction with the main system. We also talk about the issues we've found in the development process and how we solved them.

3.2 Certificates and security

3.2.1 Definition of the problem

Due to the fact that this application is based on a cloud computing environment we need to access to the information stored in the server. To obtain the information we need to do GET requests through HTTP to the server

https://docs.nubesis.com/bookitit_android/. The information is structured in XML files.

The main problem resides in the fact that the application needs to accept the server certificate as a trusted certificate. Every request to the server has to contain authentication fields to avoid malicious attacks but Android's HTTP libraries are the 4.x version of the Apache libraries, meanwhile Java usually uses the 3.x version. The new version of the libraries has some modifications that complicated a little bit the work, we explain all the problems in next sections.

3.2.2 Preparing the certificate

The certificate is a document with an electronic signature from an entity allowing anyone to recognize the entity and avoiding impersonations.

Keytools allows us to administer our keys and certificates to use them when necessary to authenticate in servers. We can create a "keystore" (KS) where we can store our certificates as trusted certificates, they are considered as trusted because a public key is attached to this cert and it can be verified anytime needed.

```
...
#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
security.provider.7=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.8=sun.security.smartcardio.SunPCSC
security.provider.9=sun.security.mscapi.SunMSCAPI
security.provider.10=org.bouncycastle.jce.provider.BouncyCastleProvider
#
...
```

Code 3.1: List of providers

When adding our certificate to our KS Java uses JKS (Java Keystore) by default and that is a problem when using the KS in Android. Android uses by default BC (Bouncy Castle) as its KS provider, so the easiest way to solve

this problem was to use the same KS provider when storing our certificate in our KS when using the *Keytools* tool.

The only thing needed is to download the .jar file with the required libraries and modify the "java.security" file that can be found in our Java folder (in Windows XP: Program Files\Java\jre\lib\security\) and add the provider to the providers list, it would look like figure 3.1.

After that we have to navigate to the Java folder where the keytool program is (this is only necessary in Windows) and execute in the terminal the following code:

```
> keytool -importcert -v -trustcacerts -file \path_to_cert\certificate.crt"
-alias \Alias for the cert" -keystore "keystore_path\myKeystore.bks"
-provider org.bouncycastle.jce.provider.BouncyCastleProvider -providerpath
"path_to_jar/bcprov-jdk16-145.jar" -storetype BKS -storepass mysecret
```

Code 3.2: Storing the certificate

If the KS does not exist the program create a new one in the specified path "keystore_path\myKeystore.bks". The **-storepass** refers to the password that is going to be used for the KS and that we will need to access it. The instructions in 3.3 let us check the information contained in our KS.

```
keytool -list -keystore "keystore_path\myKeystore.bks" -provider
org.bouncycastle.jce.provider.BouncyCastleProvider -providerpath
"path_to_jar/bcprov-jdk16-145.jar" -storetype BKS -storepass mysecret
```

Code 3.3: Storing the certificate

Once the KS is created we import it to our Android project, the right path to import it is inside our raw folder, in this case would be *res\ raw\ docsnubesis.bks*. The extension of the KS (bks) makes reference to the provider used in the creation.

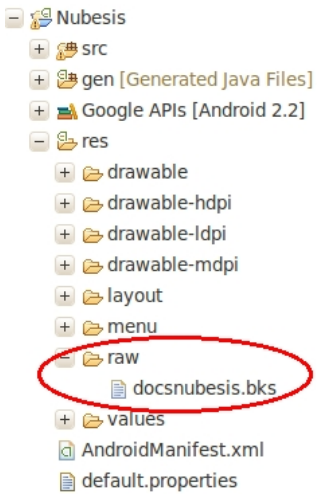


Figure 3.1: Keystore in raw folder

Now that the KS has been created and is in our project, we have to access it to create a HTTPS socket that we will associate to our HTTP client.

```
...
DefaultHttpClient httpclient = new DefaultHttpClient();
KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
InputStream in =
    this.getResources().openRawResource(R.raw.docsnubesis);
try {
    trustStore.load(in, "nubesis".toCharArray());
} finally {
    in.close();
}
SSLSocketFactory socketFactory = new SSLSocketFactory(trustStore);
Scheme sch = new Scheme("https", socketFactory, 443);
httpclient.getConnectionManager().getSchemeRegistry().register(sch);
...
```

Code 3.4: Use of the certificate to create the socket

3.2.3 Authenticating the request

When doing a request to the server we need to authenticate the request by attaching the authentication code of the request in a header. The message authentication code (MAC) is obtained by using Hash-based Message Authentication Code (HMAC), which is an specific construction to calculate

the MAC in combination with a cryptographic has function and a secret key. The cryptographic hash function used is MD5 (Message-Digest Algorithm 5) that is a widely used. It produces a 128-bit (16-byte) hash value and tipically expressed as 32-digit hexadecimal number. The resultant algorithm is called HMAC-MD5 and it allows the server to verify the integrity and authenticity of the message.

The different HTTP libraries that Android uses compared with other Java programs turned out to be a problem at the beginning. The password request from the server is obtained by applying the HMAC-MD5 algorithm to the address we want to access at the server, for example, if we access to `https://docs.nubesis.com/bookitit_android/api/companiestag/dentistas/11/20`, our password will be the result of calculating the HMAC-MD5 to "companiestag/dentistas/11/20" with our secret key. The way to obtain the correct 32-digit hexadecimal number differs a little bit when doing it in Android, in Code 3.5 we can see how it's done.

```
public String HMAC(String values, String myKeyString) throws
     InvalidAlgorithmParameterException, NoSuchProviderException {

     String output = "";
     try {
         byte[] key = myKeyString.getBytes();
         byte[] sms = values.getBytes();
         SecretKey skey = new SecretKeySpec(key,"HMAC-MD5");
         Mac mac = Mac.getInstance("HMAC-MD5","BC");
         mac.init(skey);
         mac.reset();
         mac.update(sms, 0, sms.length);
         byte[] digest = mac.doFinal();
   // convert the digest into a string

         int size = digest.length;
         StringBuffer h = new StringBuffer(size);
         for (int i = 0; i < size; i++) {
            int u = digest[i] & 255; // unsigned conversion
            if (u < 16) {
               h.append("0" + Integer.toHexString(u));//$NON-NLS-1$
            } else {
               h.append(Integer.toHexString(u));
            }
         }
```

```
        output += h.toString();
    } catch (InvalidKeyException e) {}
      catch (NoSuchAlgorithmException e) {}
    return output;
}
```

Code 3.5: HMAC-MD5

In *values* we have the address we want to access and in *myKeyString* we have our secret key. So we declare an object of the class SecretKey (skey) using our secret key and indicating the used algorithm. We define a MAC object (mac) with the **HMAC-MD5** algorithm and the **BC** provider. After initializing the *mac* variable with the *skey*, we add the message we want to calculate the hash for. With *mac.doFinal()* we obtain the resultant 16-bytes array and then we convert it into a 32-digit hexadecimal number string.

The server we are trying to access uses a **RESTful** architecture and a **BASIC** authentication scheme, with the HTTP libraries Android is using the definition of the classes and the way to add the user and the password to the request is different from how it is done in the Apache 3.x libraries. The confusion about this topic seemed to be quite big after reading through a lot of forums dedicated to Android topics, some people even adopt the position to import the old libraries to their projects to make it work as expected. The proper way to do it in Android is configuring the authentication scheme of the request, as it can be seen in Code 3.6 what we do is obtain the authentication scheme of type “basic” from the parameters set for the request and add the credentials that contain the user and the password as a header to the GET request.

```
HttpParams params = new BasicHttpParams();
HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
HttpProtocolParams.setContentCharset(params,
HTTP.DEFAULT_CONTENT_CHARSET);
HttpProtocolParams.setUseExpectContinue(params, true);
HttpClientParams.setAuthenticating(params, true);
Credentials cred = new UsernamePasswordCredentials(username, password);
DefaultHttpClient httpclient = new DefaultHttpClient();

...
/** Missing code**/
...

HttpGet httpget = new HttpGet("https://docs.nubesis.com/"+
   "bookitit_android/api/companiestag/dentistas/11/20");
httpget.addHeader("Accept", "application/xml");
AuthSchemeRegistry aut = httpclient.getAuthSchemes();
AuthScheme basic = aut.getAuthScheme("basic", params);
httpget.addHeader(basic.authenticate(cred, httpget));
try {
   ResponseHandler<String> responseHandler=new BasicResponseHandler();
   String responseBody=httpclient.execute(httpget,responseHandler);
   return responseBody;
}
catch(Exception e){
   Toast.makeText(this, "Request Failed: "+e.getMessage(),
   Toast.LENGTH_LONG).show();
}finally {
// When HttpClient instance is no longer needed,
// shut down the connection manager to ensure
// immediate deallocation of all system resources
httpclient.getConnectionManager().shutdown();
}
return "";
}
```

Code 3.6: BASIC authentication in a RESTful server

3.3 Methods and classes

3.3.1 Classes

The information about the companies and the users are not stored locally on the device, the way to acces this is information stored in the cloud must be accessed through the Internet. The requested information can refer to five

different kinds of objects: User, Company, Service, Agenda and Booking. Every class name is representative for each object, here we explain briefly the use of each one of these classes.

CCompany This class contains all the information about a company like: name, address, geographic location, description, etc ...

CUser The information related to users makes reference to all the details in their profiles.

CService When applying for an appointment is necessary to choose the service we need. For example, when applying for a dentist, we need to specify if it is a mouth cleaning or a simple visit. Each service has contains information about the kind of service and its duration.

CAgenda In some companies there will be more that one professional that can do a service, an agenda represents the times during a day a professional is available to receive a client.

CBookings When users are logged in they can check their incoming appointments or they old ones, this class stores the information for one appointment: company name, professional, kind of service and date of the appointment.

The class *CConections* contains all the methods performed to obtain the required information from the cloud. The information requested is received in XML format and parsed into objects of the previous mentioned classes to show it to the user. All the implemented methods follow the same pattern, the function showed in Code 3.7 represents the method "get_Services", all the methods are explained in next section.

Every method receives the required parameters to build the url. This url, the client and the parameters of the client are the input parameters of the *make-call* function. This function calculates the hash code for the url as shown in Code 3.6, prepare the request and execute it, the result of the request is returned and passed as a parameter to the parse function. Each method has its own parse function that returns the information tho the main program.

There are more classes in the project, some of them are explained in further sections, the rest are not mentioned for space reasons.

```
public ArrayList<CService> get_Services(String p_sClientId,
   int p_sCompanieId, DefaultHttpClient client, HttpParams params) {
      String requestToHash = "getservices/" + p_sClientId + "/"
      + p_sCompanieId;

      try {
         return parseServices(makeCall(client, params, requestToHash));
      } catch (Exception e) {
         e.getMessage();
      }
      return null;
}
```

Code 3.7: Web method example: get_Services

3.3.2 Methods

In this section we describe the methods defined in the server and used by the application to obtain and modify the information in the server. Table 3.1 shows for each method its name, the input parameters, a brief description of its function and the output.

Method	Input parameters	Description	Output
getfavorites	user id	list of companies that have been selected by the user as favorite companies	if the user doesn't have favorite companies returns null, otherwise the favorite companies
delfavorites	user id and company id	removes the company from the favorites list of the user	if the parameters are correct it will return true, otherwise false
addfavorites	user id and company id	add a company to the user's favorite companies list	returns true if everything is correct
getslots	user id, company id, day and duration of the service	obtains the agendas with the availability of the professionals in a company in a certain day	returns all the agendas if everything is correct, if not, returns an empty list
getbookshistory	user id	obtains the user's appointments historic	returns previous bookings if there is any, otherwise empty list
getbookspending	user id	obtains the future appointments for a user	returns the upcoming bookings if there is any, otherwise empty list
getservices	user id and company id	obtains the different services offered by a company	if the company has services returns a list of services, if not returns an empty list
user	user id	obtains user's personal data	returns an xml with the data of the user
uservalidate	user's email and password	validates the user in the system	if correct returns the user id, if not returns null

createevent	user id, agenda id, service id, starting day of the appointment, ending day of the appointment, starting time, ending time, name of the service and appointment description	creates an appointment	if correct returns true, if not returns false
companiestag	company type, latitude, longitude and ratio	obtains a list of the specified type of companies that are in the specified ratio of meters from the user's position indicated with the latitude and longitude	if correct, returns a list of companies, if not returns an empty list
validatefacebookuser	name, phone, facebook id and e-mail	obtains the PideCita user id linked to that facebook account	returns true and the user id

Table 3.1: Web methods

3.4 Google Maps use

3.4.1 Integrating Google Maps in the application

To use Google Maps in our application we need to follow three steps, this steps are focused for the use of Eclipse as our IDE (Integrated Development Environment), the first step may change for a different IDE.

In first place we select the correct API when creating our project. When using the ADT (Android Development Tools) plugin for Eclipse all the available API levels of Android has at least two versions, one with and one without the Google Maps API. We need to choose the one with the GM API.

In second place we must create a class that extends **MapActivity**, this is the Activity that contains the map. The layout of this Activity must include a view of the type "MapView", this view is the map onject and it can only be handled by a MapActivity. The way to do this is shown in Code 3.8.

```
<com.google.android.maps.MapView
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:apiKey="Api Key" />
```

Code 3.8: MapView for the MapActivity layout

The last thing we have to do is to obtain the Android Maps API key for our application, otherwise the MapView will not work properly. A single Maps API key is valid for all applications signed by a single certificate, so in the debugging state we use the Android debug certificate, this certificate is found in the SDK folder. The way to obtain the Maps API key is by registering the MD5 fingerprint of the certificate in this website `http://code.google.com/android/maps-api-signup.html`. To obtain the fingerprint we need to use *keytool* and the *debug.keystore* that we will find in the Android folder, Code 3.9 shows how it needs to be done.

```
$ keytool -list -keystore ~/.android/debug.keystore
...
Certificate fingerprint (MD5):
94:1E:43:49:87:73:BB:E6:A6:88:D7:20:F1:8E:B5:98
```

Code 3.9: Obtaining the certificate fingerprint

After introduce the fingerprint and accept the terms and conditions we get our Maps API key for that certificate and we have to copy it in the field "android:apiKey" in out MapView object in the layout. If we change the certificate then the map will not work, that means that when publishing an application in the market, a private suitable key needs to be obtained and we can no longer use the debug key when publishing the application, so we need to obtain a new Maps API key in that situation.

3.4.2 Interacting with the map

Once the map is integrated into our application we need to use it, between the information received about a company we also get a latitude and a longitude. In this section we explain how we managed to place in the map all the companies that a user gets when he performs a search, we first talk about the graphical part and show how it looks, and then we focus on the code. The purpose of the map in the application is to let the users inspect the real position of the companies in relation to their position.

When a user looks for a group of companies in a concrete field in the industry the application shows a list of those that are inside the specified distance ratio. The method that performs this action is "companiestag", explained in section 3.3.2. Figure 3.2 shows the resultant list in a search, this is the result of a request prepared for testing, so some companies are fake and their coordinates are random. Those companies with the yellow star are the ones

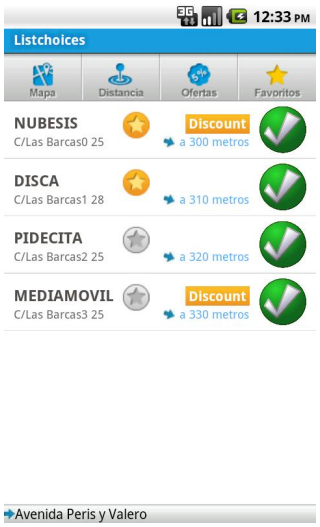


Figure 3.2: List of companies

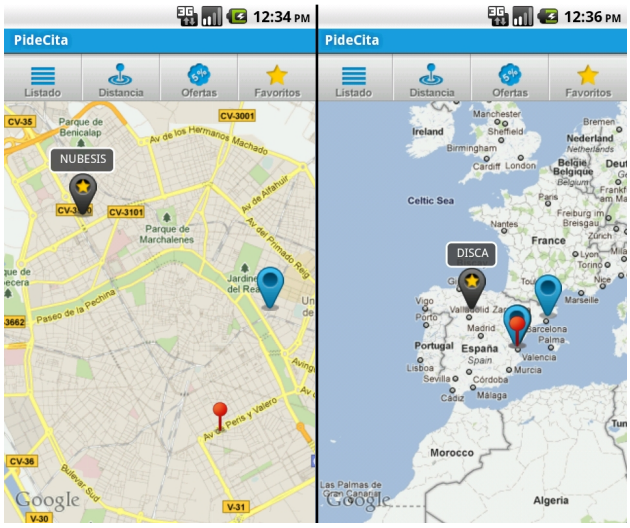
chosen by the user as favorite companies, for this reason there are two icons to distinguish between companies on the map screen, one to represent the favorite companies and the other one for the rest. If the user is not logged in then all the companies appear as not favorite.

(a) Company

(b) Favorite company

Figure 3.3: Company icons

Those companies that are considered as favorite by the user are displayed on the map with the icon 3.3b and the others with icon 3.3a. On the map screen when the user taps over an icon a bubble with the name of the company appears, a second tap on the bubble takes the user to the description screen of the company, if the second tap is in an area where there is no company then the bubble hides. In figures 3.4a and 3.4b we can see the representation of the companies on the list over the map. As mentioned before, these companies (their locations and data) are made up.



(a) Zoom in (b) Zoom out

Figure 3.4: Companies displayed on the map

To interact with the MapView we need to use a class named Overlay. The Overlay class is the class that place the items on the map. We have three classes involving the maps use: CMapChoices, MyOverlay and MyLocation. CMapChoices is the main Activity that displays the map and the top bar with the four buttons that let us choose if we want to display only those companies that has offers, those which are favorites for the user, or both of them. The other two buttons are one to the companies list and the other to change the distance ratio for the companies to be shown. The CMapChoices class extends the Activity class and it is the only one that can access to the resources of the project. To control the map we have an instance of MapView and another one for MyOverlay (*mapview* and *overlay* respectively). There is also a **List** of MyLocation, *mapLocations*, that allow us to obtain from a company only its name, location and know if it is a favorite company. The class CMapChoices is the one that receives the companies list when it is called, so we defined a function called *getMyLocations()* that iterates the array that contains all the companies and creates for each one a MyLocation object with the required information to fill *mapLocations*. In code 3.10 we can see how this function works.

```
public List<MyLocation> getMyLocations() {

/**
 * We fill the array of locations from the information of
 * the companies we have in the array @someCompanies, where
 * we store the given companies from our previous screen.
 **/

  if (mapLocations == null) {
      mapLocations = new ArrayList<MyLocation>();
      for (int i = 0; i < someCompanies.size(); i++) {
          mapLocations.add(new MyLocation(someCompanies.get(i)
          .get_sName(), someCompanies.get(i).get_dLatitude(),
          someCompanies.get(i).get_dLongitude(), someFavoriteNames
          .contains(someCompanies.get(i).get_sName())));
      }
  }

  return mapLocations;
}
```

Code 3.10: Function getMyLocations

When the CMapChoices Activity is *onCreate* the last thing we do is prepare the variable *overlay* to place all the companies on the map. To do this only two functions need to be used, one to create a new object of the class MyOverlay and the other to add the *overlay* to the map.

- overlay = new MyOverlay(this)
- mapview.getOverlays().add(overlay)

All the process to place the items on the map is defined in the MyOverlay class. When we create a new MyOverlay object we pass as parameter the reference to the CMapChoices Activity, doing so we can access the project's resources like images and views, that means we can have a reference to the MapView, because we will need to modify it and place the items over it. When the instruction "mapview.getOverlays().add(overlay)" is executed in CMapChoices then the function **draw** is called for the *overlay* variable. We override this function and define two new functions that will be called from this one: drawMyLocations and drawInfoWindow. The first one is the function that place all the companies on the map and the second one is the responsible of drawing the pop up bubble with the name of the company when the user taps over the icon. Next we explain the code of the function

drawMyLocations because its significance for the map, in code 3.11 we can see the code.

```
private void drawMyLocations(Canvas canvas, MapView mapView,
                             boolean shadow) {
   /** We draw the locations in the map, we also consider if
   * the location  belongs to a favorite company or not to
   * choose the image we want to draw. **/
  Iterator<MyLocation> iterator =
     mapLocationViewer.getMyLocations().iterator();
  Point screenCoords = new Point();

  while (iterator.hasNext()) {
     MyLocation location = iterator.next();
     mapView.getProjection().toPixels(
       location.getPoint(), screenCoords);

     if (location.isFav()) {
        canvas.drawBitmap(favoriteIcon,
            screenCoords.x - favoriteIcon.getWidth() / 2,
            screenCoords.y - favoriteIcon.getHeight(), null);
     } else {
        canvas.drawBitmap(bubbleIcon,
            screenCoords.x - bubbleIcon.getWidth() / 2,
            screenCoords.y - bubbleIcon.getHeight(), null);
     }
  }
}
```

Code 3.11: Function drawMyLocations

What we are doing is creating an iterator for the list of companies we want to place on the screen, each company is stored in a MyLocation object that makes reference to the company and its coordinates are stored as a GeoPoint, which is a class that represents a pair of latitude and longitude. We need to convert this coordinates to onscreen pixel coordinates, for that we use the function "mapView.getProjection().toPixels(location.getPoint(), screenCoords)", it needs two parameters, a GeoPoint, which contains the coordinates we want to convert and a pre-existing object of the class Point for the output, where the onscreen pixel coordinates will be stored. The object *screenCoords* contains the coordinates we need, the problem is that if we try to place an image using that point, that point would be the reference for the top left corner of the image, so we need to modify the coordinates to place the image as we want. The *if else* condition is to check whether to draw the

favorite or the regular icon for the company.

When the user interacts with the top buttons (Distance, Favorites and Offers) the set of companies can change, maybe we need to display only those with an offer, or only the favorite ones, or simply the set changes because the distance ratio has changed and there are new companies on the list. In all this situations what we do is to modify the main ArrayList that contains all the companies and call the function "redraw()", this function is the responsible of redrawing the canvas, so all the new companies will replace the previous ones.

3.5 Facebook

In the web version of the application, the user can log in using a Facebook account, so there is no need to register on the website and have a specific user and password for the website `http://www.pidecita.com`. The application pretends to offer the same facilities to the user, so we integrated a way to log in through Facebook. To use Facebook in our application we need an application id for Facebook, because the application is the same as the website both have the same application id. We let the user choose between log in with Facebook or with PideCita from the **Login** screen (Figure 3.5). When the user logs in with Facebook then we can access to the user's personal information necessary to validate the user against the user's database, we use the method *validatefacebookuser*(Table 3.1) to obtain the PideCita user's id linked to this account, if the user does not exist then a new user is created and the user's id is returned.

When the user log in correctly on Facebook then we proceed to obtain the information by doing requests using the Facebook API. The methods used are called synchronously and they will block the UI, so instead of that we use a class provided by the Facebook SDK called AsyncFacebookRunner, which perform asynchronous calls that do not block the UI. The method returns a JSON object that we need to parse in order to obtain the Facebook id of the user and the name, we will also need the phone number when using *validatefacebookuser*, but due to the fact that not everybody have a phone number in their Facebook account, what we do is obtaining the device's phone number if any, if the phone number can't be obtained then we set a default number. The default number is useful when creating a new PideCita user linked to

this Facebook account, but later on the user will need to change it because it is necessary that the users provide a real phone number in case the companies need to contact them.

Figure 3.5: List of companies

3.6 Functionality

3.6.1 Introduction

In this section we describe the transitions between the application's screens. There are several differences between the available options a user can use depending whether the user is logged in or not, because of that we only consider when a user is already logged in to explain the screen transitions but pointing out when necessary what a not logged in user can not do. We explain the process of applying for an appointment from beginning to end going through all the possible screens. After this we explain the authentication process for a user and the accessible screens related to the user's account. The data used for the application's screen captures showed during next sections is fake data prepared for test purposes.

3.6.2 Main screen

In this screen we have the icons (tags from now on) of the type of companies the user can search for. This screen contains four default tags: Sports,

Cosmetic, Health and Add. The first three make reference to a type of companies, with the last one the user can add more tags to the main screen, by doing this the user can do faster a search for a type of companies. In figure 3.6 we can see the process the user has to do to add the tag "Layers" to the main screen.

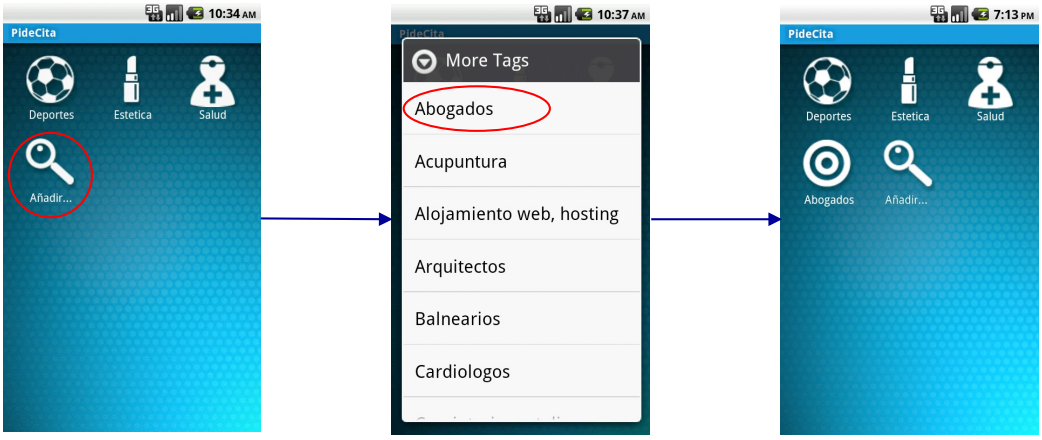


Figure 3.6: Add a tag to the main screen

These new tags added to the main screen can be deleted easily, the user only needs to press and hold over the tag and a pop up will appear with one question and two answers to choose, Q:"Do you want to delete the tag?", A: "Yes" and "No". When a tag is added to the screen then it is removed from the list of available tags, and when it is removed from the screen then it is added back to the list.

When a tag is pressed the application executes a request to the server to obtain the list of companies that match with the type indicated by the tag. This request is processed and the information is shown in a new screen called "Listchoices".

3.6.3 List and Map screen

The user is able to check the resultant companies either in a list form or in a map. Both screens have a top buttons bar with for buttons, three of them do not change between screens (Distance, Discounts and Favorites) but the other one does, the left button is used to change between these two screens. Here we explain the functionality of these buttons:

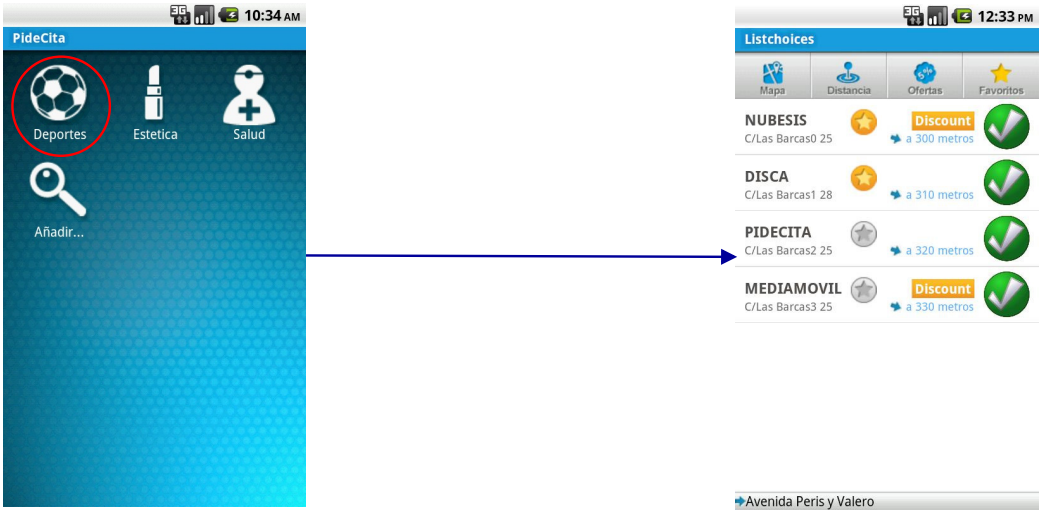


Figure 3.7: Performing a search

Distance
: This button is used to change the distance limit from our position. When this distance changes a new request is performed with the new distance.

Discounts
: Its function is to show only those companies that has an offer.

Favorites
: When pressed it shows only those companies among the list that are favorites for the user. This button is not enabled when the user is not logged in.

In figure 3.7 is visible the kind of information the user can see from a company in the list: name, address, distance from user, if it has a discount, if it is a favorite company and the company picture. The last one is represented in the test data as a green round ball with a white confirmation mark inside. If the user is not logged in then all the stars appear in grey.

For the next example we used a small group of companies that contains all the different possibilities, it is easier to see the difference on the map when a button is pressed with a reduced group. Figure 3.8 and figure 3.9 show the different transitions in each screen when the user press the buttons *Discount* or *Favorites*.

A user can see the detailed information of a company by clicking on it,

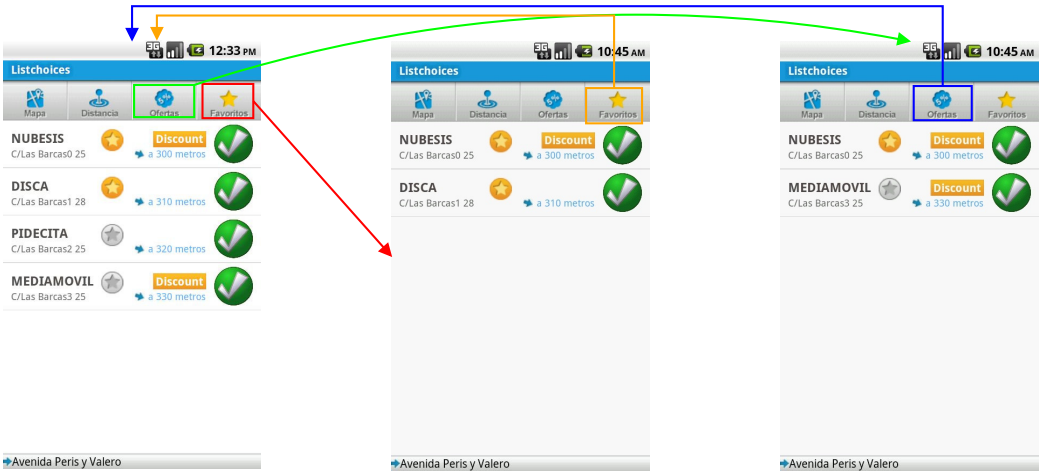


Figure 3.8: List screen transitions

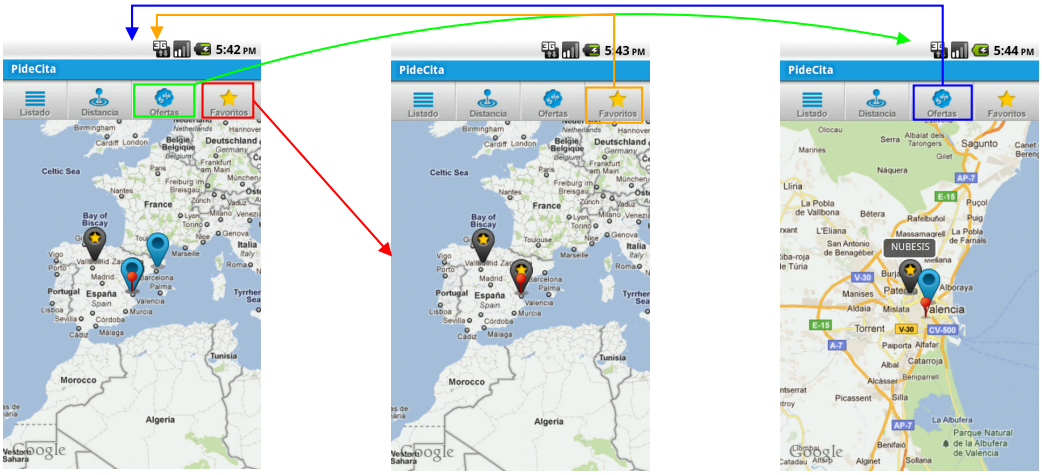


Figure 3.9: Map screen transitions

at the list screen the user only need to tap over the company row but in the map the user need to tap the icon of a company, after that the way to access to the information is by taping over the pop up that shows up right after the first tap. All the information is displayed in a new screen called “Ficha Técnica” (Company File).

3.6.4 Company File screen

This screen shows the information related to the selected company and also about the offer they have in case there is any. The user can check the details and decide whether to choose or not this company. A logged user can add

or remove the company to/from favorites by tapping the star next to the name, when the star is yellow the company belongs to the user's favorite companies group. A not logged user can't do it because there's no account to add or remove companies. When the user chooses a company then the booking button must be pressed to proceed.

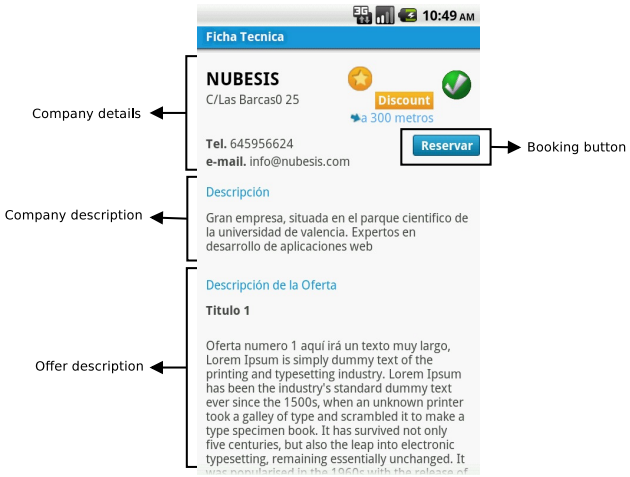


Figure 3.10: Company File screen

3.6.5 Services screen

At this point, the user has selected the company for an appointment. Most of companies can offer a number of different services, for example, if we want to do an appointment in a hairdresser we can choose between many different services we can receive: hair cut, do highlights, dye the hair, etc. In this screen we list all the services offered by a company, the user can scroll the list to check all of them if there are more services that do not fit on the screen. When the user tap over the service then this row will appear as marked, as we can see in figure 3.11, so when the service has been selected the only thing left to do is to press the button "Siguiente" (Next), this button will take us to the next screen, "Agendas".

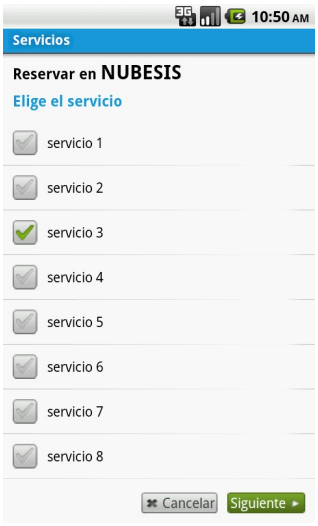


Figure 3.11: Services screen

3.6.6 Agendas screen

When the user selects a service then we perfom a request to obtain the agendas of the professionals. Each agenda received belongs to a professional and it contains his/her name and the available times on a day that this professional has to do the service. In this screen the user can navigate through the days using the grey arrows situated next to the date, every time the user wants to check a different day a new request takes place to obtain the agendas of the previous or next day. To select a day and a time for the appointment, the user only needs to navigate to the desired day and then tap over the wanted time, doing this we move forward to the last screen in this process, "Confirmación de la reserva" (Booking confirmation). In figure 3.12 we can see the transition.

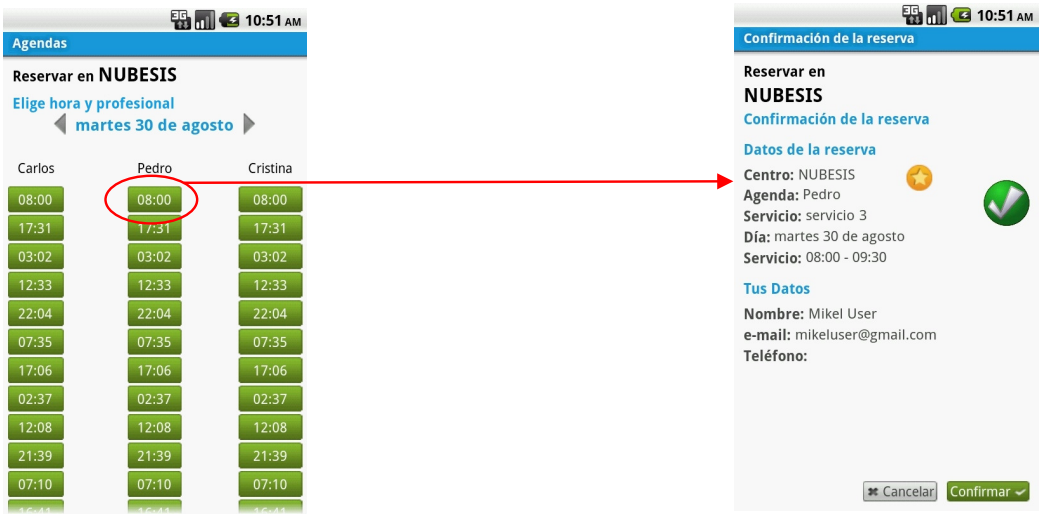


Figure 3.12: Selecting a date

3.6.7 Booking confirmation

This screen lets the user check all the information about the appointment before the confirmation. In this screen there are two main parts, one with all the details referring to the appointment and a second part with the user's details. If a user is not logged then this second part remains empty, if a user wants to confirm an appointment then he/she needs to be logged in, otherwise there is no other way to do it. In the situation a user is not logged in and tries to confirm an appointment, the user will be asked for identification and sent to the "Login" screen. In figure 3.13 we can see the example of a logged user that does not have a phone number registered in the account, this is a problem because all users that want to confirm an appointment with a company need to have a registered phone number in case the companies need to contact them. As can be seen when the user tries to confirm a pop up window appears requesting a phone number, right after the user introduce the number then he/she can confirm the appointment, if everything is correct and there are no problems during the confirmation, the request returns true and the application goes back to the main screen.

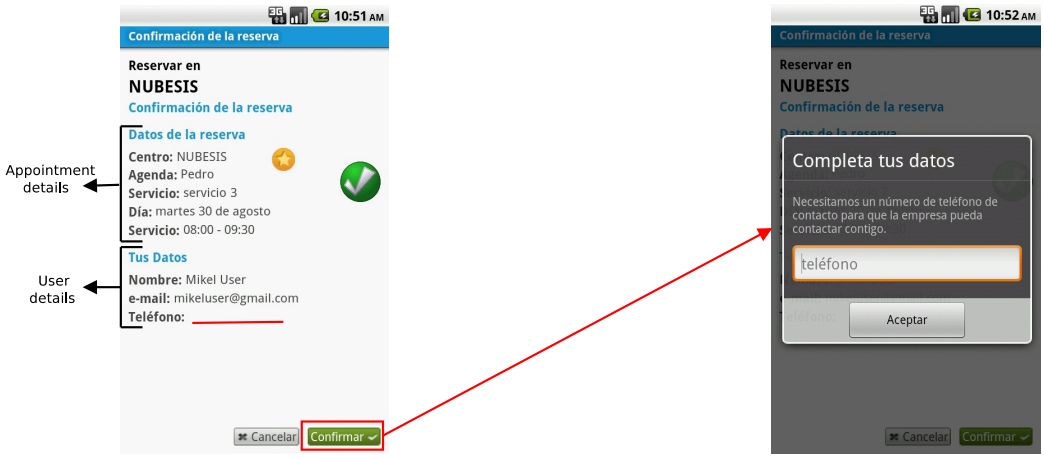


Figure 3.13: Booking confirmation

3.6.8 Log in process and user screens

As mentioned in previous section, when a user wants to make an appointment with a company using the PideCita application needs to have an account with the same website (`http://www.pidecita.com`). The users can have an account linked to their Facebook account as explained in section 3.5, or they can have their PideCita account, to do this they can register themselves through the website or with the application.

The log in option is present on the application menu, this menu is an emergent menu attached to the "Menu" button on the device, when this button is pressed once the menu appears, the second times disappear. The log in and log out options are represented by the icons in figure 3.14, this icons change in the menu depending if the user is logged in or not.

(a) Log out (b) Log in

Figure 3.14: Log icons in the menu

In figure 3.15 we illustrate the process of authentication or registration of a user, this process can be started from all the screens explained in previous sections. If the user logs in or create a new account successfully the application takes the user to the screen where the process started.

Figure 3.15: Login process

The menu has other icons, four of these icons are only accessible for logged users: "Mi perfil" (My profile), "Favoritos" (Favorites), "Reservas pendientes" (Upcoming appointments) and "Historial" (Old appointments). The other two are "Inicio" (Home) and the Log in/out button, the Home button let the user go back to the main screen at any moment, with the back button the user navigates to the previous screen, so with the Home button it is not necessary to keep pressing the back button to reach the main screen from an advance screen. The Favorites button take us to a screen similar to *Listchoices* but in this one all the companies listed are only those chosen by the user as favorite. This screen has the same top bar than *Listchoices* and *Mapchoices* screen but changing the Favorites icon for a Home icon. In this list the companies are more detailed than in a normal search and every company can be removed from the favorites list using the *Delete* button under the company's picture. The user can access to the map to see the location of the companies like in a normal search, from this map and from the favorites list the user can start the process of applying for an appointment too. The name for the other three options are self explanatory of what we are going to see. In figure 3.16 we can see the transitions for all the different menu options, of course as we need to be logged to access the options in the menu the "Log out" button is present.

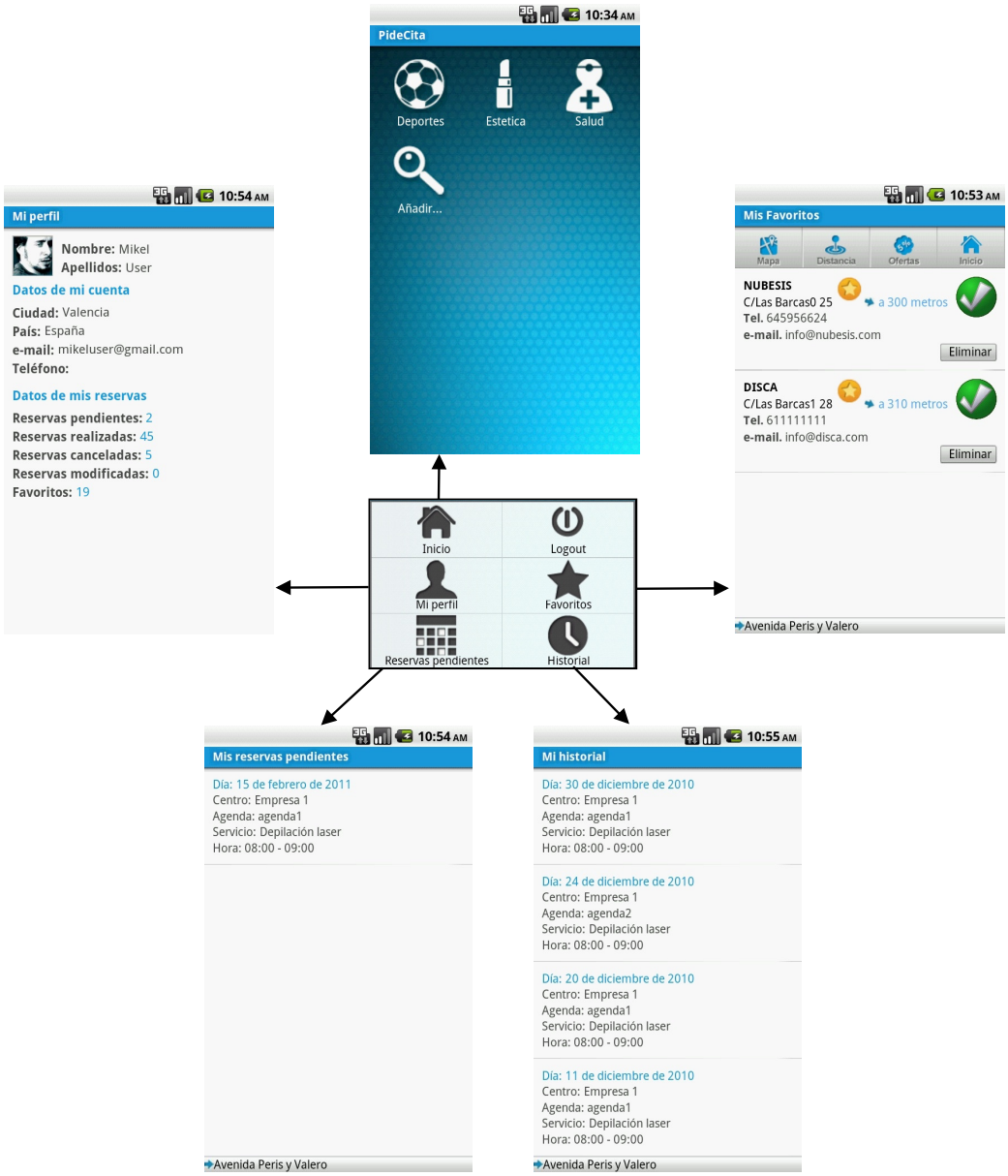


Figure 3.16: Menu transitions

Chapter 4

Connectivity issues

4.1 Background of the problem

During this last decade, the area of mobile phone devices has experienced a significant evolution. Mobile Networks (MN) have improved by leaps and bounds. The way current devices make use of MN has opened the doors to a new generation of interaction between users and their phones. This progress in MN has made the number of mobile devices increase very rapidly in the past few years. That lead us to a situation where there is a large number of users using their devices to access the Internet.

MN operators have tried to adapt to the constant evolution of mobile technologies in order to guarantee a higher customer satisfaction, sometimes this is not possible and users experience connection problems, which causes dissatisfaction. The lack of consistency in MN is due to environmental and technological limitations such as load variations, the number of cell towers, your surrounding environment etc. We are aware that these problems are more significant in some areas than in others. So users can not expect a good and consistent network coverage all the time. Inconsistent network behavior greatly affects users' *Quality of Service* (QoS), especially when they use applications that require a high use of data such as multimedia streaming services.

Network coverage is not always consistent due to the problems mentioned before. The number of cell towers in an area can vary significantly from one area to another. For example. mobile phone providers do not have equal coverage in the same country. This is influenced by how young a mobile provider is in a particular country and by the population in that area. Also,

some companies do not invest money in areas with a small population. For example, in Australia, the main cities have signal for every provider, however this is not the case in some small towns. The situation will likely continue unless companies decide to expand their presence uniformly throughout a region. So, considering an area with signal of a certain provider, when moving through that area, the signal reception can vary due to the problems we afore-mentioned. These problems affect the MN, particularly web based applications through users' QoS. This is due to the fact that applications can't know in advance these changes, so they can't prevent the consequences related to them.

4.2 How this problem affects our application

The Android application developed for NUBESIS, PideCita, is an application with cloud computing features, and for that reason one essential feature on the phone that must work properly is the Internet connection. In our application all the information is stored in a server and accessed from the device through HTTP requests, that makes things more comfortable for the user because all the information is available from anywhere and the risk of loosing something does not exist. The users can check all the appointments, modify their favorite companies or apply for new appointments either from the phone or from the computer.

When the users use their applications on the phone they expect them to be fluent and solid, if the it is not a web based application that depends only about the phone features, then the fluency between screens or the fact that something works wrong may depend only on the phone's hardware, and there's nothing we can do about that. But when it comes to applications with a continuous use of the Internet, then a bad connection can affect to the QoS the user expects from the application. In our case all the information displayed on every screen is obtained through requests to the server, and the time this request needs to obtain the information can be the difference between a fluent screens transitions or a slow one.

If we talk about the PideCita application, these problems can cause unexpected behavior on the application, for example, if the user wants to apply for an appointment and after several minutes thinking about which company to choose, the day and the time, if the connection is down when the user wants to confirm the date then this will not happen. In this kind of situations the users do not know if the problem is due to the application, the

server or just because the connection is down at the moment, the only thing the user cares about is about the time lost on applying for an appointment and the upraising frustration from not achieving the result. That kind of things can make users do not want to use the application, and this is not good for business.

4.3 A solution for the problem

We know our problem depend on the cell phones coverage, and that's something we can not change. What we can do is try to study them and prevent this lack of coverage in some areas, and this take us to the second part of the project. Applications can't find out by themselves the quality of the device's connection in advance, so our idea is to build a server that will contain information related to the MN signal for different providers in a certain location. This allows applications like multimedia streaming to take advantage of this information by preventing a lack of signal in the area the user is moving towards, thus they can adjust their data flow in order to minimize a loss in the QoS. All the information obtained is stored in a database at the server and applications will access this information through HTTP requests.

In order to collect this data, we have developed an Android application. The advantages of doing so is that we can use crowd sourcing to collect the data. By spreading the application, we can get a large number of users contributing to the database by collecting information about the MN signal in their respective areas. This application makes use of the GPS and several phone features that help us to collect all the information about the MN in a certain location. We built a server that we use to estimate the available bandwidth of the phone. During next chapters we focus on this application, how it works and explain its behavior.

Chapter 5

Bandwidth Measurement

5.1 Introduction

One of the main aims of this project is to be able to measure the download bandwidth (DBdw) in a certain location. Android doesn't provide any class to do so for a 3G connection, so we have developed a server in Java that supplies the client with packet trains of data which we will use to measure the download bandwidth that the user can get. In this next section we discuss how this server/client works and we talk about the techniques used to measure the bandwidth and the tests we performed to interpret our results.

A thing to consider when measuring the DBdw, is that the user is attached to a data plan. So one of our main consideration was the data consumption when trying to measure the DBdw because we want the users to use the application with minimal affect to much their data plan.

5.2 Measurement technique

5.2.1 Server part

The server will have to supply requests from different users, so we developed a multi thread server. The behavior of the server is shown in figure 5.1.

In the server we have a main thread that is always running and listening for new requests. In this main thread we open a *DatagramSocket* that listens through port "5555". The *DatagramSocket* class is for using UDP socket connections. Even though TCP is a more reliable protocol, when trying to establish a pure TCP connection with the *Socket* class from Java and

using 3G in the device, we saw that the connection was never establish in 3G although it was in Wifi. This is caused by the providers' NAT (*Network Address Translation*). Many mobile carriers use NAT and related technologies, so there is no guaranteed way to make a direct socket connection between two devices or a device and a PC.

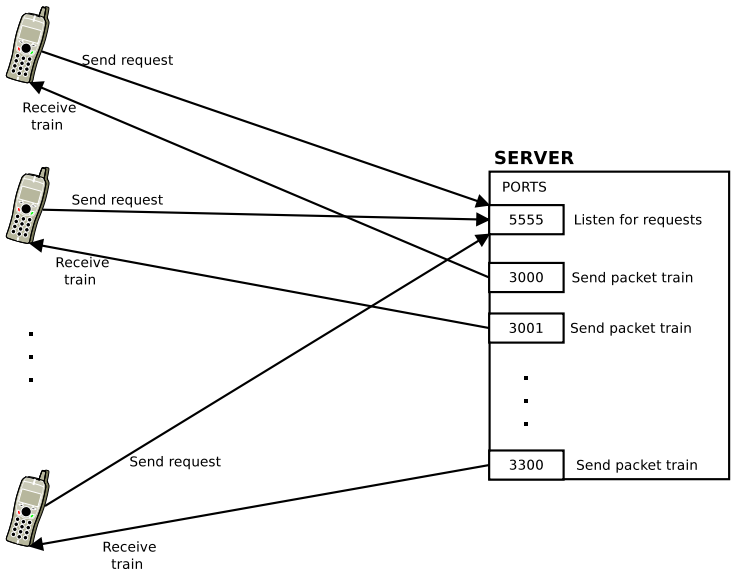


Figure 5.1: Multi thread server

As mentioned earlier, the server is requested to send a packet train of a certain size to the client and each packet with a specific size. The values for the possible number of packets in a train are 10, 20, 50, 100 and 200. The packet size is expressed in bytes and the possible values are 50, 100, 300, 500 and 1400. These values are stored in two different arrays, *npckts* for the number of packets in a train and *size* for the different packet sizes. The requests from client to server are send in UDP packets. When a new request arrives, we extract the data from the packet. The data contained in the request consist in two numbers, these numbers are indexes for the previous arrays. The first index is the position of the desired packet size and the second index refers to the train size.

To serve the clients requests we have implemented a runnable class called *Handler*. To create an instance of this class we need to pass it some parameters. The way this is done from the main thread is shown next.

```
DatagramSocket udpsocket = null;
try {
   udpsocket = new DatagramSocket(obtainport());
} catch (Exception e) {
   e.printStackTrace();
   System.out.println("Problems creating socket");
}

new Thread(new Handler(udpsocket,npckts[p],size[s],
        packet.getAddress(),data.substring(0, size[s]))).run();
```

Code 5.1: Handling clients requests

The parameters are described as follows:

DatagramSocket socket The socket created to send packets through a different port number than the main socket.

int packets The number of packets that we need to send (train size).

int size The size of each packet in bytes.

InetAddress address The IP address of the client. This address is obtained from the packet received in the request.

String data The data of the packets we are sending. In the client this data is not treated so this string is full of blanks. The number of blanks is the same than the number of bytes the packet's data must have.

As we can see in Code 5.1, we first create the socket in a port given by the function *obtainport*(), which gives an unused port between "3000" and "3300". The number of packets and the packet size is obtained from the arrays mentioned before, $npckts[p]$ and $size[s]$. In s and p are stored the numbers obtained from the client's request packet. From that packet we obtain the client's IP address with $packet.getAddress()$. The data for sending the packets is obtained from the variable *data*. This is a string created with a number of blanks equal to the maximum packet size possible, in our case this is 1400 bytes, hence it contains 1400 blanks. We use the "substring" function to pass a string that contains only the required size.

The function $run()$ in the $Handler$ class is responsible to perform the function of sending the packet train to the client. We only create one packet and we send it as many times as the train size. We don't need to create new packets because as mentioned before and as we explain in next section,

the client does not need to treat the data from the packet. The process for sending the packets is as follows.

```
public void run() {
  try {
       byte[] bufer = new byte[size];
       bufer = data.getBytes();

       DatagramPacket packet2 = new DatagramPacket(bufer, bufer.length,
            address, 5555);

       for (int i = 0; i < packets; i++) {
            udpsocket.send(packet2);
       }
  } catch (Exception e) {
       System.err.println("Error sending the train");
  }finally{
       udpsocket.close();
       udpsocket.disconnect();
  }
}
```

Code 5.2: run() function in Handler class

5.2.2 Client part

The client is developed in Java and is developed for running on an Android system. As we did with the server, we are using Java network classes to define client's behavior. This method is implemented in a class called *BandwidthMeasure* and it implements the *Runnable* interface. Now we explain how it works and what it does.

When the DBdw is calculated, the client creates a thread passing a new object of the *BandwidthMeasure* class which is created with three parameters. The first parameter refers to the number of packets we want to receive in a train of packets (train size), the second parameter refers to the size of each packet in the packet train and the last parameter is a reference to the application's main *Activity* that called the thread. As we mentioned before, the first two parameters are the values that we attach to the packet we send in our request, the server then services our request by sending a packet train with the train size we want with the requested size for every packet. The third parameter is used in order to access public functions defined in the main *Activity*. This particular function is called *addbdw(float bdw)*, its purpose is

to receive the estimated DBdw in the thread and store it in a global variable called *estimation*, this last variable will contain the estimated DBdw. This functions will be explained in next chapter.

We use the same arrays that the server uses (*size* and *npckts*) in order to know the size of the packet and the number of packets we are expecting. When the function *run*() is called, the first thing we do is initialize all the positions of *received_times* array to "-1" and all the positions of *ports* array to "0". The *received_times* array is of type "long" (because the arriving time is stored in nanoseconds, so we need long numbers) and it's used to register the arriving time of each packet, so with this information we calculate the delays between packets. The *ports* array is of type "int" and it's used to store the number of the sending port of each received packet. Their size is the same as the number of packets we are expecting to receive. Thereafter, we create and send the request to the server then we wait for the response. In Code 5.3 we can see the procedure to wait for the response.

We first set a timeout value for the socket of "time" milliseconds (this value is by default setted to 1 second). The reception of the first packet may take a long time due to the traffic, that is why the timeout for the first packet is higher than for the rest. After the first packet arrives, we change the timeout value to half a second, that value is established considering the delay between packets we would experience in a situation with a very low bandwidth. We initialize the three variables that we will use as conditions for the *while* loop. In *goodpackets* we count the packets we have received, later, we will need it inorder to calculate the bandwidth as can be seen in Code 5.4. The variable *init* is used to store the initial time of the request and *timeout* to record if the timeout occured. The conditions for the *while* loop to iterate are:

1. The number of received packets is lower than the expected. We stop looping when we achieve as many packets as we expected. This value is stored in *npckts* and $numofpac$ is the value of the parameter we received in the object's constructor and it refers to the train size.

2. When the timeout occurs, we set the *timeout* variable to true and we stop looping.

3. Knowing the initial time we can make sure that we are not receiving packets for longer than the specified time. So if the difference between the last packet received and the initial time is higher than ten seconds (10^9 nanoseconds) we stop receiving.

```
udpsocket.setSoTimeout(time);
init = System.nanoTime();

goodpackets = 0;
boolean timeout = false;

try {
   udpsocket.receive(pac);
   received_times[goodpackets] = System.nanoTime();
   ports[goodpackets] = pac.getPort();
   goodpackets++;
   udpsocket.setSoTimeout(500);
} catch (InterruptedIOException e) {
   timeout = true;
}

while (goodpackets < npckts[numofpac] - 1 && !timeout
&& (received_times[goodpackets - 1] - init) < 10000000000f) {
   try {
      udpsocket.receive(pac);
      received_times[goodpackets] = System.nanoTime();
      ports[goodpackets] = pac.getPort();
      goodpackets++;
   } catch (InterruptedIOException e) {
      timeout = true;
   }
}
```

Code 5.3: Receiving a packet train

What we do is for every packet that arrives to our system we register its arriving time in *received_times* and the sender's port number in *ports*. Every arriving time and port number for each packet are stored in the position of their respective arrays which matches the arriving order of the packets, considering the first packet number "0".

When we finish waiting for more packets, we close and disconnect the socket. Inorder to calculate the bandwidth we check if certain conditions are fulfilled. If the DBdw is low, the time to receive one hundred packets is very high, so if we stopped receiving because the ten second condition occured, then we make sure that all packets belong to the same train and then we calculate the DBdw. If this is not the case and the number of packets is higher than eighty (eighty packets give similar values to one hundred), then we calculate the bandwidth. In both situations when we calculate the DBdw we pass it to

the main *Activity* through the function mentioned before. This call uses the object *mainActivity*, which is the reference to the main *Activity* received as a parameter when the *BandwidthMeasure* object was created. This call is as follows:

mainActivity.addbdw(calculateBDW(received_times))

The parameter *received* refers to the array mentioned before, *received_times*, the parameter *ports* refers to the array *ports* and the parameter *packets* makes reference to the variable *goodpackets*, which contains the number of packets received. *Sizeinbits* contains the size in bits of one single packet, we add eight more bytes because it's a UDP packet and these eight bytes correspond to its header. In the first main "if" we check whether the first and the last ports are the same. If they are different, we try to find the index of the first packet with the same port number as the last packet. We add one to $first$ because we start calculating the delay between a packet and its previous one. The second main "if" is used to make sure that if we ended receiving and the ten second condition is not true, then if we have at least 80 packets we then estimate the DBdw. If the ten second condition is true, we then consider less packets to measure the DBdw because for lower DBdw the estimation can be obtained accurately with less packets. If any of the main "if" conditions is true, then we iterate over the arriving times calculating the delays between every two packets from begining to end and we accumulate these delays in the variable *delays*. We check if *delays* is "0" or not inorder to return the estimated DBdw. In *dlyinsecs* we store the accumulated delay in seconds and then we use it to return the estimated DBdw in bits per second (bps).

```
public float calculateBDW(long[] received, int[] ports, int packets) {
  int first = 1;
  int Sizeinbits = (size[packetsize] + 8) * 8;// + 8 bytes UDP
  long delays = 0;

  if (ports[0] != ports[packets - 1])
    for (int p = 0; p < ports[packets]; p++) {
      if (ports[p] == ports[packets - 1]) {
        first = p + 1;
        break;
      }
    }

  if ((packets - first - 1) >= 80 &&
      (received[packets-1]-init) < 10000000000f) {
    for (int i = first; i < packets; i++)
        delays += received[i] - received[i - 1];
  }
  else if((received[packets-1]-init) > 10000000000f){
    for (int i = first; i < packets; i++)
        delays += received[i] - received[i - 1];
  }

  if (delays != 0) {
    Double dlyinsecs =
    Double.parseDouble(String.valueOf(delays)) / 1000000000;
    return (float) ((Sizeinbits * packets) / dlyinsecs);
  } else {
    return 0f;
  }
}
```

Code 5.4: Bandwidth calculation process

5.3 Conducted tests

Mobile networks (MN) are changeable and that makes it difficult to find out whether the results we obtained, were good or not. It turned out that when using MN, if we measured the DBdw in very short intervals of time (five seconds difference), we could see a big difference between some results. To determine how good our results were, we performed some tests comparing the results obtained with our tool, with the results obtained with the commercial tool *Speedtest.net mobile* (ST) mentioned in section 2.2.1.

To perform these tests we ran our tool to obtain packet trains of 100 packets. We ran it three times for every test, each time with a different packet size. The packet sizes used were 100 bytes, 500 bytes and 1400 bytes. We started by running the ST tool, straight after it was finished we ran our tool with the three different packet sizes one after the other following this order:

1. 100 packets of 100 bytes
2. 100 packets of 500 bytes
3. 100 packets of 1400 bytes

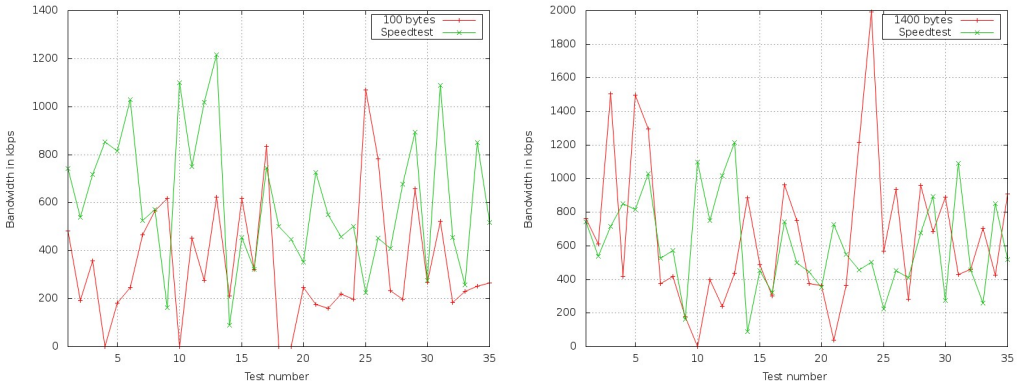
In the following sections we introduce and discuss the results obtained. We compare the results from the different packet sizes with the ones obtained with ST.

5.3.1 Results

We conducted a total of 35 tests. For some tests the result obtained with our tool was "0", which means that the client did not received the expected packet train. The data obtained from the tests is shown in table 5.2 at the end of this chapter. We can see the estimated bandwidth (BDW) for the three variations of our tool and for ST. We can also see the number of received packets (#Packets) for each different request using our tool.

Inorder to see the relationship between the results obtained with ST and our tool, we plot the data for every single variation against the data obtained using ST. Even though we can see that some results are close to the ST results, we can't tell which packet size is better only by looking at these results. To compare the different results obtained with our tool with the ones obtained with ST we conducted a *t-test*, in which we concluded that the results that were closer to the ST results were the results that were obtained with 100 packets of 500 bytes. This is explained in more detail in

section 5.3.2.



(a) 100 bytes (b) 1400 bytes

Figure 5.2: SpeedTest results vs 100 B and 1400 B

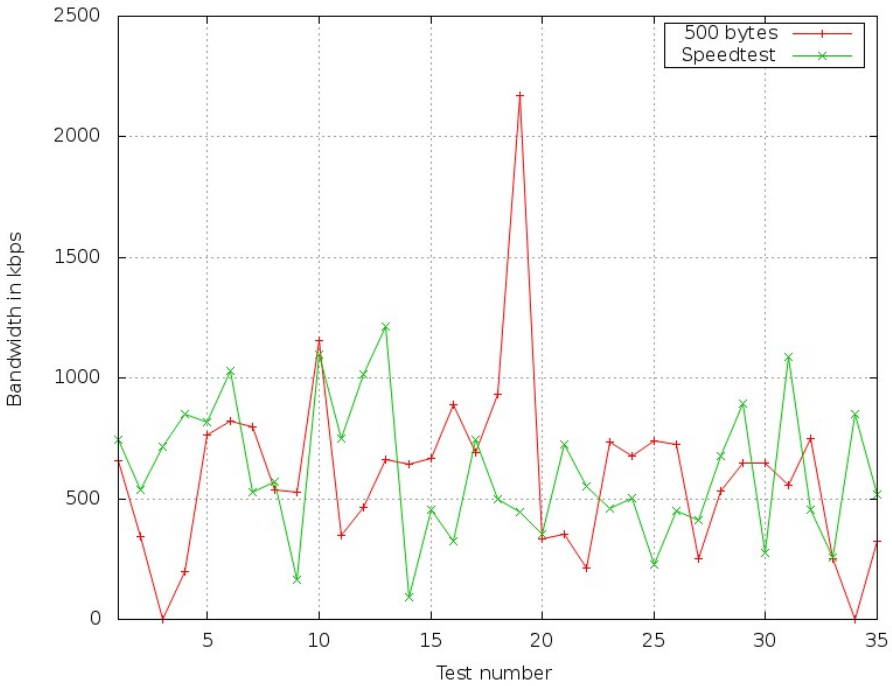


Figure 5.3: 500B packets vs SpeedTest

With 100 packets train of 500 bytes each packet we decided to calculate the bandwidths in ten packets interval, to see if it was better to use 100 packets or we could use less instead. With this calculations we observed that use

less than 100 packets to estimate the DBdw could produce under estimation. When we are doing more than one request (because we are not receiving the paquets), when a train arrives sometimes can be seen that the delays between packets is getting smaller while we receive more packets up to 100 packets. This produces that if we use less packets to estimate the DBdw, the result that we get will be much lower than with 100 packets. The difference of the estimated DBdw with 80 packets is minimum with the one obtained with 100 packets and that's the reason why we use at least 80 packets to estimate the DBdw.

5.3.2 Hypothesis Tests for Compare the Measurements obtained from Measurement Tool and SpeedTest Readings

Sample	P value	t statistic	1- α	deg. of freedom
data100 vs. SpeedTest	1.1788×10^{-4}	-4.0866	1-(0.001)	68
data500 vs. SpeedTest	0.926	-0.0975	1- (0.2)	68
data1400 vs. SpeedTest	0.5539	0.5949	1-(0.59)	68

Table 5.1: Two sided independent sample t-test summary

We have performed hypothesis tests for each data sets against SpeedTest sample for any significant differences between means. All the hypothesis tests were conducted under the $\alpha = 0.05$ (95%Confidence Interval) significance level.

Hypothesis Statement:

Null hypothesis is $\mu_{data100,500,1400}$ are same as $\mu_{Speedtest}$.

Alternative hypothesis is $\mu_{data100,500,1400}$ are not equal to $\mu_{Speedtest}$.

$$H_0 : \mu_{data100,500,1400} = \mu_{Speedtest}$$
$$H_1 : \mu_{data100,500,1400} \neq \mu_{Speedtest}$$

For data100 sample: The P value (1.1788×10^{-4}) is less than 1- α (0.999) and therefore we have to reject the null hypothesis and accept the alternative that the mean of data100 sample and ST are significantly different.

For data500 sample: The P value (0.926) is greater than 1- α (0.8) and

therefore we fail to reject the null hypothesis concluding that there is not enough evidence to suggest that the two means have a significant difference.

For data1400 sample: The P value (0.5949) is greater than 1- α (0.4) and therefore we fail to reject the null hypothesis concluding that there is not enough evidence to suggest that the two means have a significant difference.

Given the test results we accept the null hypothesis of sample data500 and data1400. From the above three hypothesis in combination with the graphs from previous section, it is evident that the 500 bytes test probes give similar results as ST measurements.

Time interval	100 bytes		500 bytes		1400 bytes		Speed Test
	BDW	#Packets	BDW	#Packets	BDW	#Packets	BDW
1	481.877	100	656.323	100	761.003	100	743
2	190.978	100	345.415	100	609.638	33	539
3	358.679	100	0	0	1504.68	100	717
4	0	0	197.441	15	417.881	8	852
5	181.708	100	765.691	100	1498.21	100	816
6	246.257	100	822.014	99	1297.03	100	1030
7	465.569	100	799.779	100	375.899	53	525
8	566.557	100	536.994	99	419.244	8	571
9	616.606	100	526.736	15	176.764	8	162
10	0	0	1154.31	100	0	0	1100
11	452.92	100	347.822	100	397.502	62	750
12	276.157	100	463.037	100	239.177	100	1017
13	621.61	100	661.436	100	436.041	46	1216
14	210.497	100	645.423	100	887.291	100	90
15	618.49	100	667.063	100	486.322	100	454
16	322.77	100	888.602	100	302.105	20	322
17	834.979	100	689.49	100	964.677	80	744
18	0	0	934.515	100	751.772	82	500
19	0	0	2170.64	100	375.807	100	446
20	247.417	100	332.575	100	364.945	100	352
21	176.568	100	351.518	100	39.7518	52	726
22	160.163	100	210.903	100	362.183	100	549
23	218.92	100	736.582	100	1213.85	100	457
24	198.412	100	676.028	100	1991.59	100	501
25	1069.4	100	737.846	100	569.45	100	225
26	783.717	100	725.043	100	938.002	99	451
27	233.822	100	252.556	15	283.899	72	409
28	197.238	100	530.396	100	959.316	86	678
29	659.358	100	648.532	100	685.851	99	894
30	268.567	100	647.933	100	891.144	100	276
31	523.636	100	557.831	100	427.6	100	1089
32	184.62	100	751.786	100	460.03	100	456
33	229.096	100	253.103	100	704.591	100	258
34	250.812	99	0	0	426.074	8	851
35	264.453	99	325.543	15	907.874	100	517

Table 5.2: Data

Chapter 6

Smartphone application

6.1 Introduction

In this chapter we describe the application we developed. The contents of this chapter will go as follows. First, we introduce the different elements we measure with the application and describe the file we upload to the server. Afterwards, we describe the different components and information about the device that we are accessing and their role in the application. Further, we discuss the main issues we had and how we solved them. In the end we explain how the application works and its interaction with the users.

6.2 Information collected by the application

In this section we describe the information we are obtaining from the application and how we upload it into the database server. We first briefly look at the different data we store for each measurement one at a time. Afterwards, we discuss in more detail the structure of the file that contains all the measurements and the uploading process.

6.2.1 Measurements

The application we developed retrieves information related to the device. This information can be divided into four different groups: device, location, network provider and network connection. Now we identify and define them. In later sections we talk about how we obtained them and their role in the application. We must mention that in addition to this data we also save the date, starting time and ending time for each measurement but explanations are left out for simplicity's sake.

1. Device

   Device identifier This number is the IMEI for GSM and the MEID or ESN for CDMA phones and is unique for each device.

   Phone type Name of the radio type the device uses.

2. Location

   Latitude & Longitude These are the coordinates of the device's GPS in the moment a measurement is taken.

   Accuracy Represents the accuracy in meters of the coordinates received from the GPS.

3. Network provider

   Operator identifier Known as a "MCC / MNC tuple", this number is a combination of two numbers, the *Mobile Country Code* (MCC) and the *Mobile Network Code* (MNC). The MCC is the ISO country code of the operator and the MNC makes reference to the operator in that country (This is explained in more detail in section 6.3.3).

4. Network connection

   Local area code A unique number assigned to a "location area", where this "location area" is a set of base stations that are grouped together to optimise signaling.

   Base station identifier Identifier of a base station in a "location area".

   Signal strength This number represents the signal strength and its value is between 0-31 or is equal to 99 as defined in TS 27.007 8.5 (*3GPP Technical Specification*).

   Network type Depending on whether the phone is a GSM or a CDMA phone, this value lets us know what kind of network is in use (GPRS, EDGE, UMTS, HSDPA, ...).

   Bandwidth Represents the download bandwidth when the measure took place and it is obtained with the tool mentioned in chapter 5.

Instead of uploading every single measurement that a user makes, what we do is register each measurement in a Java class we created called $Measurement$. This class contains a field for all information required in a measurement and

it provides setters and getters to access these values. In the main class we have declared an *ArrayList* of *Measurement* objects (*measurementset*), so all the measurements are stored in this array till the file is upload to the server or till the user exits the application, in that case we write the measurements into the file. Using an array makes it very easy to add/delete new measurements and access them. Once the file is uploaded, the array is cleared to avoid uploading the same measurements twice.

6.2.2 Measurements file

All the measurements taken while the application was running are saved in an XML file called "*UNSWBandwidthData.xml*". This file is stored in the external memory of the device, in a folder created by the application. So if the application is removed, then the folder and all its contents are also removed. The structure of the file and how it is treated by the application is explained below.

The file's structure is easy to understand now that we know the values it contains. Here we describe the relationship between the tags and the measurement values. In figure 6.1 we can see an example of a file that contains two single measurements.

List of tags:

<**measurements**> Start tag of the document, its attribute "user_phone_id" is the device identifier of the phone that uploads the file.

<**measure**> First tag for each measurement and has two attributes: "date" that represent the date of the measurement in "dd/mm/yyyy" format and "time", which represents the starting time of the measurement in "HH:mm:ss" format.

<**endtime**> The time when the DBdw estimation finished in the format "HH:mm:ss", which is the ending time of the measurement.

<**BaseStationId**> The value contained in this tag is the base station identifier.

<**LAC**> Local area code of the base station.

<**operatorid**> The MCC+MNC of the network operator.

<**signalstrength**> The value of the signal strength when that measurement was taken.

<**phonetype**> The type of phone, GSM or CDMA.

<**networktype**> The kind of network connection we have (GPRS, EDGE, UMTS, HSDPA,...).

<**lat**> The latitude of the coordinate given by the GPS.

<**long**> The longitude of the coordinate given by the GPS.

<**accuracy**> The accuracy in meters of the geographical coordinate.

<**bdw**> The value obtained in kilobits per second from the Measurement tool (Chapter 5).

When the application is started, it checks if the file exists. If so, that means it contains measurements from previous runnings, so adding new measurements would produce a malformed .xml file. We extract the information about the previous measurements by parsing the file and we store them in the *ArrayList<Measurement>* we mentioned in previous section (6.2.1). With all these measurements stored, the application is ready to start obtaining new data that will be added to the previous array.

```
−<measurements user_phone_id="354957033076459">
  −<measure date="27/06/2011" time="17:01:00">
      <endtime>17:01:13</endtime>
      <BaseStationId>78937180</BaseStationId>
      <LAC>215</LAC>
      <operatorid>50503</operatorid>
      <signalstrength>19</signalstrength>
      <phonetype>GSM</phonetype>
      <networktype>UMTS</networktype>
      <lat>-33.91842097</lat>
      <long>151.23096665</long>
      <accuracy>13.416408</accuracy>
      <bdw>876.5607</bdw>
    </measure>
  −<measure date="27/06/2011" time="17:02:00">
      <endtime>17:02:09</endtime>
      <BaseStationId>78937180</BaseStationId>
      <LAC>215</LAC>
      <operatorid>50503</operatorid>
      <signalstrength>19</signalstrength>
      <phonetype>GSM</phonetype>
      <networktype>UMTS</networktype>
      <lat>-33.91787845</lat>
      <long>151.23106647</long>
      <accuracy>11.313708</accuracy>
      <bdw>854.5374</bdw>
    </measure>
  </measurements>
```

Figure 6.1: UNSWBandwidthData.xml

When we stop the application, we must save all the measurements we took. What we do is iterate over the *ArrayList* and write them in the file as shown in figure 6.1. When we write the file, it does not matter if the file exists or not because we are not appending information to it.

At the time the user decides to upload the file, if the upload is successful we delete it to avoid future conflicts with the data. We explain in section 6.5.3 the uploading process.

6.3 Device components

The application requires access to several pieces of information provided by the device: location information, device details, network provider information and network connection information. In this section, we describe them one by one and we refer to the Java classes used in each case.

6.3.1 Location information

One important thing for us is to know the user's location, so we can see the relationship of the obtained data in a certain location. Hence, in the later phase of the work we will be able to build a map with this data.

When developing a location-aware application for Android, developers can use GPS and Android's Network Location Provider to acquire the user location. Although GPS is more accurate, it only works outdoors and it consumes more battery power than others. We are using GPS because it fits better to our purposes, we also need to be as accurate as possible and GPS gives more accurate results.

The Java class we are using is *LocationManager*, this class provides access to the system location services. These services allow applications to obtain periodic updates of the device's geographical location. This updates are obtained by means of callback. We define a *LocationListener* that we pass to our *LocationManager* and this listener must implement several callback methods. The *LocationManager* calls these methods when the user location changes or when the status of the service changes. We get the updated location through the method *onLocationChanged(Location location)* and we get the new location in the *Location* parameter. The behavior of our application when the location is changed is defined in this method.

In section 6.4.1 we discuss in more detail the updating time for the GPS listener and its relationship with the accuracy of the obtained coordinates.

6.3.2 Device details

A way to identify the user's contribution to our database is establishing a relationship with a user and its phone. The best way to do that and at the same time minimize the user's interaction with the application is by using the device identifier, which is unique for every device. Storing the relation between the user's email and user's device identifier in the database, we can know who uploaded the data to the server and how much this user contributed to our database.

The access to information about the telephony services on the device is provided by the $TelephonyManager$ (TM) class. Once this class is instantiated to the telephony service of the device, we can easily get this identifier by calling the method $getDeviceId()$. This value is part of every measure we upload to the database and it is a primary key of the data in the database.

6.3.3 Network provider information

We can't expect that all users use the same network provider, so due to the fact that there are different network providers, we need to know which one the user is using when doing the measurements about the network. This will allow us in later work to classify the obtained data in different sets and compare the data we collected between the different providers.

To access this information we use the same class we mentioned before, TM. We get this information by calling the method $getNetworkOperator()$ and we get the numeric name of the current registered operator. This number is a combination of two numbers, the *Mobile Country Code* (MCC) and the *Mobile Network Code* (MNC), also known as a "MCC / MNC tuple". The MCC is the ISO country code of the operator and is part of the International Mobile Subscriber Identity (IMSI) number, which uniquely identifies a particular subscriber and is stored on a (usually) removable SIM card. It is always three numbers and in our case it's 505, which is the Australia's MCC. The MNC is always used in combination with a MCC and uniquely identify a mobile phone operator in the country referenced by the MCC. In our tests we were using a Vodafone AU SIM card and the value of the tuple is "50503".

6.3.4 Network connection information

We need to collect information related to the phone's network for our database. This information is obtained from the network features, such as the signal strength, the phone type (GSM or CDMA), the network type (CDMA, UMTS, EDGE, ...) and also information related to the current base station, the base station identifier and local area code (LAC). The LAC is a unique number assigned to a "location area", where this "location area" is a set of base stations that are grouped together to optimize signaling.

To get this information we use the TM class to obtain the first features (signal strength, phone type and network type) and we are using another class called $GsmCellLocation$ for the other features. Phone type and network type can be obtained by doing a simple call to $getPhoneType()$ and $getNetworkType()$, respectively, from our TM object. Signal strength on the other hand, requires overriding a callback function in $PhoneStateListener$. This listener must be registered in our TM object and it is used for monitoring changes in specific telephony states on the device. So, when registered in our TM object, we pass the listener and a flag indicating which state we want to monitor (in this case, LISTEN_SIGNAL_STRENGTHS). Overriding the *onSignalStrengthsChanged(SignalStrength signalStrength)* method lets us control the behavior when the signal strength changes.
To get the information about the base station another callback function needs to be overridden and another listener flag is required (this time it is LISTEN_CELL_LOCATION). With this flag we supervise the changes in the device's cell location and by overriding *onCellLocationChanged(CellLocation location)* we get the related information to the cell location (stored in location). Inorder to obtain the information about our current cell location, we need to cast the $CellLocation$ object into a $GsmCellLocation$ object, so then we can call two functions ($getCid()$,$getLac()$) to obtain that information.

6.4 Issues

While developing the application we found some issues regarding the best way to get our measurements. Here we identify them and explain how we managed to solve them. These issues are related to the GPS accuracy and with the amount of data consumed from the user's mobile plan.

6.4.1 GPS accuracy

The accuracy of the coordinates obtained from the GPS are affected by the amount of time the GPS signal is alive. The GPS needs a start up time but this time is not specified anywhere because it may vary from one GPS to another. So we performed a test to find out how to get a high level of accuracy without affecting the battery consumption greatly.

The test took place at the UNSW and it consisted of running an application to obtain the coordinates give by the GPS and its accuracy. We ran this application four times and each time with a different updating time for the location. Our values where 5, 10, 30 and 60 seconds. The test was run for 14 minutes for 5 and 10 second intervals and for 28 minutes for 30 and 60 second intervals. The length of the data set is different for each one, this is due to the fact that when the updating time is shorter we get more GPS coordinates per minute. The graphs shows the results through the time. First, in figure 6.2 we show the frequency of the results for each updating time. As can be seen, when the GPS updates its location every 5 seconds, more than 95% of the coordinates obtained are at least 20 meters accurate and more than 35% are at least 10 meters accurate. For the other updating times, between 60% and 70% of their results are 20 meters accurate.

If we have a look to figure 6.3a, we can see the variations in the accuracy for 30 and 60 second updating times. Even though we get some accurate results for 60 seconds it is quite irregular and the same happens for 30 seconds. This is caused by the behavior of the GPS, it is designed to save battery, so when the time interval is that long, the GPS gets a location according to its updating time and then its status is set to “STOP”. When the updating time passes, it changes its status to “START” because of these changes, the accuracy of the location obtained is affected and is not always as accurate as it could be with a shorter time. By looking at figure 6.3b, we see that for 10 seconds there are still large variations but for 5 seconds the values obtained appear to be more regular.

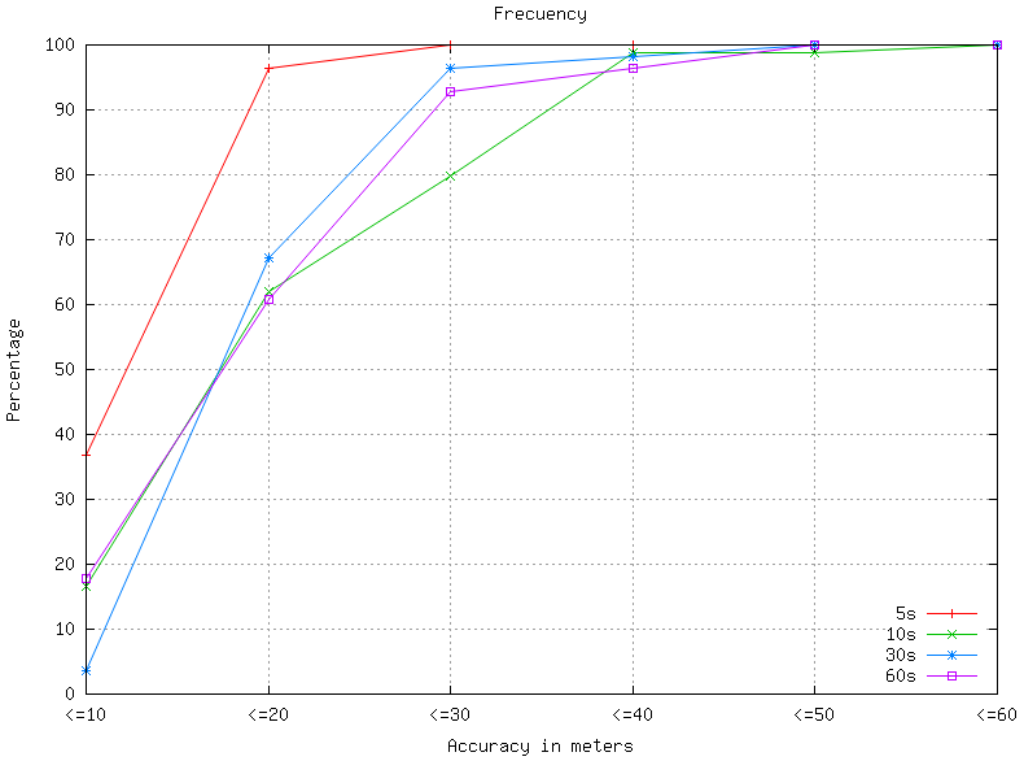
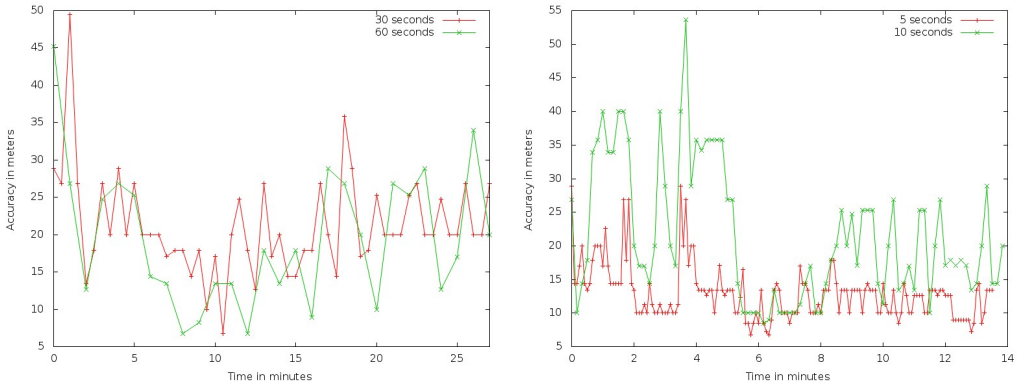


Figure 6.2: Accuracy frequency



(a) 30s vs 60s

(b) 5s vs 10s

Figure 6.3: Accuracy comparison

To show the results in figure 6.4 we used the data obtained during the first 14 minutes for 30 and 60 second intervals inorder to compare them with the results obtained from 5 and 10 second intervals.

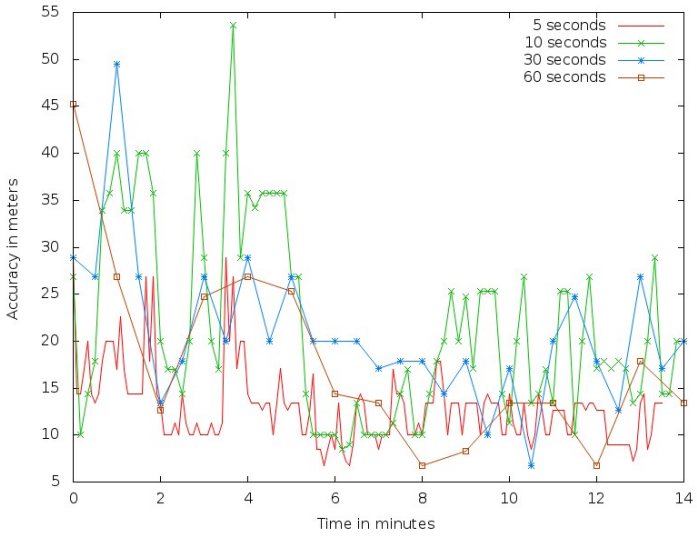


Figure 6.4: Cross data

We conducted another test with a time interval of 500 milliseconds, as we expected the results where more accurate than the rest but this time interval has a much higher consumption of battery. In figure 6.5 we can see the difference with 5 seconds.

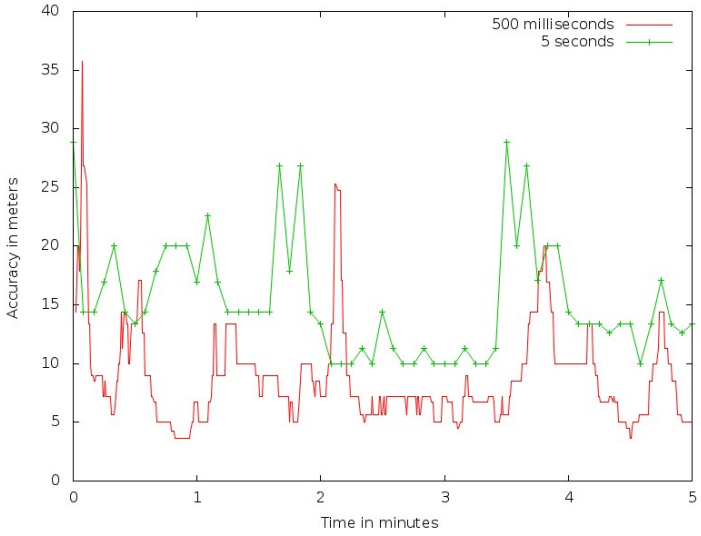


Figure 6.5: 5 seconds vs 500 milliseconds

6.4.2 Data consumption

In order to talk about the data consumption, we first need to know how we do the measurements. We are using an abstract Android class called *AsyncTask*. This class allows to perform background operations and publish results on the user's interface (UI) thread without having to manipulate threads and/or handlers. We created a class called *Bandwidthtask* that extends *AsyncTask*. In this class we override two methods: *doInBackground* and *onPostExecute*. The first method is responsible for executing the code we want and the result from this method is received in the second method. How this task is invoked is explained in section 6.5.2. Now we explain the behavior of this task in figure 6.6.

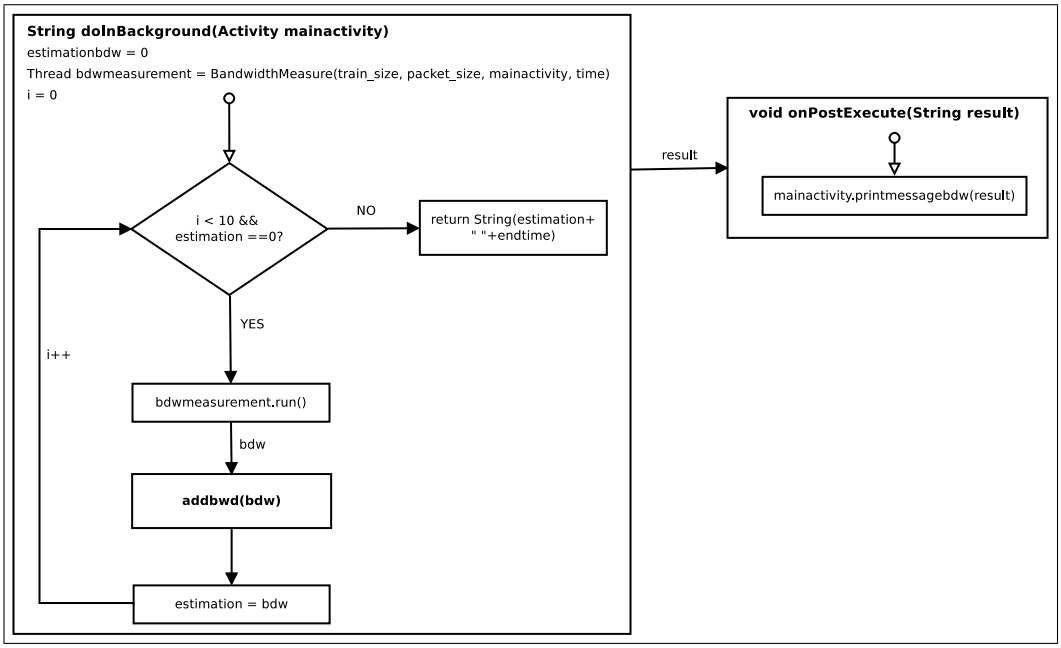


Figure 6.6: Task process

When the task is invoked, in the *doInBackground* function we first initialize the variables *estimationbdw* and i and create a thread with a *BandwidthMeasure* object. In *estimationbdw* we store the DBdw in bits per second (bps) calculated in the thread as mentioned in section 5.2.2. The variable i is used to iterate and run up to ten requests. This is done because while receiving packets, the *BandwidthMeasure* object can timeout before the first packet is received, doing so ensures that we will receive at least one packet train.

Next, we run the thread and wait to receive the DBdw estimation. This value is obtained by the function *addbdw*(*bdw*). From the *BandwidthMeasure* object we call this function in the main *Activity* using the reference passed when created. This function receives the value as the parameter *bdw*. What we do is set the value of *estimationbdw* to this value. While the number of requests is less than ten and the DBdw obtained is "0" we keep requesting. When we finish, we send the value obtained for the DBdw and the ending time to *onPostExecute*. In this function we call the function *printmessagebdw* in the main *Activity*. We split the result in *bdw* and *time*, then we check if the DBdw obtained is different from "0", in this case we store the DBdw (in kbps) and the ending time in the *Measurement* object that was created for this measurement. If it is "0" then we remove this object from *measurementset* (the array that contains the measurements). The process done in *printmessagebdw* can be seen in figure 6.7.

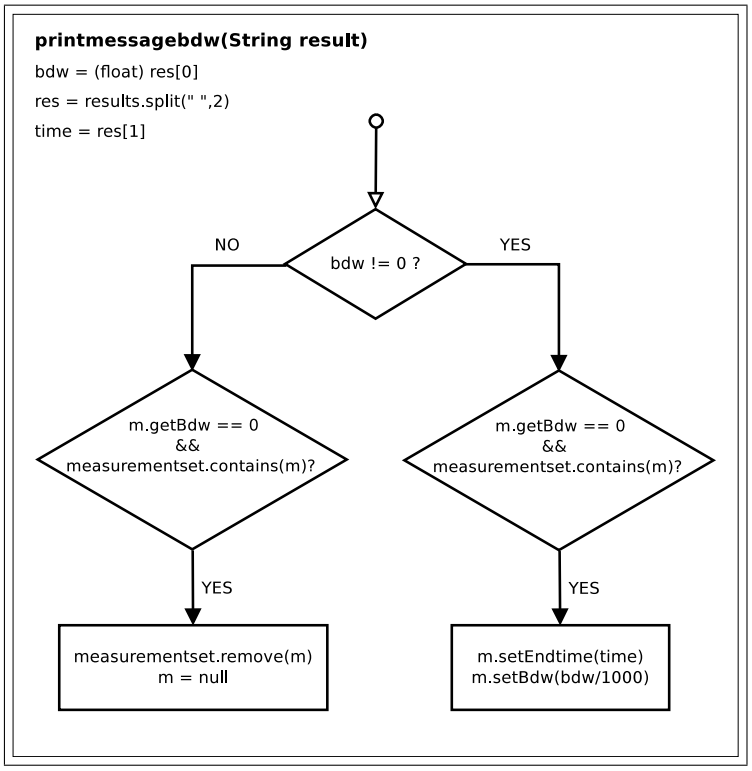


Figure 6.7: *printmessagebdw* function process

So, when estimating the DBdw we are requesting a train of 100 packets, where the size of each packet is 500 bytes. Thus for every full train that

we receive we are consuming 50 kilobytes. Considering that no packets are lost when we receive a request, we must be aware that for every measurement done, 50 KB is the amount of data consumed from the user's data plan.

Consider the following scenario regarding data consumption. This scenario was based on a one hour trip for a user to get from home to university. We consider this as a reasonable scenario since we are planning to distribute the application amoungst students. Based on this, the user would be taking measurements for two hours a day (go and return). Table 6.1 shows the estimated data consumption (Data) and the number of measurements (#M) taken for one day, five days (one school week) and a full month (four weeks) according to different time intervals between measurements. We consider the measurements between work days (Monday to Friday).

Time interval	Day		5 Days		4 Weeks	
	#M	Data	#M	Data	#M	Data
5	1440	72 MB	7200	360 MB	28800	1440 MB
10	720	36 MB	3600	180 MB	14400	720 MB
15	480	24 MB	2400	120 MB	9600	480 MB
20	360	18 MB	1800	90 MB	7200	360 MB
25	288	14.4 MB	1440	72 MB	5760	288 MB
30	240	12 MB	1200	60 MB	4800	240 MB
35	205	10.3 MB	1028	51.4 MB	4114	205.7 MB
40	180	9 MB	900	45 MB	3600	180 MB
45	160	8 MB	800	40 MB	3200	160 MB
50	144	7.2 MB	720	36 MB	2880	144 MB
55	130	6.5 MB	654	32 MB	2618	130.9 MB
60	120	6 MB	600	30 MB	2400	120 MB
120	60	3.MB	300	15MB	1200	60MB

Table 6.1: Relation between data consumption and number of measurements

According to these values, the decision about which time interval to choose should be based on the number of users contributing to the database. How much of their monthly data plan they decide to use for contributing to our database must be considered. This value can be adjusted depending on the user, so users with higher data plans can contribute more than users with lower data plans if they want to. To perform our tests we are using a fifteen second interval because we are using a SIM card with a high data plan.

6.5 Application work

In this section we explain how the application works, we first introduce the interface, thereafter we explain the main functions that integrate the application. We discuss in detail the behavior of the uploading part and then we show a flow chart for the whole process and explain the interaction between the functions. We also explain the behavior of the application but we don't get deep into the code.

6.5.1 Interface

The application's interface is very simple, it only contains three buttons: "Start", "Stop" and "Upload File". First of all we focus on what the purpose of each button is and then we explain the behavior of the application when these buttons are pressed. The interface is shown in figure 6.8, here we see that all buttons are enabled but this was only done for illustration purposes. We will also briefly discuss the interaction between buttons.

Start This button triggers the measurement functions. When pressed, the network connection is checked and if there is a one active connection (Mobile Data or Wifi) then the listener for the location is defined and the device starts monitoring the GPS for location changes. Once the "Start" button is pressed, it becomes disabled until the "Stop" is pressed.

Stop After pressing Start, this button gets enabled. When pressed, we check there is a *Bandwidthtask* object that is running and if it is, we check if the DBdw field is still "0", in that case we remove the object from *measurementset*. If any measurements were obtained before pressing the button then we store them in the file mentioned in section 6.2.2 and we remove the listener for the location. We then enable the "Start" button again and disable this button.

Upload File This button is only enabled when the file that contains the measurements exists and only in the situation that the application is not taking measurements. When pressed the file is uploaded to the database's server through an http POST method. This process runs in an asynchronous task, so in case it takes some time it won't block the interface. When uploading the file some feedback is always given to the user, so the user can see if it is uploaded correctly or there is a problem during the process.

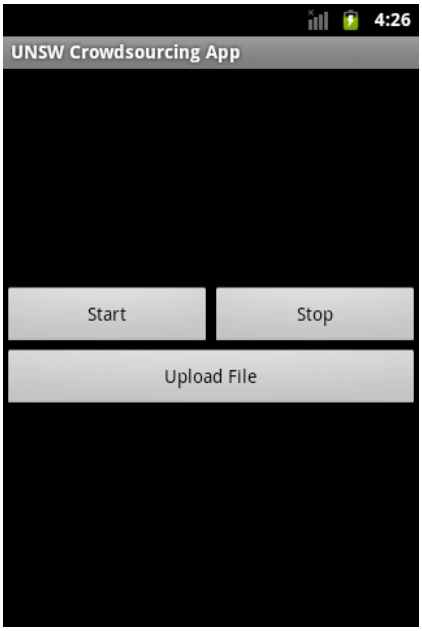


Figure 6.8: UNSW Crowdsourcing Application

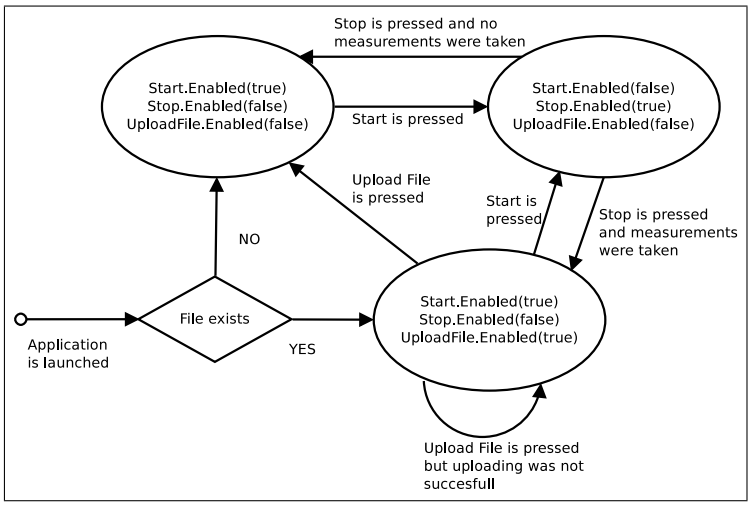


Figure 6.9: Buttons flow

Figure 6.9 shows the flow chart for the buttons between the enabled and disabled states.

6.5.2 Functions

In this section we discuss the behavior and the purpose for some functions. We explain them one by one and we see flow charts to illustrate the behavior of more complex functions.

checkNetwork() We need to make sure that there's a network connection in order to get the measurements. With this function we avoid any kind of networking problems for the application. Trying to send or receive packets to and from the network without an active connection would produce exceptions, so, what we do is check if there's a connection and differentiate between mobile and wifi for our measurements. If there's no connection we show a message to the user to inform him/her.

checkFile() As mentioned before, the button to upload the file will be enabled only in the case there's a file to upload. This function makes sure this happens. This function is used when the application is first launched and after trying to upload the file. Its behavior is very simple, it checks the file's existence inorder to enable the *Upload File* button. In the case where the file does not exist, then it clears the $Measurement$ objects array ($measurementset$) if it's not empty, this will only happen after a successful uploading, if it exists, we call *readXML(File f)*.

readXML(File f) When we first launch the application, a file with previous measurements can exist. As we are using one single file to store all the measurements taken through different application runs, we read the file if it exists (the file given as a parameter f) and parse it. By parsing the file we store all the measurements recorded in previous launches in $measurementset$, so when we start obtaining measurements, the new measurements will be added to the array and when stopped, all measurements will be stored in the file, erasing previous information. This is done because we must preserve the xml structure so we can't simply add new measurements to the file.

writeXml(ArrayList<Measurement> measurements) Using the information related to the measurements taken, which is stored in the array mentioned before, we create the xml file mentioned in section 6.2.2.

get3Ginfo(Measurement measure) This function receives a $Measurement$ object that has all values to their default value. In this function we get all the data using the methods explained in section 6.2.1 to fill all the fields from the $Measurement$ object received but the DBdw and the ending time, which are stored when the $Bandwidthtask$ object finishes

as explained in section 6.4.2. When storing the *Network type*, we first check if the network we are connected is mobile or wifi, if it's wifi then we set its value to "WIFI".

onLocationChanged(Location location) This function was explained in section 6.3.1, but now we'll see how it works.

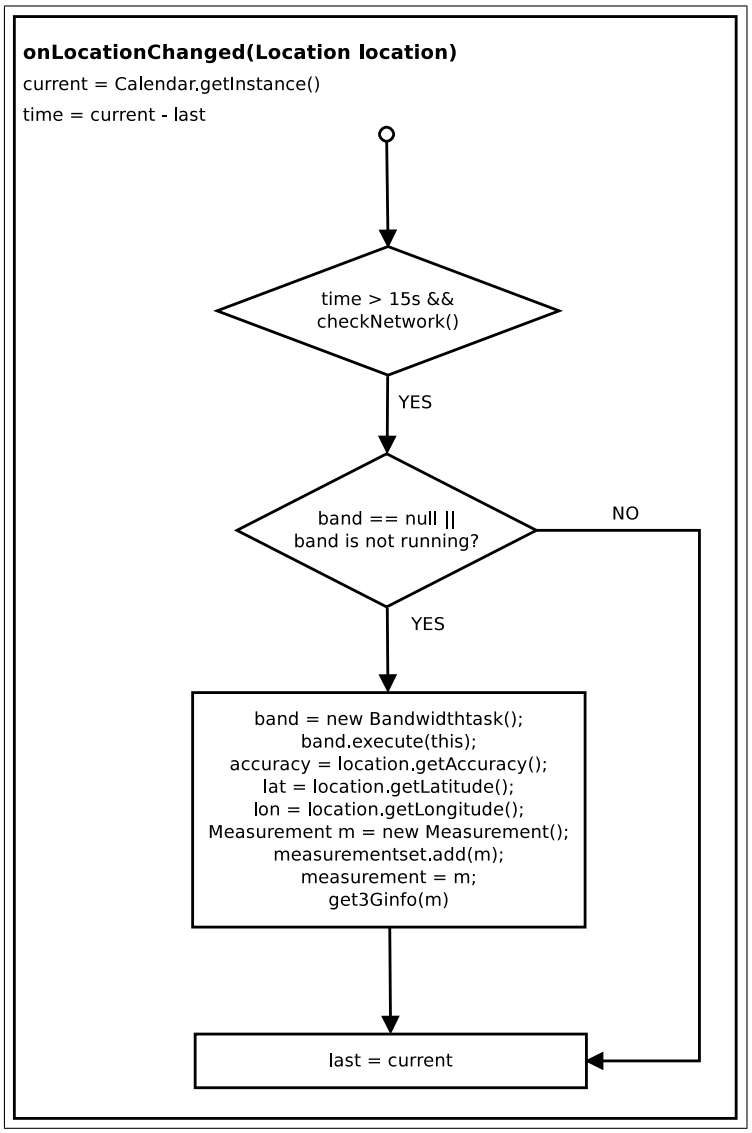


Figure 6.10: onLocationChanged(Location location)

The variables *current* and *last* are objects from the *Calendar* class. This variables store the time, so in *time* we store the difference between the cur-

rent time and the last time we did a measurement. If this time is higher than 15 seconds and the network connection is available, we proceed to do a measurement. Then we check the global *Bandwidthtask* object (*band*), if it is still null (it is the first time we try to run it) or it not running, then we do a measurement. The process to do a measurement can be seen in figure 6.10. First we create a new *Bandwidthtask* object and we execute it. Meanwhile it is running in the background we store the information related to the GPS (accuracy, latitude and longitude) and we create a new $Measurement$ object, which we add to *measurementset* and we store the reference to that object in *measurement*, which is a global variable of the class $Measurement$. Then we pass this $Measurement$ object to the $get3Ginfo(Measurementmeasure)$ function to obtain the rest of the required information.

6.5.3 Uploading

For uploading the file we created a class called $UploadingTask$ that extends $AsyncTask$ (explained in section 6.4.2). In this class we override two methods: $doInBackground$ and $onPostExecute$. Now we explain the code used for uploading the file to the server (Code 6.1).

```
try {
    HttpClient httpClient = new DefaultHttpClient();
    HttpPost request = new HttpPost(
                  new URI("http://129.94.172.130/"));
    MultipartEntity entity = new MultipartEntity();
    entity.addPart("upfile", new FileBody(Measurements_file));
    request.setEntity(entity);

    HttpResponse response = httpClient.execute(request);
    int status = response.getStatusLine().getStatusCode();
    if (status == HttpStatus.SC_OK) {
          Measurements_file.delete();
          return 1;// "Upload Complete";
    } else {
          return 2;// "Couldn't upload, pleas try later...";
    }
} catch (Exception e) {
    return 3;// "There was a problem while trying to upload";
}
```

Code 6.1: doInBackground()

What we do is create an *HttpClient* and prepare the POST request with the

server's IP address. We can't simply attach the file to our request with the standard Java libraries for http connections, it is a little bit tricky. Instead, we are using two third party libraries called "apache-mime4j-0.4.jar" and "httpmime-4.0-beta1.jar" that implemented methods that already embrace all the work. These libraries allow us to use the *MultipartEntity* class and with this class we can attach the file to our request by setting the request's entity to it. After uploading, if the response is "OK" (200), then we delete the file.

```
if (checkNetwork()) {
   if (!uploadtask.getStatus().equals(Status.RUNNING)) {
      if (!uploadtask.isCancelled())
          uploadtask.cancel(true);
      if (Measurements_file.exists()) {
          Toast.makeText(this, "Wait while we upload the file",
                        Toast.LENGTH_LONG).show();
          uploadtask = new UploadingTask();
          uploadtask.execute(this);
      } else
          Toast.makeText(this, "There's no data to Update",
                        Toast.LENGTH_SHORT).show();
    }
} else
    Toast.makeText(this, "You need connection to upload the file",
                   Toast.LENGTH_SHORT).show();
```

Code 6.2: Upload File

Code 6.2 shows the behavior when the "Upload File" button is pressed. When the user press the button we try to upload the file. Before we upload the file we check that all the conditions for uploading are satisfied. These conditions refer to the network connection and to the *UploadingTask* object. We check if the network connection is active and then before launching a new *UploadingTask* object we make sure there's not any other object already running. So, if the file exists, we create a new task and we execute it to upload the file. We use a global object from this class, *uploadtask*, by doing so we can check its status and make sure we don't start two tasks at the same time. If the file does not exists or network connection is not available, we print a message to give some feedback to the user.

6.6 Full process

To explain how the application works we first introduce a transition diagram with the different states of the application. Then we show how the measuring part works with a flow chart and we explain its behavior.

In the transition diagram the flow is shown from the application launch. After we initialize the variables we check if the file $UNSWBandwidthData.xml$ exists. If so, we can upload or start measuring ("Start or Upload" state), if it does not exists then we can only start the measuring part ("Start" state). When we press the "Start" button, it is possible that there's no connection or the GPS is disabled, in that situation we reach a state which we will only leave when both of them are enabled. When everything is enabled and the "Start" button is pressed, then we reach the "Measuring" state where we collect the data. We stop measuring when the user presses the "Stop" button. When we stop measuring, if the *measurementset* array is empty (so there are no measurements to store in the file), we return to the "Start" state, otherwise we go to the "Start or Upload" state. When we are in the "Start or Upload" state and we try to upload the file, if the upload is successful we move to the "Start" state, if the upload is unsuccessful then we stay in this state. This behavior is presented in figure 6.11.

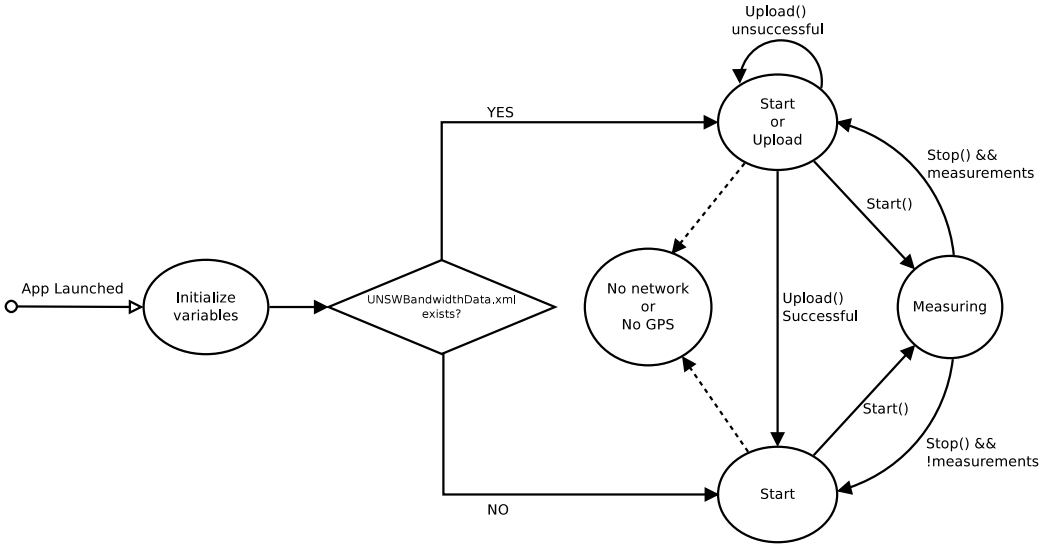


Figure 6.11: States Diagram for the application

The behavior of the "Measuring" state is shown in figure 6.12. When the "Start" button is pressed we check the network connection and if it is active,

then we register the *LocationListener* with a 5 second updating time. If it is not active then we move to the "No network or No GPS" state and afterwards we go back to the calling state. Setting the updating time to 5 seconds means that every 5 seconds the GPS updates its location and this location is received in *onLocationChanged(Location location)*. The behavior of this function is explained in section 6.5.2. When the "Stop" button is pressed, we then check the array where we store all the measurements (*measurementset*), if it is not empty, then we store the measurements in the xml file *UNSWBandwidthData.xml* and we go to "Start or Upload" state. If the array is empty, then we go to the "Start" state. While measuring, if the GPS is disabled, we check if any measurements have been taken, if so we store them in the xml file and then we move to the "No network or No GPS" state. If there are no measurements then we move straight to this state.

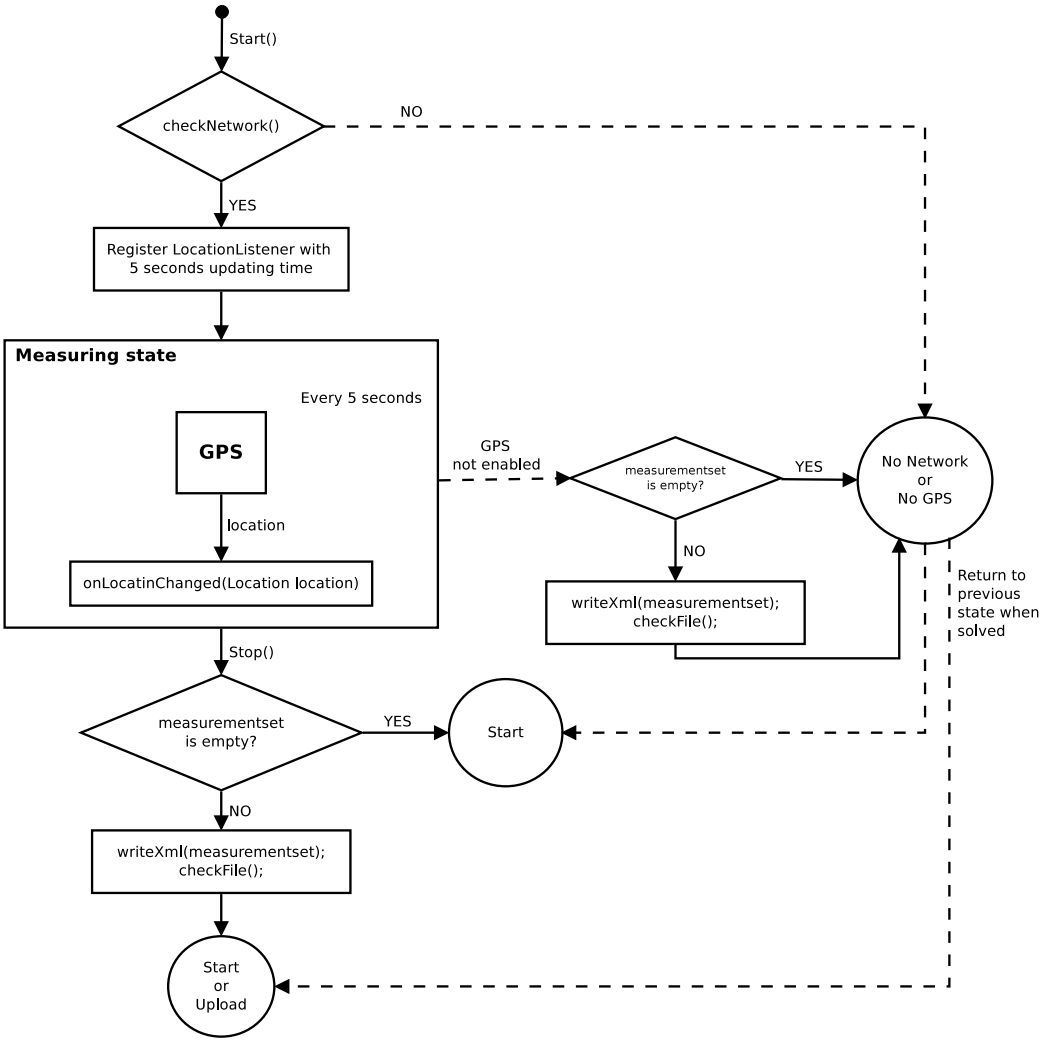


Figure 6.12: Measuring flow

Chapter 7

Conclusions

Finally, we attempt to complete the jigsaw puzzle. We have noted the meteoric rise of mobile smart phones. As a consequence, applications for these smart phones, particularly those related to multimedia streaming are in huge demand. This necessitates the need for better and more consistent data throughput to provide a better user experience. This report is about how we have designed and implemented a mobile application which collects user's perceivable download data bandwidth by acting as a crowd sourcing sensor on behalf of a geo-intellegent system, and also about how developed a web based application that would perfectly be able to use these features for itself.

The application developed in second place measures 3G/HSDPA download data bandwidth actively and automatically thereby reducing user interaction and data consumption. By integrating this functionality in the PideCita application will lead us to more measurement contributors, eventually resulting in a vast measurement database.

Efficient mobile phone power consumption and the user data consumption were our two main considerations. This presented an optimization problem. On the one hand we wanted to maximize GPS accuracy but at the same time, we also wanted to minimize the power consumption. We therefore endeavoured to find an optimal balance between GPS accuracy and power consumption whilst minimizing the consumption from the user's current data plan. Adding the measurement functionality to PideCita can make things easier, as we are using an HTTPS connection for our requests we could use the same request to measure the bandwidth, avoiding like this an extra connection.

Packet Pair probing is the base technique used in our measurement ap-

plication. Packet Pair probing was a convenient choice because of its low computational overhead. Also, it allows for the ability to reconfigure the measurement tool at anytime with very little effort and the fact that it is capable of providing highly accurate results only serves to reinforce this point.

When running our tests for the measurement tool we used measurements obtained from the commercial mobile application *Speedtest.net mobile* as a benchmark. We tested the hypothesis that the results obtained from our application are equal to the results obtained from Speed Test using a two-sided T-test. We conclude that our measurements lie within the 95% confidence interval around the mean of the results obtained using Speed Test, 80% of the time.

In the future we hope to be able to implement the PideCita application on other mobile platforms. This will not only extend our reach in the market, merging both applications will also facilitate access to a wider range of contributors which will serve to enrich the quality and increase the size of the database.

Concluding, adding these new features to the PideCita application would imply that the application would be more prepared to handle connection problems. Users won't suffer the connection problems as many times as they do now, this would benefit this application and also other web based applications. Other future applications developed by NUBESIS could use this information and become more reliable.

Bibliography

[1] Android Developers, *http://developer.android.com/index.html*.

[2] Constantinos Dovrolis, David Moore, and Parameswaran Ramanathan, *What do packet dispersion techniques measure?*

[3] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore, *Packet dispersion techniques and a capacity estimation methodology.*

[4] Rohit Kapoor, Ling jyh Chen, Alok Nandan, Mario Gerla, and M. Y. Sanadidi, *Capprobe: A simple and accurate capacity estimation technique for wired and wireless environments*, UCLA Computer Science Departament, Los Angeles, CA 90095, USA.

[5] Rohit Kapoor, Ling jyh Chen, M. Y. Sanadidi, and Mario Gerla, *Accuracy of link capacity estimates using passive and active approaches with capprobe*, UCLA Computer Science Departament, Los Angeles, CA 90095, USA.

[6] SpeedTest.net, *http://speedtest.net/*.