POLITECNICO DI TORINO

Bachelor Degree Course in Computer Engineering

Bachelor Degree Thesis

Development of a tool to measure smartphone's battery consumption

Supervisors
Assistant Prof. Luca Ardito
Research Assistant. Riccardo Coppola

Candidates
Tristán Armendáriz Irigaray
Student Number: s267000

Academic Year 2018-2019

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, the smartphone has become a fundamental and essential part of the day-to-day people and an almost indispensable tool which we use, on average, about 5 hours per day. One of the biggest problems that users usually find on these devices is the short life of the battery and the high battery consumption of certain applications in our daily use. People depends every day on charging the phone, two or three times per day on some smartphones. This is also the main problem which manufacturers find when they are designing these phones because they always try to combine the highest hardware performance in the shortest possible space. The obstacle is always the size of the smartphones., which has to be limited.

Due to the increasing development of technology in recent years, it has been possible to implement much more powerful electronic components in an ever smaller space. This is why the development of mobile phone batteries has had to deal with a shrinking and limited space.

For all this, it is difficult to increase the capacity of the batteries that are deployed in mobile phones, because the higher the capacity the battery has, the greater the size it is.

It is also important for manufacturers to try to improve the consumption of all hardware components of the device, but it is very complicated due to the technological development of these components, because the

higher the power of a component is available, the higher the energy consumption of the device is. They have more and more power but also have a higher consumption.

As a result, during these last years there has been an attempt to improve the optimization of the software that each manufacturer introduces to the device. Therefore, this project has as its main objective the comparison of consumption that each application has in each different model, to see which software optimizes better the energy consumption.
The thesis "Development of a tool to measure smartphone's battery consumption" aims to provide to the user with a software tool to measure the battery consumption of different applications and services in different models of smartphones and, with it, make it easier for users to decide to choose between different devices.

This is a thesis proposed by Luca Ardito, professor at DAUIN (Department of Control and Computer Engineering) and it is supported by Riccardo Coppola, research assistant PhD student.

This thesis is structured as follows:

1. Background and related work: The first part of this thesis includes some work done previously by professors of the Polytechnic University of Torino and people outside this university who have developed different research on energy consumption. It is about trying to contextualize the development of this thesis and getting some tools already studied to help us achieve our goal.

2. Study design: In this section will be talked about the study that have been carried out on all the elements to develop this thesis. It will be split in two: in the first we will talk about all the tools which have been studied and use, and in the second about the different user usage profiles of the devices. The first part consists about investigating and using the APIs provided by the Android Operating System (or with third party hardware if needed) to achieve exact data about the battery consumption of each application and the hardware components (screen, CPU, network connections ...) and

looking for a tool which automatically executes these scenarios on each device, to get the same execution on each device and, with it, get valid results. In the second part, there will be planned and described different scenarios which contain different daily usages of a smartphone and studying which are the most common apps people use.

3. Procedure: The next step will be, with the tools which have been already studied, to create a tool programmed to automate the connection with the device and the execution of each scenario a certain number of times. Will be collected all the interesting data and saved it on different local databases, from which we will get the data we are interested in to show the user.

4. Results: After saving the results of the different tests in our databases, the next step is to find the best way to show to the user these data, building charts and creating tables and there will be detailed the problems and issues that have been encountered throughout the time of development of this thesis. Finally, there will be an analyse of all the data and try to understand each value to get a conclusion.

Chapter 2

Background and related work

Before starting to study people's smartphone usage profiles, it seems right to give a background for the development of this thesis. We have collected some information about other works and studies about energy consumption of different devices and the measurements that previously have been done. They serve as a base and a previous study about the consumption and optimization of the battery of Android devices. We can divide the structure of the related work into two sections:

In the first part we will talk about the problems of energy consumption, specifically portable devices such as smartphones and computers, and the development in their study in recent years.

In the second section we will talk about the tools and techniques that have been used to measure this consumption in the different devices, and some tools which are used on this thesis to automatize processes.

2.1 Energy issues

As we mentioned in the introduction, the increasing development of portable device technology has achieved high performance of hardware components and this has influenced the high-power consumption of end devices. At the same time, development has also taken place in the fields of energy efficiency of components and in software efficiency. The development and improvement of all hardware components (high CPU speed, more efficient displays, more powerful data storage, Wi-Fi/GPRS/UMTS network adapters, advanced 3D graphics, etc) have clearly affected energy consumption.

In this section, we will talk about different studies that have been carried out on the energy problems of the devices in general and how the different authors report about them.
Along these last years there has been a period in which there is a great development of the mobile technology and the beginning of the migration of different services that before were made with the computer and now it is possible to make them with a mobile device, such as online chats. [1]. This recent increase in electronic mobile devices running 24 hours a day has been the cause of the high increase in global energy consumption and Ardito and Torchiano [3] literally comments that "it is easy to measure dozens of TWH per year attributable to this kind of devices", demonstrating the importance of development in the efficiency of different software techniques.
The increase of devices such as laptops, smartphones, tablets and IOT devices with wireless connectivity [15] has led to an increase in energy consumption [8] and, therefore, an increase in the electricity bill. With this, as an example could be the Bitcoin mining centres [2], which are industrial warehouses with a high number of computers working at 100 capacity, which raises a concern about the amount of energy they consume and how this consumption can be reduced through different techniques.
We can also find references about the increase in the possibility of access to the Internet in underdeveloped countries [17], with an increase of 90% over the last few years and that, by 2030, the development of

the energy efficiency of portable devices can save up to 10 times the carbon emissions they generate.

With all this, most of the studies we collect in this work focus on the study and development of software tools [2] [4] [7] to measure and optimize energy consumption, although we also find some work on measuring certain components of hardware to make comparisons between different device models [6] [16].

Other works such as the one of Rashid, Ardito, and Torchiano [1] go beyond research and investigate the software itself. It talks about all the research on energy consumption generally falls into hardware, or software optimization. It focusses on the study of analysing energy consumption for image encoding and decoding without external devices. It says that the consumption of audio and multimedia video consumes a high amount of resources and therefore energy, and that his study focuses on analysing the energy consumption of the different algorithms of encoding and decoding images and videos, specifically on a ARM-based device: a Raspberry Pi.

These authors also conduct another research [5] about the different sorting algorithms (Bubble, Counting, Merge, Quick...) and explain that, because current mobile devices run programs written in different programming languages, it is important to research and compare the energy consumption of the different programming languages implemented, also conducting an experiment with the Raspberry Pi.

2.2 Tools and techniques for measurement

Having analysed all the problems about the high energy consumption of the devices being developed, we proceed to study different techniques and tools to carry out the measurement of that consumption and its subsequent analysis by different ways. We will also discuss some examples of different tests and some tools which are used for the automation of these tests, comparing them to those that I have used in this project.

In general, almost all the research and studies that have been collected perform physical measurements on the devices that are to be tested, since it is the most direct measure and of which there are the most tools to carry it out. The way to make the measurements is by directly connecting the device with a measuring instrument, to obtain the total instantaneous consumption in mWh and the current discharge current in mAh. Rashid, Ardito and Torchiano [1] acquire energy consumption data through a power meter called USB Tester OLED Backpack 2.0 1. This device gets current and voltage with a sampling frequency of 10 Hz, and it is placed between the power source and the device to test. In a second work by the same authors, [5] the energy consumption data was acquired through a NIUSB-6210 1 DAQ, which is a similar tool as the first one. [6] In the research of Ardito, Procaccianti, Torchiano and Migliore, the way to carry out the measurements is physical, that is, they are taken by Data Acquisition Board (DAQ) and is obtained the actual and pure consumption of the device. In a similar research, Procaccianti, Vetro, Ardito, and Morisio [7] make the measurements physically and by a PloggMeter. This instrument is a power monitoring device that measures the total and pure consumption of the device, without differentiating the consumption of each hardware component or applications or services.

The measures carried out by means of physical instrumentation, such as those mentioned above, give us a general idea about the overall consumption of the device. Some studies go further and attempt to perform measurements using software tools. These measures allow us to isolate the consumption of different applications and components of the device and obtain detailed information of the energy consumption of each service.
This is the case of this thesis, where thanks to the Android Batterystats tool [9], we can obtain detailed information about the specific power consumption of each service and application of the device.

Tawalbeh, Eardley and Lo'ai [16] carry out the measurements by using the PowerTutor and AMobiSense app tools to measure the power consumed by the some major components like CPU, WiFi Interface, GPS

Unit, Video playback, and both operation modes: normal and airplane. Otherwise, Ardito and Torchiano [4] uses the energy consumption model for a Linux Single Board Computer for converting data about resource usage into energy or power consumption information.

We also find differences in the way the authors propose the tests to carry out the measures. Scenarios are raised for certain hardware components of the devices (WiFi, GPS, NFC...) [6] and others describe scenarios applied directly to specific applications (Skype, Gmail...) [7]. Depending on the complexity of the scenarios raised, some works use automation tools for these scenarios. These tools allow you to schedule the automatic execution of tests, to achieve more accurate and valid results. Examples of automation tools used by different researchers are AutoHotKey3 [6], Espresso and Calabash [20], UI Automator [13], SikuliX [11], Repository, Test Driver, Report, Exporter [19] and finally Appium [12].

This last one (Appium) is one of the most used tools for task automation and is the one used for the automation of scenarios in this thesis, due to its simplicity and possibility of working cross-platform, both on computers and different smartphones and tablets. There is a lot of documentation and research with this tool online, like the ones explaining Appium architecture [21] and how does it work, by Micah Williams [22].

Meiliana, Septiana and Setiawan [20] compare different automation tools, such as those we have discussed before and describing some advantages of Appium against its rivals. Ashwaq and Rizwan [19] use Appium as a novel tool (the work takes place in the year in which Appium breaks into the market) and perform different tests with this tool, comparing the final results to others obtained through the other tools of automation in several previous studies.
Finally, by opting for Appium for the automation of the scenarios of this thesis, we rely on research such as the one of Shiwanqi, Rucha and Ayushi [23], which conduct a study on various automation tools and finally highlight Appium as the more complete tool, mainly because of

the possibility to implement it on any platform (Android, IOS, Windows, Linux...), supports various programming languages (Java, C++, Ruby...) and allows applications to be tested without recompiling it.

Chapter 3

Study design

This section will be divided into two parts. In the first, we will discuss the different tools that have been studied and those that have finally been chosen. The second will study the different usage profiles that can be known to a person and describe some scenarios that match these profiles.

3.1 Instruments

3.1.1 Dumpsys with batterystats

These scenarios described in the previous section help us to understand the different applications and services which users manage every day. To achieve our final goal, we should get a final measure of the battery consumption from each device as close as possible to a real consumption in the day to day. Also, it would be interesting to look for a tool which show us information about each service and app consumption, like the screen, CPU, network components... Get this isolated consumption result from each app will be interesting to finally compare the battery consumption of each app by each different smartphone model, and so the user will get more detailed information about the different consumption to be able to choose the smartphone with the lowest consumption when buying.

In this section, we are going to talk about the different tools we have looked for and the ones we are using finally.

During the initial approach of this thesis, we studied the possibility of carrying out the measures physically, like it is done in other works cited in the background section. This means that, during the execution of the tests, the device would be connected to meters of consumption, such as multimeters or other devices like Data Acquisition Board (DAQ) and PloggMeter. With this, we could get a global value about the device consumption and it would be a good data to work with and to show to the user.
Otherwise, the problem of performing measurements in this way was the difficulty and complexity for a user who wanted to use our tool, because it would be essential to have these measuring devices at the time of testing. Because of this, we strived to find a way to create a tool that was more accessible to all users and does not require a lab to carry out the measures.

Watching the development that the Android software has carried out over the last few years, we realized that there was the possibility of obtaining very detailed information about the consumption of energy with own software provided from Android directly.
First, we had to get a tool which would showed us statistics about the battery consumption from the execution of the scenario. Besides, we needed these statistics to show the isolated consumption of each application or service and thus show the user more specific and clearly information.

Android Debug Bridge [1] (from now on we will call it adb) is a tool provided by Android which let us communicate with an Android device via command-line. One of the options adb has is dumpsys with batterystats, which is shown in the next command:

[1]ADB: https://developer.android.com/studio/command-line/adb

adb shell dumpsys batterystats > batterystats.txt

By executing this adb command, we get the batterystats.txt text file from which we can obtain all the information about battery consumption.

This is a small extract of the "batterystats.txt", from which we have collected the most important data and that we are interested in:

-Battery History (1% used, 420 used of 256KB, 7 strings using 294):
-Estimated power use (mAh):
-Capacity: 3220, Computed drain: 29.9, actual drain: 0-32.2
-Screen: 17.3
-Uid u0a108: 5.59 (cpu=5.17 wifi=0.359 sensor=0.0621)
-Uid 2000: 2.49 (cpu=2.23 wifi=0.256)
-Uid 1000: 1.33 (cpu=1.31 sensor=0.0171)

We can see clearly from here the total capacity of the device (3220 mAh), the total consumption of the test (29.9 mAh) and the screen consumption (17.3mAh). Then, we can appreciate some elements with different Uids.
These are the different apps of the device and the next value is the battery consumption of the app.
We can also appreciate, in parentheses, data about CPU consumption, WiFi and sensors during the execution of the application.

So, the steps that are needed to collect these statistics and that we will perform are:

- Reset the battery statistics with the following command:

>adb shell dumpsys batterystats –reset

- Perform the scenario we want to test
- Execute the batterystats command to get "batterystats.txt" file
- Read the file and save in the database all the data we are interested in (later it will be explained how we read and take these values and

save them in databases).

The steps will be performed and repeated as many times as we want.

The main problem with using this tool is the inability to collect battery consumption statistics if the device is charging during the test run. Therefore, we had to find a way to connect the device to the computer without it charging the battery at the same time. This way will be explained later.

3.1.2 Vysor and SikuliX

After knowing that we could get all the battery info with the adb implementation, we started looking for some tools with which, in some way, we could get all the devices to automatically execute the applications of the scenarios we have described. To achieve the greatest similarity in the execution of the tests, we should also look for or build a tool which automates the execution of these scenarios.

The first tools we started studying were about GUI automation. This type of automation works by automating the different testing processes of software with graphical user interfaces and image recognition. Image recognition can be used as a tool for scenario automation. When it is desired to automate a process, a series of actions that need to be performed are detailed, either on a mobile device or a computer. These actions are usually clicks or keystrokes, and reference images of the elements on which you want to perform the action of each application are used to automate these processes. To apply it to this work, we follow the steps below.

First, we must find a way to mirror the device´s screen in the computer you are working with. We found Vysor [10], which is a computer program, working like a Google Chrome extension, with which you can mirror the device´s screen and control it with the computer´s keyboard and mouse.

The first thing is done, but only with this tool we cannot obtain any process automatically. The complementary tool we had to use was SikuliX [11].
SikuliX is a tool which automates anything you can see on your computer´s screen. SikuliX allows us to work with it in different ways. One of them is that it provides a program or graphical interface with which we can easily create scripts. The other is that it implements a library in Java with which by code we can automate our scenarios. Working with Vysor to get the reference images and then with SikuliX to perform actions on these images, we managed to automate different scenarios.

In addition, it must be said that this kind of GUI automation testing tools can be used also to automate scenarios of interaction without testing purposes. In general, people can also use these tools to carry out automating repetitive and frequent operations. An example of this usage which have been investigated for this thesis is the use of these tools by some Instagram "influencers" to carry out time-consuming actions, such as accepting a large number of follow-up requests.

At first, using image recognition was the main option to start with the development of this thesis, because the scenarios we had raised worked on specific applications, which have identical interfaces on all Android devices. After starting working with these tools, we founded that due to the different screen sizes and aspect ratios, Vysor mirrored in a different way all the devices. As a result, image recognition did not work for the different sizes of the reference recognition images which were used. In addition, we found another problem related with this recognition. There are a large number of Android devices in the world and there are terminals between which there are several years difference. This phenomenon is called "Device Fragmentation". Application developers must be able to cover the largest number of devices, and this is why there are differences in the distribution of the elements of the same application on two different devices. Therefore, we should create different scripts for each device model and it would be impossible to implement.

3.1.3 Appium with Java

After these problems with the tools we have described in the previous section, we had to look for other ways which would solve those problems. After a research, we found Appium [12].

Appium is an open-source tool designed specifically for the automation of processes on different applications, specifically on mobile devices and web apps. It has a client/server architecture which works as follows:
The server creates a REST API interface which receives a connection from a client. Client sends server a series of operations and commands (adb commands) and the server executes this order on the mobile device we are going to test. Finally, it responds with the results of these commands to the client, with an HTTP response showing all the results about the execution of the orders. We explain each part of the architecture:

-Appium Server.

Appium provides us, through the full package we can download from the website, with a program that creates a REST API interface in a determined IP address and a port. There are two different IP addresses which can be used. One is the local host 127.0.0.1:4723 (IP:Port) which is used when the goal is to not test a real device but using an emulator. The other one, and the one we are using, is the 0.0.0.0:4723, used when the objective is to test a real device.
This Appium server must be initialized always before starting running the test, to don´t get any exception about the problems with the connection with the server.
So, in short, we have now a listening interface in 0.0.0.0:4723.

This server is written in Node.js [2]. This is basically an open-source platform that uses JavaScript code scripts on the server-side. The server also interacts with UIAutomator [13], which is a framework provided

[2]Node.js: https://nodejs.org/en/

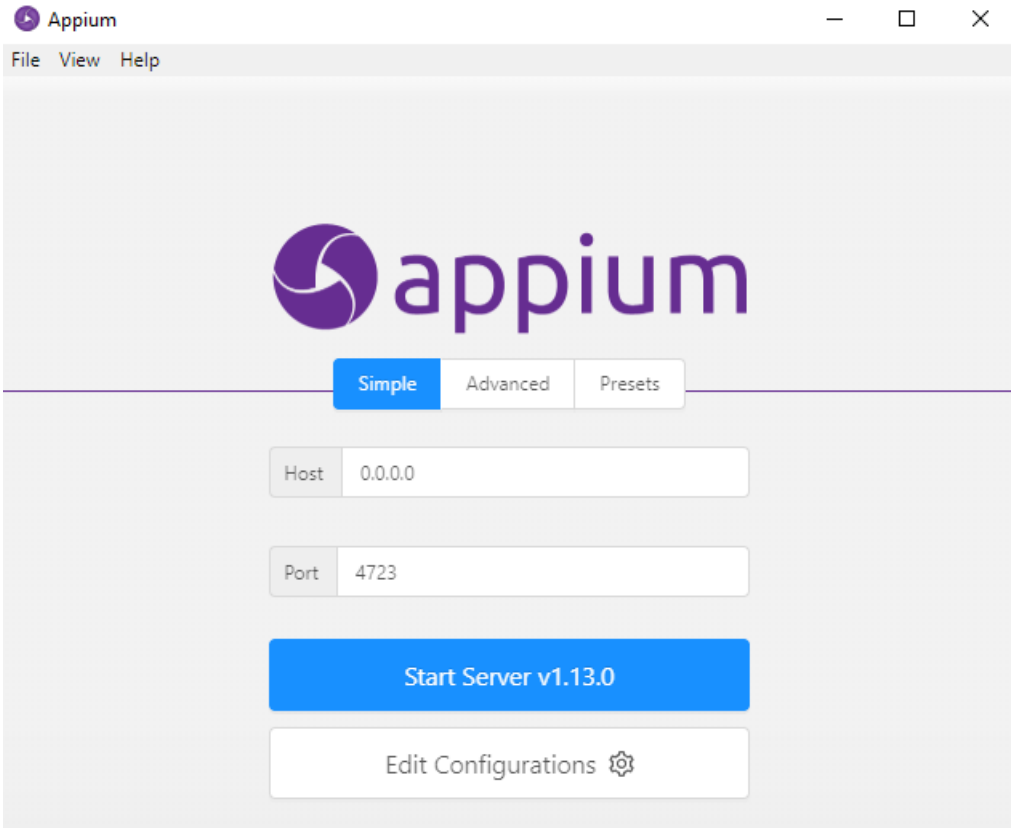

Figure 3.1. Initial screen view of the program "Appium Server"

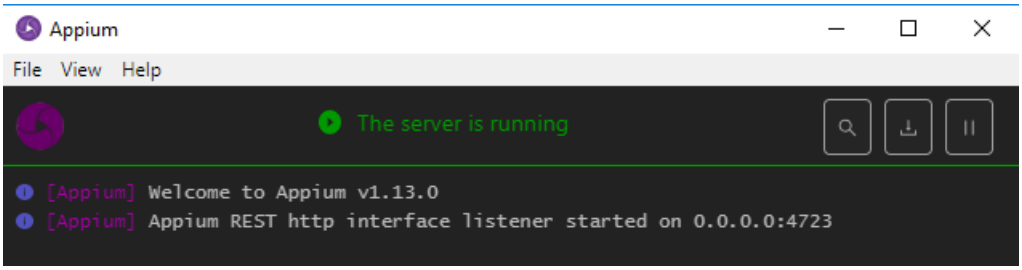

Figure 3.2. View of the "Appium Server" program running the server

in the Android SDK tools created to recognise each element from an Android´s app activity.

An activity is every different page of an Android app, like the login, the main page or a search page, so we can work with all the different

elements of an app. It´s called Page Object Model (POM): for each activity from the application we want to test, there are classes and references about objects in the page (buttons, edit-texts, layouts, labels, checkboxes, forms, images...), so it´s simple for the client to get a path to these objects.

-Appium Client.

This client-side is the part we must program. Appium allows working with client libraries in Java, Python, Ruby, JavaScript, PHP and C. These libraries implement Selenium libraries, which were created for exchanging commands with the server using the JSON protocol. Appium libraries, in addition to Selenium, add specific classes and methods for device automation, such as clicks, double-clicks, sending keyboard´s keys, going back between different activities or sending submits.

In this project we will use Java libraries [3] due to the student's knowledge about this language, the great support and documentation that it has on the internet community and the compatibility with the development of Android, where most of the applications currently run on Java. We will use the Eclipse [4] development environment, also due to the student's familiarity with this program.

To get the complete configuration of Appium and have the work environment ready to perform the automation, we follow these steps:

1. **DesiredCapabilities class:**

   For proper communication between the server and the device, it is necessary for the server to know all the technical characteristics of the device (device's name, platform we are working with, Android version, app´s package name,activity name,etc). It is also important to communicate between them some details about the session.

[3]Java libraries: https://docs.oracle.com/javase/7/docs/api/

[4]Eclipse: https://www.eclipse.org/

Appium calls all these features and details "Desired Capabilities".

Appium implements the Java class Desired Capabilities on its libraries and we must include this information on each session we want to create for each scenario. We will use the method setCapability to achieve this. All these features and session information will be sent to the server when we open the connection. The most important capabilities which must be sent are:

- **Device name and Android version:** there must be included the real name (deviceName) of the device and the current Android version (platformVersion) that the device is running.
- **Package name and main Activity name:** each application has a unique package (appPackage) with a specific name to differentiate applications. We must also indicate the name of the main page (appActivity) that we want to launch at the beginning of the execution.
- **Make the keyboard never appear:** including this capabilitie (unicodeKeyboard), it is ordered that the keyboard does not appear during the execution of the scenario so as not to interrupt its correct operation.

An example of these can be seen in the next figure 3.3

```
//Set the Desired Capabilities
DesiredCapabilities caps = new DesiredCapabilities();
caps = new DesiredCapabilities();
caps.setCapability("deviceName", deviceName);
caps.setCapability("platformName", "Android");
caps.setCapability("platformVersion", androidVersion);
caps.setCapability("appPackage", "com.spotify.music");
caps.setCapability("appActivity", "com.spotify.music.MainActivity");
caps.setCapability("noReset", "true");
caps.setCapability("newCommandTimeout","1000");
caps.setCapability("unicodeKeyboard", true);      //make the keyboard never appear
caps.setCapability("resetKeyboard", true);
```

Figure 3.3. Assigning capabilities in our Java project

2. **Create a connection between Appium Server and the client:**

   As we explained earlier, there must be a connection between the client and the server, which must be initiated by the client. The client´s request is a POST /session request to the server. It includes a JSON object with all the Desired Capabilities that we have explained in the previous step. In Java, Appium library contains the "AppiumDriver" class built for creating a session, in this case with a mobile device. With this, we only have to instantiate an object with two arguments: the URL of the IP:Port from the server and the capabilities of the device, as we can see in the figure 3.4.

```
//Instantiate Appium Driver
 try {
   AppiumDriver<MobileElement> driver;
   driver = new AndroidDriver<MobileElement>(new URL("http://0.0.0.0:4723/wd/hub"), caps);
   driver.context("NATIVE_APP");
```

Figure 3.4. Connecting to the Appium server

   We already have all the initial configuration done and we have created the connection between the client and the server. Now we have to achieve a proper communication between the server and the device.

3. **ADB configuration (via WI-FI):**

   The device of our test must be connected by adb to get the server to run the adb commands. There are two ways for making this connection: via USB cable or WiFi.

   In chapter 3.1.1 we talked about the impossibility of collecting statistics on battery consumption if the device was charging while it was connected to the computer. For this reason, we had to find

a way to connect the smartphone and at the same time not charge the device. One way was by connecting via USB, but we needed the phone to be root to force it not to charge while testing.
Therefore, we were forced to make the adb connection via WiFi, even if this connection is more unstable than the wired one.
To connect the device via adb WiFi we should follow the next steps:

(a) Connect the device via USB cable and allow USB debugging

(b) Take the IP address of the WiFi connection from the device. For this, we get the IP of the "wlan0" interface of the computer we are using for the test. We get the IP by executing the following command:

>adb shell ifconfig wlan0

(c) Then, we create an adb listening TCP interface in the port 5555

>adb tcpip 5555

(d) Finally, we disconnect the USB cable and execute the next command:

>adb connect "DeviceIP":5555

At this point, we already have all the configuration done and we have created the connection between the client, the server and the device. The next step would be starting to create the orders and commands that we want to send to the server to run on the device we are testing.

4. **Scenarios class:**

From this point we will be able to start writing each of the 9 scenarios we have raised at the beginning of this work. Each scenario will be written in a different class, to facilitate error handling and because we have to create a different connection for each application that we want to test, since it changes the information of the capabilities ("Package name" and "Activity name") which we must

send to the server.

In the next section, the methods which will be used for the creation of orders in the devices will be detailed.

-Java methods for creating orders on the device.

The methods which Java implements by Appium libraries and which we will use in our script to localize the desired elements are the following three:

(a) **findElementById(id):**

With this method we can find the desired element by introducing the id. This id is a unique identifier which identifies each element of an Android activity, so only by introducing this value we can select the element. This is the easiest method to find an element, because the id is unique and therefore, we will not have any problems.

(b) **findElementByClassName(ClassName):**

With this second method we can find the element by the class name. Classes on Android identify each type of item you can find in an activity. These types of elements can be buttons, editable text fields, images, different layers... An example of an Android class could be the next one: "android.widget.ImageView". It means that the selected element belongs to an image file. This is not very used because it´s not unique, and there can be other elements with the same class name and can be difficult to identify the one required.

(c) **findElementByXPath(Path):**

This last method is used when it is impossible for the user to use the others, because the developers of the applications do not provide with an id to all the elements and therefore, we have to indicate the hierarchical path where the element is located. This route must be obtained manually and because of this it

is the most hard and heavy way to locate an item. We must indicate how the marked element is reached, so we must follow the hierarchical order and also taking into account the number of the index, to differentiate elements from the same class.

For example, the current element could be located by this path: //android.widget.FrameLayout//android.widget.LinearLayout//android.widget.RelativeLayout//android.widget.ImageView[@index='0]'

After locating the desired item on the page, there are other methods to perform certain actions on these items. We will use two:

- **click():**

  With this method we can act on the elements by clicking. It's the main method we'll use.

- **sendKeys(String):**

  With this method we can act on the elements by sending text strings. We will use it when our scenario requires the input of text by keyboard, such as searching for a web page in the browser.

With these two methods, the client can start ordering instructions to execute them on the device.

5. **Preconditions:**

Finally, we want to ensure that the measures we obtain are the result of running the scenarios with identical conditions on all devices. To do this, we must meet certain preconditions in all devices:

• Disable notifications: we must turn off the notifications to do not interrupt the execution of the scenarios. If, for example, there is a pop-up notification during the scenario, Appium cannot recognise an element from an activity. We will use the following command:

>adb shell settings put global heads up notifications enabled 0

• Closing app´s background processes

>adb shell am kill-all

• Highest screen brightness

>adb shell settings put system screen brightness 255

• Make the keyboard to not appear. We do it on each scenario class:

>caps.setCapability("unicodeKeyboard", true);

• Kill adb proceses and restart adb server

>taskkill /F /IM adb.exe /T

>adb kill-server

3.2 Usage profiles and scenarios description

After studying and commenting some fundamental tools for the realization of this thesis, we believe that the next step is try to find certain applications that users use on a daily basis, in order to create specific scenarios focused on the use of those apps.

To achieve the main goal of this project (compare the energy consumption of different apps between different models of smartphones), we must define one by one these scenarios or usages profiles and that is, describe and detail those daily activities (apps and services) which a common user performs in the day to day.

In this part of the project we will explain why we have chosen each scenario and the elements of the application that we will explore. In the chapter 6 of this thesis, it will be described in detail each scenario with all its features, conditions and steps that can be performed on each application.

1. **Standby Consumption**

   A first scenario that should be posed is measuring the battery life of each device simply with the screen on to see which model manages to hold more time on. This might be interesting to check which smartphone model optimizes better its own software, and to see the management that it makes to applications and services in the background.

   This first scenario would give us a global view of the battery life, but for the end user it is more important to know what the battery consumption is after a daily use of different applications and daily services.

2. **Multimedia music consumption: Spotify**

   One of the most important part we must consider and that has been fundamental for the continuous development and improvement of the smartphones is the multimedia section. The recent increase in the use of multimedia services has directly influenced the manufacture of devices, increasing year after year the size and quality of the screens and improving their performance. The problem with this development is the high battery consumption of these applications, due to the large amount of resources they use.
   Therefore, we can raise 4 scenarios to measure battery consumption in relation to the most used multimedia applications for each case.

   The first multimedia scenario is about testing Spotify. Spotify has become the most used music consumption app in the world today, due to its quality, variety (you can find any desired song) and simplicity to use the app. Therefore, we chose this app to evaluate battery consumption due to streaming music playback. We will search for specific playlist and we´ll reproduce a determine number of songs.

3. **Multimedia video consumption: Vimeo**

   Video consumption is the main cause of the increase in internet traffic over the past decade [24] and IP video traffic will be 82 percent of all IP traffic (both business and consumer) by 2022.

   This has been due to the continuous technological improvement of portable devices. Smartphones have ever larger screens and thus increases the amount of video we consume daily. That is why we will describe two scenarios focused on this consumption.

   The first one is Vimeo. Vimeo is a social network created for the

publication of high-quality videos by content creators. It is a service like Youtube, but we will use Vimeo because of the difficulty of controlling the version of the Youtube application, as we will explain in 5.2 section. Vimeo allows us to search videos by different categories and therefore choose the one we are interested in. On this scenario we will search for some videos and reproduce them all.

4. **Multimedia video consumption: Netflix**

The other app also focused on video consumption that we will use is Netflix. Netflix has become one of the world's leading platforms for consuming TV series, movies and documentaries. One of the greatest qualities of Netflix focuses on the possibility of using the application on multi-platform devices (PC, Smart TV, PlayStation, Apple TV...) and that is why we will study in this case the consumption of this application on the different models of smartphones that we will test. To achieve this goal, we will search for a specific series and we will watch a short time of a chapter.

5. **Multimedia consumption: Camera**

The implementation of the camera on mobile devices was one of the greatest advances of the mobile technology in 2002. At that time, it was almost impossible to think about being able of taking photos and to share them at the same time with other people.
Today, due to the normalization of the use of social networks and sharing photos and videos constantly,it is the hardware component that most differentiate the models of smartphones.
For this reason, we propose two scenarios that use this component to measure the impact it has on energy consumption.

The first scenario we propose is the simple use of the device's own camera. The main problem we encounter is that each manufacturer implements different software on the device, so the camera interface

varies for each smartphone model. For this, we will use a common app for all android devices, which is Open Camera.
Open Camera is a free application which allows us to use the smartphone's camera and that we will test using both cameras (front and back) for taking photos and videos, also activating the flash to measure energy consumption.

6. **Social network: Instagram**

The other app that also uses the device's camera, apart from multiple other functions, is Instagram. Instagram has become today's most widely used social network. Initially, it was used only to share photos through posts on a timely and almost weekly basis, but due to the implementation of the "Instagram stories", it has become a social network in which we continuously share daily photos and videos. It is among the 5 most downloaded apps in history and we can also ensure that it is the most used application by young people, thus forcing us to create this scenario for their test and study. We´ll perform a test posting pictures, sharing stories and doing normal navigation through the app, like posting comments and clicking on liking.

7. **Use of e-mail: K9 Mail**

The high development of smartphone technology has allowed us to concentrate most of the work, which we previously did with the computer, on the mobile phone. It is more convenient for us to perform simple tasks, such as exchanging emails, browsing the internet or exchanging files, using the mobile device and not relying on the availability of a computer. That's why we describe three scenarios focused on these uses that we make today from our mobile phone.

Being able to view email on mobile makes the job easier and saves us time when sending and receiving emails. For this reason, we

want to study the impact that the use of e-mail on mobile has on energy consumption. We will use the K9 Mail application, which allows us to configure any email account that allows POP or IMAP protocols. In this scenario, we will send simple text emails and emails with attachments, because email it´s commonly used as a tool for exchanging short files.

8. **Web Navigation: Opera Browser**

   As mentioned earlier, there has been the migration of the tasks that we previously performed on the computer to the mobile device. One of these tasks is about web browsing. Now we can browse and search for contents in a way almost similar to that allowed by the computer and therefore we will study the energy consumption that this implies in the device.

   Opera Browser has been chosen because of its fluidity and convenience when it comes to any search or activity. In this test we will navigate through two pages, viewing and interacting with the different contents.

9. **Data transfer: Dropbox**

   We also use mobile devices to exchange files with others, save items to the cloud, and download files from servers. This data traffic generates a certain consumption in the wireless components of the mobile device and our goal is to measure and compare the impact it has on energy consumption. For this reason, we propose a scenario involving a service application to save files to the cloud.

   Dropbox has been chosen for being a world-renowned application for these services and because of the option of being able to use a free account with a space of 2GB, enough for the test of our scenario. In this scenario we will run an upload of a 500MB file.

With all these scenarios described in this section, it is enough to give us an idea of the different applications that are used daily, and we will be able to carry out the execution of tests on these applications to check the impact they have on consumption battery on each device.

Chapter 4

Procedure

Now that we have seen the study design of the thesis with all the Appium settings configured in Java , we can start putting everything together in a script, which will finally be the tool that will be used for the tests.

We will create a Java project called "JAVA_APPIUM", which will contain all the classes and libraries of our project. The main class will be named "Final_Script.class", in which we will implement the following contents:

1. **ADB CONNECTION**

   The first thing we have to do is to force adb to start connecting the device via USB cable. Next, we kill any process that may exist from "adb" and reset the adb process. For all of this, we need to run the following three commands:

   >adb usb
   >taskkill /F /IM adb /T
   >adb kill-server

   For executing shell commands with Java, we use the Runtime class and the method exec("argument") and we add as an argument an

array of Strings where in each position must go every word of the command that we want to execute. We can see it in the following two figures 4.1 and 4.1:

```
String [] c00 = {"adb","kill-server"};
execCMD(c00);
Thread.sleep(1000);
```

Figure 4.1. Example of description of commands

```
public static void execCMD(String [] cmd){

    try {
      Runtime.getRuntime().exec(cmd);

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Figure 4.2. Example of execution of adb commands

This is the method which will be used to execute all the adb commands of the script when we do not need to get any answer to save.

2. **GETTING DEVICE´S CAPABILITIES**

After having the device connected via USB, we should obtain the capabilities of the Android terminal, to achieve the correct communication with the server as we discussed in the subsection about the Desired Capabilities class. For this, we also have to execute the following commands:

>adb shell getprop ro.build.version.release
>adb shell getprop ro.product.name
>adb shell getprop ro.boot.serialno
>adb shell getprop ro.product.manufacturer
>adb shell getprop ro.product.model

With this we get, in order: Android version, device name, serial identifier, brand and model of the device.

Here we need to pick up the response after the command execution, so we create a specific method to get this response and save it to our variable. We can have a look on it in the figure 4.3. The main difference with the figure 4.2 method is that here, after the execution of the adb command, we use the BufferedReader class to read the line of the Runtime response and return it to save in a String object.

```
public static String execCMDwithanswer(String [] cmd){
    Process p;
    try {
      p = Runtime.getRuntime().exec(cmd);
      p.waitFor();
      BufferedReader reader = new BufferedReader(new InputStreamReader(p.getInputStream()));
      String line = "";
      while ((line = reader.readLine())!= null) {
        return line;

      }
      reader.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
    return "null";
  }
```

Figure 4.3. Example of the method created for executing commands and reading the response

3. FORCE DEVICE TO CONNECT VIA ADB WIFI

We force the device to connect via adb WI-FI, executing all the commands we have described in the 4th step about Appium configuration.

4. **OBTAINING APP´S UIDs**
   We obtain all the app's uid. This is a unique identifier that each device assigns to an application, service, or process in the terminal itself. Therefore, each application will have a different identifier than another device. For this, we have to execute the following commands:

   >adb shell dumpsys package com.spotify.music | grep userId
   >adb shell dumpsys package com.vimeo.android.videoapp | grep userId
   >adb shell dumpsys package com.netflix.mediaclient | grep userId
   >adb shell dumpsys package net.sourceforge.opencamera | grep userId
   >adb shell dumpsys package com.instagram.android | grep userId
   >adb shell dumpsys package com.fsck.k9 | grep userId
   >adb shell dumpsys package com.opera.browser | grep userId
   >adb shell dumpsys package com.dropbox.android | grep userId

5. **PRECONDITIONS COMMANDS**

   We disable the notifications to not interrupt the script execution, we close all the background app´s processes and we put the screen brightness on the maximun:

   >adb shell settings put global heads_up_notifications_enabled 0
   >adb shell am kill-all
   >adb shell settings put system screen_brightness 255

6. RESTARTING BATTERY STATISTICS

Before starting to execute all the different scenarios, we restart the battery statistics information. This is an important step to avoid that the battery consumption information your phone has previously saved does not overlap.

>adb shell dumpsys batterystats –reset

7. EXECUTION OF THE SCENARIOS

After the reset of the battery information, we execute the different scenarios. Each scenario will be executed 30 times. To do this, we create the following "for" loop in which we introduce the following steps, seen in the figure 4.5

```
for(int i=0;i<30;i++){
    Scenario_1_Spotify.main(androidVersion, deviceName);
    getBatteryStats("C:\\JAVA_APPIUM2\\Script_Final\\BatteryStats\\ejecutable.cmd");
    Thread.sleep(4000);
    BatteryStats_EachScenario.main(deviceBrand,deviceModel,uidSpotify,"Spotify",i+1);
    batterystatsReset();

}
```

Figure 4.4. Example of the loop created for scenario execution and saving statistics

- **Running the scenario**
  The first thing we do after resetting the battery statististics is calling each scenario class. Each scenario is made in a different class. Each class contains two methods. In the main method we first create the capabilities of the device, which are required by the Appium served to create the connection with the device, as we discussed in section 3.1.3 and seen in the figure 3.3. Then

we execute the second method, the test method. This method contains, the connection to the Appium server, also seen in the figure 3.4.

As we explained on section 3.1.3, Appium library for Java implements, by the "Mobile Element" class, a series of methods to locate all the elements of any activity from an Android application. In our scenarios, we will localize the different elements of an Android activity by Id, ClassName and by Path.

Therefore, we must find a tool for getting the “Id”, “ClassName” or “Path” from each different elements. To achieve this, we will use the UIAutomatorViewer tool.

This is a software provided in the Android SDK tools [1] which let us to take screen captures from all the apps we want to test. After taking the screenshot of the Android activity, it shows us that image and allows us to select each item we want.This tool provides us with a simple interface consisting of three parts.

The first figure shows us the image of the screenshot where we can select the item from which we want to obtain the information (figure 4.5). From this screen, we can select all the items which are available to perform actions. For example, in the figure, we can appreciate that we can perform actions on the URL by typing text, or refreshing the web, going back and home, clicking on the links, etc.

The second part shows the hierarchy of all the elements of the current activity. Each Android activity is divided into main elements and sub-items (figure 4.6). As we can see on this figure, there are main elements which are FrameLayout, RelativeLayout or LinearLayout and which in turn contain different sub-elements like ImageView or EditText, which are the ones

[1] Android SDK tools: https://developer.android.com/studio/releases/sdk-tools

with which we want to perform actions.

The third part shows us the information needed to identify that element within the hierarchy (figure 4.7). These are identifiers with which we can differentiate all the elements within the same activity, like the resource-id, class, package or index. The identifier which will be used the most is the id because it´s unique for each element. It can be also seen on this figure if the current item can be clickable or not.

With this, we have the necessary information to tell the server how to identify each item.

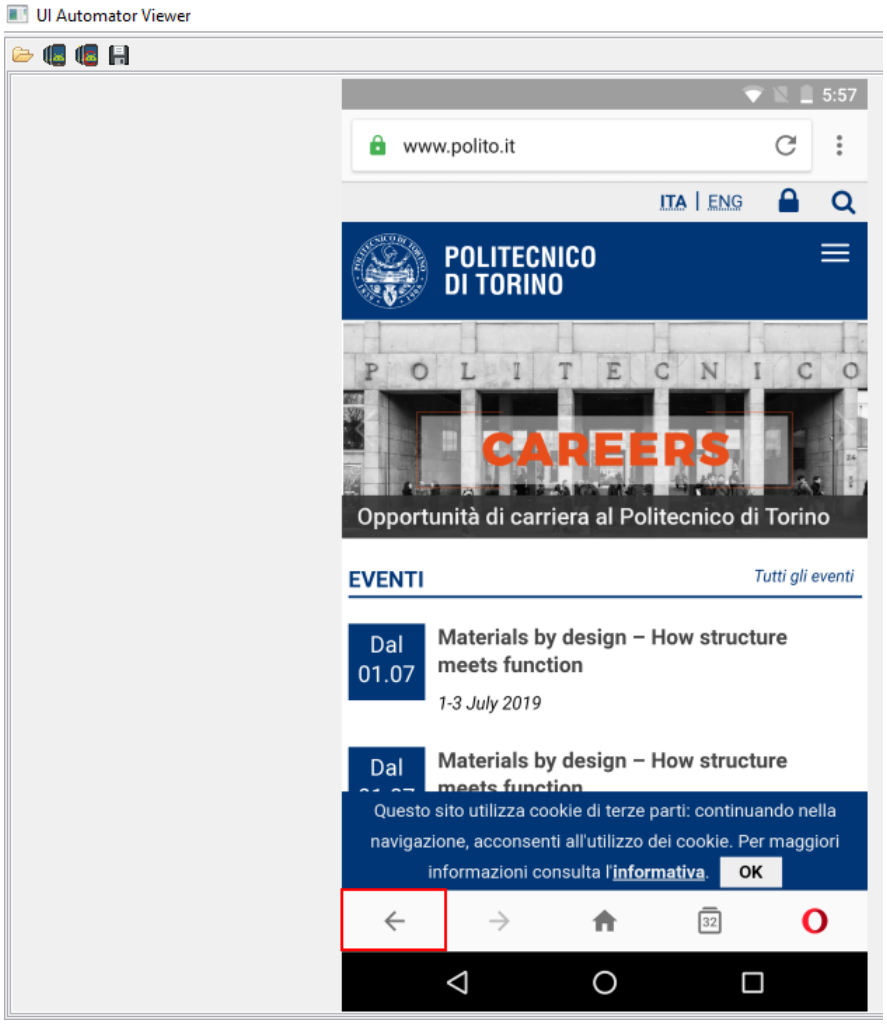

Figure 4.5. Example of screenshot taken by "UIAutomatorViewer"

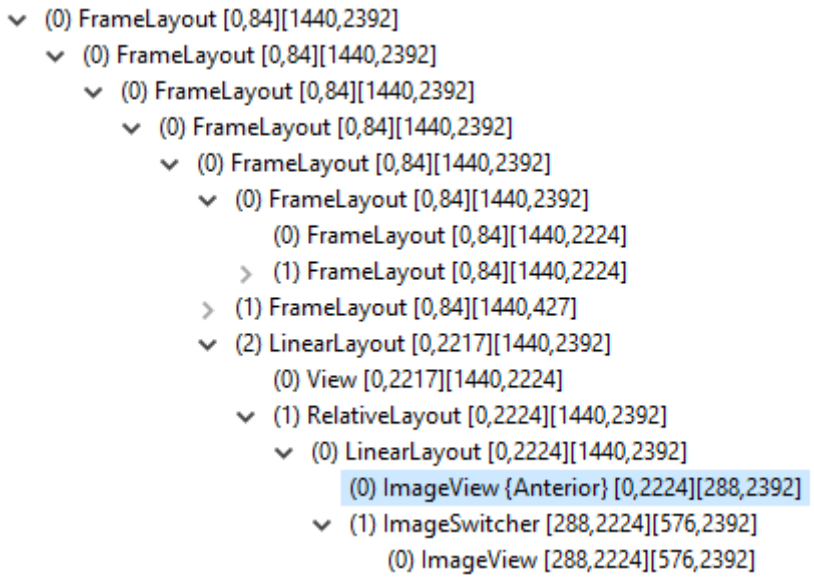

Figure 4.6. Hierarchy of the elements of the current activity

Node Detail	
index	0
text	
resource-id	com.opera.browser:id/bottom_navigation_bar_back_button
class	android.widget.ImageView
package	com.opera.browser
content-desc	Anterior
checkable	false
checked	false
clickable	true
enabled	true
focusable	true
focused	false
scrollable	false
long-clickable	true
password	false
selected	false

Figure 4.7. Information needed to identify an element within the hierarchy

In the next figure 4.8 can be seen how we create the orders to be executed in the scenario class. We will use as an example the scenario created for web browsing, because it includes almost all the different methods that we will use for the creation of orders for the device. In this figure we can see how we first enter by keyboard, with the sendkeys() method, the url to which we want to navigate, in this case "www.polito.it". Then we unfold the menu of the page and select different tabs, performing a conventional navigation, using different paths to locate the elements and indicating also the text content to help with the location. Finally, we close the connection with the quit() method.

```
driver.findElementById("url_field").click();
driver.findElementById("url_field").sendKeys("www.polito.it");
Thread.sleep(5000);
driver.findElementById("right_state_button").click();
Thread.sleep(5000);
driver.findElementById("menuTopCollapse").click();
Thread.sleep(5000);
driver.findElementByXPath("//android.widget.Spinner[contains(@text,'DIDATTICA E STUDENTI')]").click();
Thread.sleep(5000);
try{
driver.findElementByXPath("//android.view.View[contains(@text,'Studenti internazionali')]").click();
Thread.sleep(5000);
driver.findElementByXPath("//android.view.View[contains(@text,'Studenti internazionali')]").click();
Thread.sleep(5000);
}catch (NoSuchElementException e){

}
driver.findElementByXPath("//android.view.View[contains(@text,'INFORMAZIONI PRATICHE')]").click();
Thread.sleep(5000);
driver.findElementByXPath("//android.view.View[contains(@text,'Alloggiamento')]").click();
Thread.sleep(5000);
driver.findElementByXPath("//android.view.View[contains(@text,'https://www.cercoalloggio.com/#!/international')]").click();
Thread.sleep(15000);
driver.findElementByXPath("//android.widget.Button[contains(@text,'Toggle navigation')]").click();
Thread.sleep(5000);
driver.findElementByXPath("//android.view.View[contains(@text,'CHI SIAMO')]").click();
driver.quit();
```

Figure 4.8. Example of the instructions created in Java to be executed in the scenario class

- **Getting battery statistics about the execution**

  After the execution of each scenario, the method getBatteryStats("command") is called. This will execute the next command:

  <adb shell dumpsys batterystats > batterystats.txt

```
public static void getBatteryStats(String cmd){

    try {
      Runtime.getRuntime().exec(cmd);

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Figure 4.9. Example of the method for executing an adb command

With this, we´ll create the "batterystats.txt" file with all the information about the battery consumption of the scenario. Previously, in section 3.1.1, we discussed the contents of this text file and all the information that we could obtain from it.

- **Reading "batterystats.txt" file, getting the battery consumption and inserting into database**

  As we can see in figure 4.10, after getting the "batterystats.txt" file we call the main method of the class "BatteryStats_EachScenario". This method does the following:

  First, we use the BufferedReader and FileReader classes to read each line of the file. While reading each line, we check if that line contains the different keywords from each line we need, such as "Estimated power use", "Capacity", "Uid...".
  Finally, if it contains them, we build an array in which each

position contains a word of that line and with it, we can access the desired positions and guide the data of consumption at each variable, because we know the position of each value that interests us of that line.
The following figure 4.10 shows the Java code and how we do what we've just explained now.

```
FileReader f = new FileReader(archivo);
BufferedReader b = new BufferedReader(f);
String cadena;
int sum=0;
while((cadena = b.readLine())!=null) {
    if(cadena.contains("Estimated power use (mAh):")){
        sum++;
    }
    if(cadena.isEmpty()){
        sum=0;
    }
    if(sum>0){
        if(cadena.contains("Capacity")){
            String [] cadena2 = cadena.split(" ");
            deviceCapacity = cadena2[5];
            totalConsumption=  cadena2[8];
            deviceCapacity = deviceCapacity.substring(0, deviceCapacity.length() - 1);
            totalConsumption = totalConsumption.substring(0, totalConsumption.length() - 1);
        }
        if(cadena.contains("Uid u0a"+uid0)){
            String [] cadena2 = cadena.split(" ");
            appConsumption = cadena2[6];
            for(int i=0;i<cadena2.length;i++){
                if(cadena2[i].contains("Including")){
                    appConsumption = cadena2[i+2];
                }
                if(cadena2[i].contains("wifi")){
                    WiFiconsumption = cadena2[i];
                    WiFiconsumption = WiFiconsumption.substring(5);
                }
                if(cadena2[i].contains("cpu")){
                    CPUconsumption = cadena2[i];
                    CPUconsumption = CPUconsumption.substring(4);
```

Figure 4.10. Java code for reading "batterystats.txt" file and getting the battery consumption

After saving the corresponding value to each variable, we continue saving these values in the databases we have created, one for each application. These databases are local, use SQLite [2] and they are created using the "DB Browser[3]" program. The fields in these databases are:

- **Scenario_Id:** here it will be saved an id (from 1 to 10) to differentiate from which scenario is current test data.
- **Device:** here it will be saved the name of the testing device.
- **Capacity:** here it will be saved the total capacity of the battery, mAh.
- **Total_Consumption:** here it will be saved the total consumption of the test.
- **App_Consumption:** here it will be saved the consumption by the app itself.
- **CPU:** here it will be saved the consumption only due to the CPU.
- **WiFi:** here it will be saved the consumption only due to the WiFi.
- **Camera:** here it will be saved the consumption only due to the Camera.
- **StandBy:** here it will be saved the consumption of the phone due to the StandBy during the test.

For using these databases, first in Java we must create the connection with our local database. To do this, we use the "Connection" class, with which it is created a “connect” object to which we assign the file "(name).db" to create the connection. We can see it in the code in the following figure 4.11:

[2]SQLite: https://www.sqlite.org/index.html

[3]DB Browser: https://sqlitebrowser.org/

```
//Database connection
    try {
        Class.forName("org.sqlite.JDBC");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    String url = "C:\\JAVA_APPIUM2\\Script_Final\\DataBase\\Consumption_"+appName+".db";

    try {
        connect = DriverManager.getConnection("jdbc:sqlite:"+url);
        if (connect!=null) {
            System.out.println("Connected to Database");
        }
    }
    catch (SQLException e) {
    e.printStackTrace();
    }
```

Figure 4.11. Java code for creating the connection with the database

After creating the connection to the database, from the "connect" object we create a statement to execute the query. This query will be a String object with all the data that we want to insert into our database. The format we should use for the construction of this String should follow the SQLite format, which is as follows: "INSERT INTO table_name (fields of the table) VALUES (data to insert to each field);".

We can see it in the code in the following figure 4.12:

```
connect.setAutoCommit(false);
Statement stmt = connect.createStatement();
String sql = "INSERT INTO Consumption_"+appName+" (Scenario_Id, Device, Capacity,"
      + " Total_Consumption, Screen, App_consumption, CPU,"
              + " WiFi, Camera, StandBy) " +
              "VALUES ('"+appName+" "+id+"','"+deviceBrand+" "+deviceModel+"','"
              +deviceCapacity+"','"+totalConsumption+"','"+screen+"','"
              +appConsumption+"','"+CPUconsumption+"','"+WiFiconsumption+"','"
                      + "'"+cameraConsumption+"','"+standby+"');";
stmt.executeUpdate(sql);
stmt.close();
connect.commit();
```

Figure 4.12. Java code for inserting into database all the data

- **Reset the battery statistics and begin again**

  Finally, after saving all the data we have collected from the scenario in the database, we reset the battery statistics and begin again executing the same scenario. We will do these 30 times for each different scenario.

8. GETTING INFO FROM DATABASES AND CREATING GRAPHICS

The final step is to show the user the data obtained about the test itself and the comparison of this data with the other measurements performed to different device models.
For this, we decided that the best way to display the data to the user was by creating a graph with all the data obtained from the current test and the data about different models tested above.

There are different ways to create charts. Because of that our intention is to perform the entire process automatically, we decided to look for a tool compatible with our work that would collect the data generated by the test and the data from previous results and build a chart in an automatic way. After evaluating different options, we decided to use Java's own tools, to simplify the work and have all the job content designed in one language.

One tool available in Java that is the one that we will use is the JFreeChart library [14]. JFreeChart is a free tool which allows the construction of visually pleasing charts writing simple codes in Java, and it contains a lot of documentation on its website. Through this page, we can download the library that we will add to the build and work path of our project.

For the creation of these charts, we must first obtain all the values that we have saved from the execution of the different tests in our databases. To do this, we create the "BatteryStats_Reader" class. This class contains a reading method that does the following:
First we create the connection with our local database, as explained before. After this connection, we create the statement, to execute the query. We create the String following the SQLite format, but in this case of selection and not insert.

It will be like this: “SELECT * FROM table_name”. With this query we get all the results that we have saved in the database of each scenario and we get the different models of devices that exist in our database. We will save each different device name in our first ArrayList “Devices”.

But what we are interested in is to obtain only the average battery consumption, and also separated by each different smartphone model.

Therefore, we must make a second query specifying that we want to obtain the battery consumption of each record but separated by smartphone model. Then, when we have all the saved consumption values of all executions of the same scenario, we perform the average and save the value to a position in our second ArrayList that we have created “AverageConsumption”. This will be the value that we will show the user about the consumption of that device in that particular scenario.

The following figure 4.13 shows the Java code of what we have just explained. First we query for each available model and with a while loop we read each element of our database and accumulate the total consumption value in the variable "consumption". After reading all the elements of our database, we perform the average and save the value in the ArrayList.

```
for(int j=0;j<Devices.size();j++){
    ResultSet rs2 = stmt.executeQuery( "SELECT Total_Consumption FROM "
            + "Consumption_"+scenario+" WHERE Device='"+Devices.get(j)+"';" );
    double consumption=0;
    int sum2=0;
    while ( rs2.next() ) {
        String consumption_string = rs2.getString("Total_Consumption");
        consumption=consumption+Double.valueOf(consumption_string);
        sum2++;
    }
    double average_consumption = (consumption/sum2);
    AverageConsumption.add(average_consumption);
}
```

Figure 4.13. Java code reading from database all the data

After getting all the ArrayLists with the different total consumptions for each device, the last step is to create the desired chart. The type of chart we choose is the horizontal bar chart, since we find it attractive to the user to contemplate the differences in battery consumption of the different smartphones tested.

For creating charts with the JFreeChart tool, we divide the work into four blocks:

- **Assign the data to the chart:**

  The first step is to assign all the data that we want to be displayed in our chart. To do this we use the class "DefaultCategoryDataset" to create the object "dataset". Forcreating this object, we can use the addValue() method to insert the desired data. We will introduce three arguments, the first will be the value on the X axis, that is, the average of total consumption in mAh. The second data we will enter will be the name of the device that has that value, and finally we will enter the name of the scenario that we want to add. In the next figure 4.14 we can see how we add all the ArrayList values, on the “Web Navigation” scenario.

```
for(int i=0;i<webNavigationDevices.size();i++){
    dataset.addValue(webNavigationAverageConsumption.get(i),
            webNavigationDevices.get(i), "Web Navigation");
}
```

Figure 4.14. Java code to add data to the "dataset"

- **Chart creation:**

  For the creation of the graph, we use the class "CharFactory" and the method createBarChart() to assign the "dataset" that we created in the previous step and all the characteristics of the chart, such as the axis titles, orientation, legend, etc. We can see in the following figure 4.15 how we do it in our Java program:

```
JFreeChart chart = ChartFactory.createBarChart(
        "Battery consumption comparation devices/app", // chart title
        "Scenarios/Apps", // domain axis label
        "Battery Consumption (mAh)", // range axis label
        dataset, // data
        PlotOrientation.HORIZONTAL, // orientation
        true, // include legend
        false, // tooltips?
        false // URLs?
        );
```

Figure 4.15. Java code to assign the "dataset step and all the characteristics of the chart

- **Modify the style of the chart:**

  To modify the appearance and style of our chart, we will use the class "BarRenderer", to create the object "renderer". With different methods, we can change the colors of the chart and the bars, the margins between them, where we want the values to be displayed,etc.

We can see how we do all this in our Java class in the following figure 4.16:

```
BarRenderer renderer = new BarRenderer();
renderer.setSeriesPaint(0, Color.ORANGE);
renderer.setSeriesPaint(1, Color.RED);
renderer.setSeriesPaint(2, Color.BLUE);
renderer.setItemMargin(0);

CategoryItemLabelGenerator generator = new StandardCategoryItemLabelGenerator(
        "{2}", new DecimalFormat("0.00"));
renderer.setBaseItemLabelGenerator(generator);
renderer.setBasePositiveItemLabelPosition(new
        ItemLabelPosition(ItemLabelAnchor.OUTSIDE3, TextAnchor.HALF_ASCENT_LEFT));
renderer.setBaseItemLabelPaint(Color.WHITE);
renderer.setBaseItemLabelFont(new Font("Dialog", Font.PLAIN, 19), false);
renderer.setBaseItemLabelsVisible(true);

chart.getCategoryPlot().getRangeAxis().setRange(0, 70);
chart.getLegend().setPosition(RectangleEdge.RIGHT);
chart.getCategoryPlot().setRangeAxisLocation(AxisLocation.BOTTOM_OR_RIGHT);
chart.getCategoryPlot().setRenderer(renderer);
chart.getCategoryPlot().setOutlinePaint(Color.WHITE);
chart.getTitle().setPaint(Color.WHITE);
chart.getCategoryPlot().getRangeAxis().setLabelPaint(Color.WHITE);
chart.getCategoryPlot().getRangeAxis().setAxisLinePaint(Color.WHITE);
chart.getCategoryPlot().getRangeAxis().setTickLabelPaint(Color.WHITE);
chart.getCategoryPlot().getRangeAxis().setTickMarkPaint(Color.WHITE);
chart.getCategoryPlot().getDomainAxis().setLabelPaint(Color.WHITE);
chart.getCategoryPlot().getDomainAxis().setAxisLinePaint(Color.WHITE);
chart.getCategoryPlot().getDomainAxis().setTickLabelPaint(Color.WHITE);
chart.getCategoryPlot().getDomainAxis().setTickMarkPaint(Color.WHITE);
chart.setBackgroundPaint(Color.DARK_GRAY);
chart.getCategoryPlot().setBackgroundPaint(Color.DARK_GRAY);
```

Figure 4.16. Java code for modifying the appearance of the chart

- **Creating the image file to display the chart:**

  Finally, we only have to show the chart that we have created to the user. To do this, we build an image file "Final_Chart.png", which we will save in the directory to show the user finally. To create the image, we use the Java's own class "BufferedImage", to which we assign the chart. We can see how we do this in Java in the following figure 4.17:

```
BufferedImage image = chart.createBufferedImage(1000, 800);
ImageIO.write(image, "png", new File("Final_Chart.png"));
```

Figure 4.17. Java code to create the ".png" file

Chapter 5

Tests, results and conclusion

After we having created and detailed the tool and the whole development process, it is time to try it and run the tests on the different devices that we are about to test.

5.1 Description of the test

The actual test machine is the personal computer of the student running Windows XP, with Appium, Java and adb already configured as detailed on the previous pages, but can be any computer running the tool we have created and meeting all the requirements.

The devices which are going to be tested are the Xiaomi Redmi Note 5 of 2018, Motorola Nexus 6 of 2014 and Samsung Galaxy S3 of 2012. Requirements for devices for running the tool are:

- Android system
- USB debugging activated
- Appium apk installed
- Minimal basic operation: being able to interact in different applications without overlading your phone

For each scenario and for each device, we will run each test 30 times to achieve a result as in line with the average prolonged battery consumption as possible.

We also wanted to check if the consumption varied depending on the amount of battery available on the device. To do this, we performed the same tests when the battery percentage was 100%, 50% and 25%. We found that the results obtained were similar and that the variations were negligible, so we did not take into account the initial battery percentage of the device for the tests, but we require that the phone be 100% charged due to the long-time occupied to complete the full test.

5.2 Issues during the execution of the tests

Finally, we will briefly discuss some of the problems that we have encountered when choosing and testing these scenarios and what solutions we have found for this:

- **Security issues and permissions:** some apps like Twitter, Google Drive, Google Maps, etc, block the use of automation tools like the one we will use for security reasons. We believe that, for example Twitter, wants to prevent the development of bots for the creation of fake profiles.

- **Pre-installed apps:** we have scrapped the use of apps like Youtube, Google Chrome and Gmail because they come pre-installed from the factory on most devices, and therefore we cannot fix which version of the application we want to use for the test. We have replaced them with Vimeo, Opera Browser and K9 Mail, respectively.

- **Find items in apps:** in some apps, such as Instagram, it becomes difficult to find buttons, links or text boxes because they don't provide identifiers to them and as a result it's difficult for us

to automate some actions on these apps.

- **Apps that don't work on older versions of Android:** there are some apps that aren't available for some older versions of Android, such as Netflix. Therefore, this scenario will not be run in very old devices.

- **Predownloaded songs in Spotify:** while running the Spotify scenario on different devices, we encountered the problem that some of these had some songs downloaded, so during the execution of the stage the data about battery consumption decreased remarkably irregularly because the phone's internet connection was not used and therefore the test result was not valid for our case.

- **Adb WiFi connection:** wireless connections tend to be more unstable than wireframes. That is why during the execution of the tests, the device sometimes got disconnected with the computer, and therefore there was a failure in the execution of the test.
- **Notifications appear during the test:** depending on the devices, most of these have floating notifications enabled, which take up a certain screen space. If during the execution of the test this notification covers us some element on which we must perform some action, we get an error in the execution. That is why we turned off notifications at first.
- **Appium doesn´t initialize apps:** for Appium to initialize an app, it must be correctly indicated by the name of the activity and package, or if applicable an .apk file for you to install. If any of these indications are not correct, Appium cannot initialize the application and therefore the test cannot be performed.

5.3 Results

After having performed all the tests, we have all the data stored in our corresponding databases. Our main objective is to create a single graph that shows a comparison of the results between the different models, but it is also important to us to be able to show a sample of how we have saved all the data collected in our database. Because of this, we will show in this project a sample of the data collected for one scenario as an example.

The data that we show and that we find the most important are the following: test number identifier, brand and model of the device, total capacity in milliamps per hour, total consumption in milliamps per hour and screen consumption, also in milliamps per hour.

In the next 3 pages we can see a sample of the data collected in the Spotify app test, made for 3 different devices and executed 30 times on each device.

Regarding these tables, we can comment that, for each model, we note that the consumption due to the screen is similar for the two smartphones with the FullHD screen, but the smartphone with 2K screen has a higher consumption. Also, the total consumption results remain more or less constant during the execution of the 30 tests, although they vary more depending on the device. We see that in the smartphone Motorola Nexus 6, the difference between the maximum consumption obtained and the minimum is 4.4mAh, which represents 0.14% of the total battery capacity. These variations will be negligible because we will take the average of the 30 tests and this will be the result shown to the user.

Scenario_Id	Device	Capacity (mAh)	Total_Consumption (mAh)	Screen (mAh)
Spotify 1	Xiaomi Redmi Note 5	4000	28.1	11.3
Spotify 2	Xiaomi Redmi Note 5	4000	27.1	11.1
Spotify 3	Xiaomi Redmi Note 5	4000	26.7	11
Spotify 4	Xiaomi Redmi Note 5	4000	28.3	11.1
Spotify 5	Xiaomi Redmi Note 5	4000	26.8	11.2
Spotify 6	Xiaomi Redmi Note 5	4000	26.9	11.3
Spotify 7	Xiaomi Redmi Note 5	4000	27.1	11.1
Spotify 8	Xiaomi Redmi Note 5	4000	26.7	11.3
Spotify 9	Xiaomi Redmi Note 5	4000	26.7	10.7
Spotify 10	Xiaomi Redmi Note 5	4000	27.9	10.9
Spotify 11	Xiaomi Redmi Note 5	4000	25.9	10.8
Spotify 12	Xiaomi Redmi Note 5	4000	25.7	10.5
Spotify 13	Xiaomi Redmi Note 5	4000	26.6	10.6
Spotify 14	Xiaomi Redmi Note 5	4000	24.9	10.5
Spotify 15	Xiaomi Redmi Note 5	4000	28	10.6
Spotify 16	Xiaomi Redmi Note 5	4000	27.5	10.7
Spotify 17	Xiaomi Redmi Note 5	4000	27.6	10.5
Spotify 18	Xiaomi Redmi Note 5	4000	27	10.8
Spotify 19	Xiaomi Redmi Note 5	4000	26.9	10.8
Spotify 20	Xiaomi Redmi Note 5	4000	26.5	10.7
Spotify 21	Xiaomi Redmi Note 5	4000	26.4	10.7
Spotify 22	Xiaomi Redmi Note 5	4000	26.2	10.6
Spotify 23	Xiaomi Redmi Note 5	4000	26.2	11
Spotify 24	Xiaomi Redmi Note 5	4000	26.1	10.7
Spotify 25	Xiaomi Redmi Note 5	4000	25.7	10.4
Spotify 26	Xiaomi Redmi Note 5	4000	26.6	10.8
Spotify 27	Xiaomi Redmi Note 5	4000	26.9	10.8
Spotify 28	Xiaomi Redmi Note 5	4000	27.3	11
Spotify 29	Xiaomi Redmi Note 5	4000	28.3	10.8
Spotify 30	Xiaomi Redmi Note 5	4000	26.6	10.7

Table 5.1. Xiaomi Redmi Note 5 - Spotify app consumption

Scenario_Id	Device	Capacity (mAh)	Total_Consumption (mAh)	Screen (mAh)
Spotify 1	samsung GT-I9300	2100	14	9.93
Spotify 2	samsung GT-I9300	2100	13.4	9.5
Spotify 3	samsung GT-I9300	2100	13.6	9.55
Spotify 4	samsung GT-I9300	2100	13.3	9.44
Spotify 5	samsung GT-I9300	2100	13.1	9.28
Spotify 6	samsung GT-I9300	2100	13.1	9.26
Spotify 7	samsung GT-I9300	2100	13.1	9.27
Spotify 8	samsung GT-I9300	2100	13	9.24
Spotify 9	samsung GT-I9300	2100	13.3	9.42
Spotify 10	samsung GT-I9300	2100	13.1	9.27
Spotify 11	samsung GT-I9300	2100	13.21	9.32
Spotify 12	samsung GT-I9300	2100	13.2	9.35
Spotify 13	samsung GT-I9300	2100	13	9.25
Spotify 14	samsung GT-I9300	2100	13.1	9.32
Spotify 15	samsung GT-I9300	2100	13.1	9.28
Spotify 16	samsung GT-I9300	2100	13.2	9.39
Spotify 17	samsung GT-I9300	2100	12.9	9.17
Spotify 18	samsung GT-I9300	2100	13.2	9.35
Spotify 19	samsung GT-I9300	2100	13.1	9.31
Spotify 20	samsung GT-I9300	2100	13.3	9.46
Spotify 21	samsung GT-I9300	2100	13.1	9.3
Spotify 22	samsung GT-I9300	2100	13.9	9.84
Spotify 23	samsung GT-I9300	2100	13.5	9.39
Spotify 24	samsung GT-I9300	2100	13.2	9.39
Spotify 25	samsung GT-I9300	2100	13.3	9.41
Spotify 26	samsung GT-I9300	2100	13.1	9.31
Spotify 27	samsung GT-I9300	2100	13.2	9.38
Spotify 28	samsung GT-I9300	2100	13.4	9.53
Spotify 29	samsung GT-I9300	2100	13.3	9.46
Spotify 30	samsung GT-I9300	2100	13.4	9.34

Table 5.2. Samsung GT-I9300 - Spotify app consumption

Scenario_Id	Device	Capacity (mAh)	Total_Consumption (mAh)	Screen (mAh)
Spotify 1	motorola Nexus 6	3220	35.6	19.9
Spotify 2	motorola Nexus 6	3220	34.1	18.2
Spotify 3	motorola Nexus 6	3220	34.2	18.7
Spotify 4	motorola Nexus 6	3220	34.3	20.7
Spotify 5	motorola Nexus 6	3220	35.2	20.8
Spotify 6	motorola Nexus 6	3220	36.1	21.5
Spotify 7	motorola Nexus 6	3220	35.7	20.8
Spotify 8	motorola Nexus 6	3220	36.6	21
Spotify 9	motorola Nexus 6	3220	34.8	19.9
Spotify 10	motorola Nexus 6	3220	35.8	20.9
Spotify 11	motorola Nexus 6	3220	34.6	21.2
Spotify 12	motorola Nexus 6	3220	33.8	22.6
Spotify 13	motorola Nexus 6	3220	33.7	22.1
Spotify 14	motorola Nexus 6	3220	34.6	22.6
Spotify 15	motorola Nexus 6	3220	33.5	22.3
Spotify 16	motorola Nexus 6	3220	35.4	23.2
Spotify 17	motorola Nexus 6	3220	34.1	22
Spotify 18	motorola Nexus 6	3220	33.8	19.4
Spotify 19	motorola Nexus 6	3220	34.8	19.5
Spotify 20	motorola Nexus 6	3220	34.9	19.6
Spotify 21	motorola Nexus 6	3220	34	19.4
Spotify 22	motorola Nexus 6	3220	32.7	19.6
Spotify 23	motorola Nexus 6	3220	32.2	19.2
Spotify 24	motorola Nexus 6	3220	33.7	19.4
Spotify 25	motorola Nexus 6	3220	33.3	19.3
Spotify 26	motorola Nexus 6	3220	33.2	19.5
Spotify 27	motorola Nexus 6	3220	33.5	20.1
Spotify 28	motorola Nexus 6	3220	33.5	21.8
Spotify 29	motorola Nexus 6	3220	33.3	21.8
Spotify 30	motorola Nexus 6	3220	33.2	21.7

Table 5.3. Motorola Nexus 6 - Spotify app consumption

After these tables with all the data, we show below the final graph that results from collecting all the data from the databases and mounting the graph, as explained in step 8 of the section 4 "Procedure".

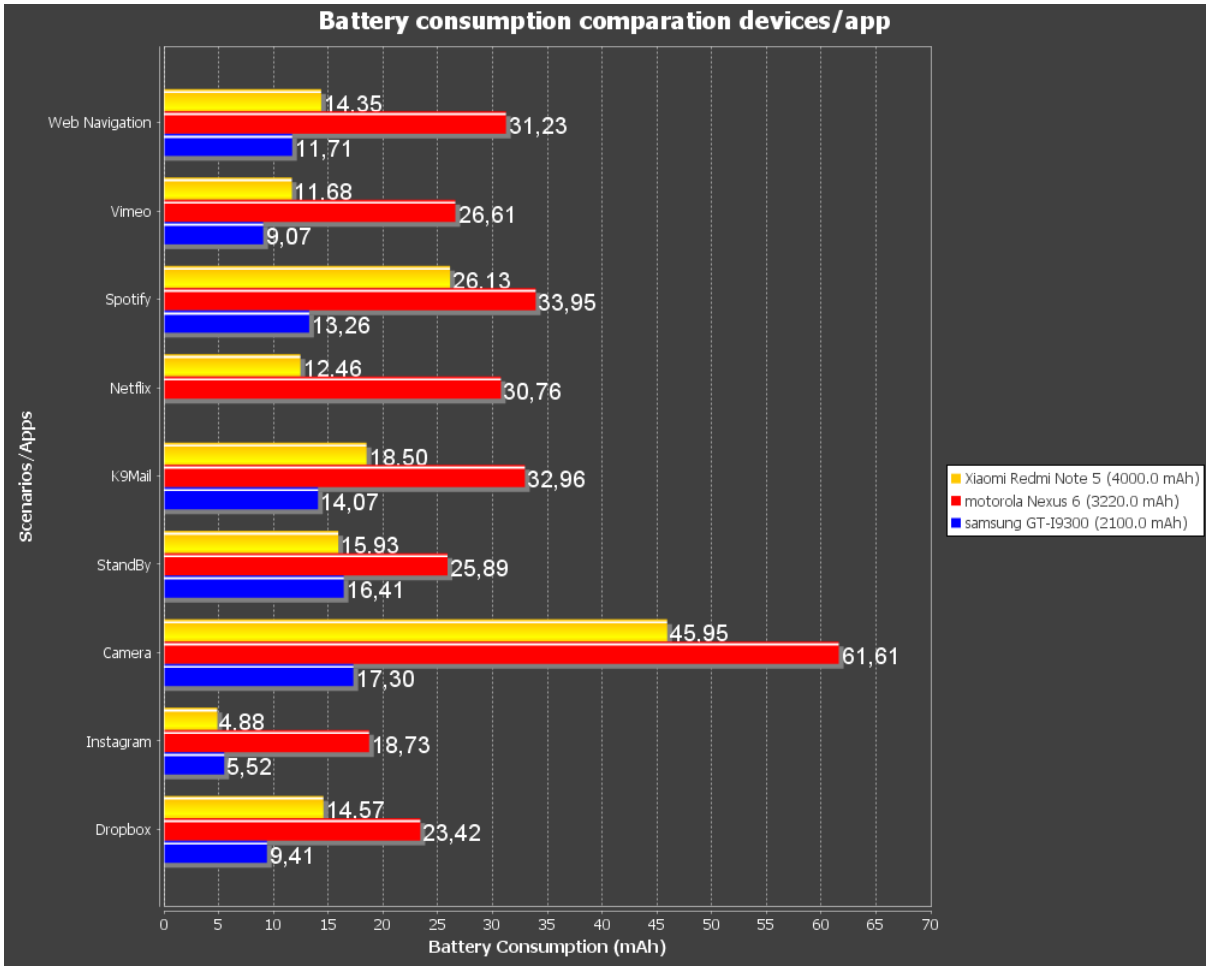

Figure 5.1. Final chart with the results of the test

On the left, on the Y axis, we can observe the different scenarios that we have raised in our thesis. Each scenario has three bars with different colors, representing each device on which the test has been run. The X axis represents the mAh value consumed in total on average during the test.

We can see that the Motorola Nexus 6 device is the one with the most battery consumption in each scenario, due to its high screen resolution and also high processor capacity. The other two devices have similar consumption in some scenarios, but the consumption is also different when the camera comes into play, because they are devices quite a few years apart, in which the camera has undergone a high development.

We can also see the differences in consumption in the Instagram scenario. We see that the Motorola Nexus 6 has a significantly higher consumption of the other two models, due to what we said before that it has a more advanced camera and processor.

5.4 Conclusion

In order to finish this thesis, it is necessary to detail a series of conclusions obtained during this thesis.

Throughout this period, we have studied different tools with different objectives. First, we studied different tools to obtain information about battery consumption in a mobile device, finally opting for tools developed by software. Next, we studied automation tools, first investigating GUI automation techniques and finally opting for software automation, with Appium. Finally, after describing the different usage profiles and all the scenarios, we proceeded to carry out the tests.

It can be commented that the results obtained have been in accordance with what was expected before the execution of the tests. We have been able to observe the differences that exist in the consumption when the camera comes into play, obtaining greater consumption in terminals with more advanced cameras. It can be also seen differences in consumption when the screen is larger and has a higher resolution.

As regards future work, it might be interesting to continue researching other automation tools developed in recent years, because tools may appear in years after the presentation of this thesis to correct the errors

that have those currently available. It could also be useful for running the tests to separate Android devices depending on the version of operating system they run, so as to obtain more accurate results by completely eliminating the differences that exist in the optimization of the software by the different versions of Android.

Appendix A

Tables about scenario´s details

Use case	Battery consumption due to StandBy
Scope	Measure battery consumption due to the consumption when leaving the phone on without performing any activity
Level	The user will have a tool to compare the battery consumption of different smartphones, and thus have the ability to choose which model to buy.
Intention in context	Development of a tool to compare battery consumption in different smartphone models
Primary actor	This primary actor it´s a smartphone´s common user
Preconditions	Preconditions: Battery charged at 100%, closed background applications, maximum level of brightness on the screen, the lowest number of activated services, a Google Play account with Spotify and Netflix premium,free storage space for downloading songs, device with internet access, screen lock dis-activated. Technical characteristics of the smartphone: *To complete by the user*
Main success scenario	1) Locate on the main screen and leave the phone 4 minutes with the screen on

Table A.1. SCENARIO 1 – StandBy consumption

Use case	Music multimedia consumption on Spotify
Scope	Measure battery consumption due to multimedia music with the app of Spotify
Level	The user will have a tool to compare the battery consumption of different smartphones, and thus have the ability to choose which model to buy.
Intention in context	Development of a tool to compare battery consumption in different smartphone models
Primary actor	This primary actor it´s a smartphone´s common user, specifically a multimedia music consumer
Preconditions	Preconditions: Battery charged at 100%, closed background applications, maximum level of brightness on the screen, the lowest number of activated services, a Google Play account with Spotify and Netflix premium, free storage space for downloading songs, device with internet access, screen lock disactivated. Technical characteristics of the smartphone: *To complete by the user*
Main success scenario	1. Locate on the main screen and open Spotify App. Spotify main page is shown 2. Click on the search button, write "Baila Reggaeton" playlist and press on "search". A list of available playlists will be shown. 3. Click on the first playlist and then click on the green button "play". The song screen will be shown 4. Wait 20 seconds and then click for the next song 5. Repeat the step 4 five times and close the app completely

Table A.2. SCENARIO 2 - Music: Spotify

Use case	Video multimedia consumption on Vimeo
Scope	Measure battery consumption due to multimedia video in the app of Vimeo
Level	The user will have a tool to compare the battery consumption of different smartphones, and thus have the ability to choose which model to buy.
Intention in context	Development of a tool to compare battery consumption in different smartphone models
Primary actor	This primary actor it´s a smartphone´s common user, specifically a multimedia video consumer
Preconditions	Preconditions: Battery charged at 100%, closed background applications, maximum level of brightness on the screen, the lowest number of activated services, a Google Play account with Spotify and Netflix premium, free storage space for downloading songs, device with internet access, screen lock disactivated. Technical characteristics of the smartphone: *To complete by the user*
Main success scenario	1. Locate on the main screen and open Vimeo App. Vimeo main screen will be shown 2. Click on "explore" and select the first category. A list of videos will be shown 3. Click on the second video. The video will start 4. Wait 30 seconds and then click again on explore. A list of categories will be shown 5. Select the second category and click on the second video. The video will start 6. Wait 30 seconds and then close completely the app

Table A.3. SCENARIO 3 - Multimedia video: Vimeo

Use case	Video multimedia consumption on Netflix
Scope	Measure battery consumption due to multimedia video in the app of Netflix
Level	The user will have a tool to compare the battery consumption of different smartphones, and thus have the ability to choose which model to buy.
Intention in context	Development of a tool to compare battery consumption in different smartphone models
Primary actor	This primary actor it´s a smartphone´s common user, specifically a multimedia video consumer
Preconditions	Preconditions: Battery charged at 100%, closed background applications, maximum level of brightness on the screen, the lowest number of activated services, a Google Play account with Spotify and Netflix premium, free storage space for downloading songs, device with internet access, screen lock disactivated. Technical characteristics of the smartphone: *To complete by the user*
Main success scenario	1. Locate on the main screen and open Netflix App. A list of available profiles will be shonw 2. Select the first profile. The main screen of Netflix will be shown. 3. Click on the search button, write “La casa de papel” and press on “search”. A list of available series will be shown 4. Click on the first series and then click on play. The chapter will start 5. Wait 120 seconds and then close the app completely

Table A.4. SCENARIO 4 - Multimedia video: Netflix

Use case	Using the camera for taking photos and videos
Scope	Measure battery consumption due to the use of the smartphone´s camera
Level	The user will have a tool to compare the battery consumption of different smartphones, and thus have the ability to choose which model to buy.
Intention in context	Development of a tool to compare battery consumption in different smartphone models
Primary actor	This primary actor it´s a smartphone´s common user, specifically a habitual user of the camera
Preconditions	Preconditions: Battery charged at 100%, closed background applications, maximum level of brightness on the screen, the lowest number of activated services, a Google Play account with Spotify and Netflix premium, free storage space for downloading songs, device with internet access, screen lock disactivated. Technical characteristics of the smartphone: *To complete by the user*
Main success scenario	1. Locate on the main screen and open Open Camera App. The interface of the camera will be shown 2. Take 10 normal pictures and then change to the video mode. The video mode will be shown 3. Record a video for 1 minute. Then, turn on the flash and record another 1-minute video. 4. Change to the front camera and repeat step 2 and 3 without flash if there´s not the option 5. Close the app completely

Table A.5. SCENARIO 5- OpenCamera: Use of the camera

Use case	Using social networks: Instagram
Scope	Measure battery consumption due to the execution of an app about social networks, specifically Instagram
Level	The user will have a tool to compare the battery consumption of different smartphones, and thus have the ability to choose which model to buy.
Intention in context	Development of a tool to compare battery consumption in different smartphone models
Primary actor	This primary actor it´s a smartphone´s common user, specifically a habitual social network user of Instagram
Preconditions	Preconditions: Battery charged at 100%, closed background applications, maximum level of brightness on the screen, the lowest number of activated services, a Google Play account with Spotify and Netflix premium, free storage space for downloading songs, device with internet access, screen lock disactivated. Technical characteristics of the smartphone: *To complete by the user*
Main success scenario	1. Locate on the main screen and open Instagram App. Instagram main screen will be shown 2. Click on the search button, write "Lamborghini" and press on the first profile. The Lamborghini´s profile will be shown 3. Click on the first picture posted, click on like and then post a comment. The comment will be posted 4. Go back and then click on the camera button and post a "instastorie". The storie will automatically be uploaded 5. Click on the post button and upload a picture by posting. The posted picture will be shown 6. Close completely the app

Table A.6. SCENARIO 6 - Social Networks: Instagram

Use case	Receiving and sending email with K9 Mail
Scope	Measure battery consumption due to the execution of an email app (K9 Maill)
Level	The user will have a tool to compare the battery consumption of different smartphones, and thus have the ability to choose which model to buy.
Intention in context	Development of a tool to compare battery consumption in different smartphone models
Primary actor	This primary actor it´s a smartphone´s common user, specifically an email user
Preconditions	Preconditions: Battery charged at 100%, closed background applications, maximum level of brightness on the screen, the lowest number of activated services, a Google Play account with Spotify and Netflix premium, free storage space for downloading songs, device with internet access, screen lock disactivated. Technical characteristics of the smartphone: *To complete by the user*
Main success scenario	1. Locate on the main screen and open K9 Mail App. The main screen will be shown 2. Click on writing a new email and fill in the address, affair and the body and click on send. The e-mail will be sent and the main screen will be shown 3. Repeat step 2 but in this case attaching a file. The e-mail will be sent and the main screen will be shown 4. Click on send and then close the app completely

Table A.7. SCENARIO 7 - Use of email: K9 Mail

Use case	Web navigation with Opera Browser
Scope	Measure battery consumption due to the execution of a web navigation app, specifically Opera Browser
Level	The user will have a tool to compare the battery consumption of different smartphones, and thus have the ability to choose which model to buy.
Intention in context	Development of a tool to compare battery consumption in different smartphone models
Primary actor	This primary actor it´s a smartphone´s common user, specifically an internet user
Preconditions	Preconditions: Battery charged at 100%, closed background applications, maximum level of brightness on the screen, the lowest number of activated services, a Google Play account with Spotify and Netflix premium, free storage space for downloading songs, device with internet access, screen lock disactivated. Technical characteristics of the smartphone: *To complete by the user*
Main success scenario	1. Locate on the main screen and open Opera Browser app. The search page will be shown 2. Click on the search tab and write “www.polito.it”. It will be shown the page of the Polito 3. Click on the menu, click on “didattica e student” and click on “Studenti Internazionali”. A list of available links will be shown 4. Click on “informazioni pratiche”, then click on “Alloggiamento” and then click on the link to “CercoAlloggio2". Cerco Alloggio page will be shown 5. On this new page click on the menu and then click on “chi siamo”. A informative text will be shown 6. Close completely the app

Table A.8. SCENARIO 8 - Web navigation: Opera Browser

Use case	Conducting a test to exchange a file using WiFi connectivity
Scope	Measure battery consumption due to the execution of a file exchange by WiFi
Level	The user will have a tool to compare the battery consumption of different smartphones, and thus have the ability to choose which model to buy.
Intention in context	Development of a tool to compare battery consumption in different smartphone models
Primary actor	This primary actor it's a smartphone's common user who upload and download some files from the web
Preconditions	Preconditions: Battery charged at 100%, closed background applications, maximum level of brightness on the screen, the lowest number of activated services, a Google Play account with Spotify and Netflix premium, free storage space for downloading songs, device with internet access, screen lock disactivated. Technical characteristics of the smartphone: *To complete by the user*
Main success scenario	1. Locate on the main screen and open Dropbox App. The Dropbox main screen will be shown 2. Click on upload a new file. The menu with the internal files of the device will be shown 3. Locate on the downloads page. A list of files downloaded on the device will be shown 4. Select the 512MB file preloaded. The upload will start 5. Wait until it´s uploaded and then close the app completely

Table A.9. SCENARIO 9 - Data Transfer: Dropbox

Bibliography

[1] Mohammad Rashid, Luca Ardito, and Marco Torchiano. *Energy Consumption Analysis of Algorithms Implementations*. Department of Control and Computer Engineering, Turin, Italy.

[2] Luca Ardito, Marco Torchiano, Maurizio Morisio. *Getting Energy Consumption of Software Applications*. e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009.

[3] Ardito L., Torchiano M., Marengo M., Falcarin P.. *gLCB: An Energy Aware Context Broker*. In: SUSTAINABLE COMPUTING, vol. 3, pp. 18-26. - ISSN 2210-5379

[4] Luca Ardito and Marco Torchiano. *Creating and Evaluating a Software Power Model for Linux Single Board Computers*.GREENS 2018, May 27 2018, Gothenburg, Sweden.

[5] Mohammad Rashid; Luca Ardito; Marco Torchiano. *Energy Consumption Analysis of Image Encoding and Decoding Algorithms* In: 4th International Workshop on Green and Sustainable Software (GREENS), 2015, Firenze, Maggio 2015. pp. 15-21.

[6] Luca Ardito, Giuseppe Procaccianti, Marco Torchiano, Giuseppe Migliore. *Profiling Power Consumption on Mobile Devices*. Dipartimento di Automatica ed Informatica, Politecnico di Torino (Torino, Italy).

[7] Giuseppe Procaccianti, Antonio Vetro', Luca Ardito, and Maurizio Morisio. *Profiling Power Consumption on Desktop Computer Systems*. D. Kranzlm¨uller and A M. Tjoa (Eds.): ICT-GLOW 2011, LNCS 6868, pp. 110–123, 2011. Springer-Verlag Berlin Heidelberg 2011.

[8] Luca Ardito, Maurizio Morisio. *Green IT – Available data and guidelines for reducing energy consumption in IT systems*. Sustainable

Computing: Informatics and Systems 4 (2014) 24–32.

[9] Android Developers. *Dumpsys with Batterystats.* [Online]. Available: https://developer.android.com/studio/command-line/dumpsys.

[10] Vysor Developers. *Vysor for Google Chrome.* [Online]. Available: https://www.vysor.io/.

[11] Kiem Ton, Evgeny Shragovich, Barry J Busler, Matthew Molloy,Mathias Krizanac. *SikuliX.* [Online]. Available: http://sikulix.com/.

[12] Appium Developers. *Appium.* [Online]. Available: http://appium.io/docs/en/about-appium/intro/?lang=en.

[13] Android Developers. *UI-Automator.* [Online]. Available: https://developer.android.com/training/testing/ui-automator.

[14] Andreas Viklund. *JFreeChart.* [Online]. Available: http://www.jfree.org/jfreechart/.

[15] Abdifatah Farah Ali ,Najwa Hanis, Mohd Hatta ,Koh Tieng Wei. *Literature review on energy consumption and conservation in mobile device.* ISSN: 1992-8645

[16] Mohammad Tawalbeh Alan Eardley, Lo'ai Tawalbeh. *Studying the Energy Consumption in Mobile Devices.* Procedia Computer Science 94 (2016) 183-189.

[17] Janine Morleya, KellyWiddicksb, Mike Hazasb. *Digitalisation, energy and data demand: The impact of Internet traffic on overall and peak electricity consumption.* Energy Research & Social Science Volume 38, April 2018, Pages 128-137.

[18] IngeRøpkeaToke, HaunstrupChristensenb. *Energy impacts of ICT - Insights from an everyday life perspective.* Telematics and Informatics Volume 29, Issue 4, November 2012, Pages 348-361.

[19] Ashwaq A. Alotaibi, Rizwan J. Qureshi. *Novel Framework for Automation Testing of Mobile Applications using Appium.* I.J. Modern Education and Computer Science, 2017, 2, 34-40.

[20] Meilianaa Irwandhi, Septiana Ricky Setiawan Aliantoa Daniela. *Comparison Analysis of Android GUI Testing Frameworks by Using an Experimental Study.* Procedia Computer Science Volume 135, 2018, Pages 736-748.

[21] *A Deconstruction of the Appium Architecture..* [Online]. Available:

https://www.edureka.co/blog/appium-architecture/.

[22] Micah Williams. *Insights of Appium*. [Online]. Available: https://dzone.com/articles/insights-of-appium.

[23] Shiwangi Singh , Rucha Gadgil and Ayushi Chudgor. *Automated Testing of Mobile Applications using Scripting Technique: A Study on Appium*. International Journal of Current Engineering and Technology E-ISSN 2277-4106.

[24] Cisco. *Cisco Visual Networking Index: Forecast and Trends*. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html.