eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea
The University of the Basque Country

Informatics Engineering Degree
Software Engineering

Degree's Final Project

Testing the performance and feasibility of Bluetooth communications in pervasive systems

Author
Xabier Gardeazabal

Director
German Rigau

informatika fakultatea
facultad de informática

2014

Abstract

Smart and mobile environments require seamless connections. However, due to the frequent process of ”discovery” and disconnection of mobile devices while data interchange is happening, wireless connections are often interrupted. To minimize this drawback, a protocol that enables an easy and fast synchronization is crucial. Bearing this in mind, Bluetooth technology appears to be a suitable solution to carry on such connections due to the discovery and pairing capabilities it provides. Nonetheless, the time and energy spent when several devices are being discovered and used at the same time still needs to be managed properly. It is essential that this process of discovery takes as little time and energy as possible. In addition to this, it is believed that the performance of the communications is not constant when the transmission speeds and throughput increase, but this has not been proved formally. Therefore, the purpose of this project is twofold: Firstly, to design and build a framework-system capable of performing controlled Bluetooth device discovery, pairing and communications. Secondly, to analyze and test the scalability and performance of the *classic* Bluetooth standard under different scenarios and with various sensors and devices using the framework developed. To achieve the first goal, a generic Bluetooth platform will be used to control the test conditions and to form a ubiquitous wireless system connected to an Android Smartphone. For the latter goal, various stress-tests will be carried on to measure the consumption rate of battery life as well as the quality of the communications between the devices involved.

Keywords: Bluetooth, performance, device discovery, sensors, Android, battery, Arduino.

Acknowledgements

First and foremost, I would like to acknowledge the help and contributions made by Borja Gamecho, as well as his encouragement, without which I wouldn't have worked as enthusiastically. I would also like to thank the *Egokituz* laboratory for the resources left at my disposal, as well as its members and their support, who have made my time during the development of this project much more bearable. Finally, I would like to thank German Rigau for his guidance throughout the project.

Contents

List of Figures

List of Tables

1. CHAPTER

Project Charter

1.1 Introduction

This document is a memory for the Degree's Final Project on Software Engineering in the Faculty of Informatics of the University of the Basque Country, by Xabier Gardeazabal. In this document all the whereabouts concerning this particular project are thoroughly explained. A detailed structure of this memory is explained later in this chapter (see section 1.4).

1.2 Motivation

The idea of this work resulted from the work done with Borja Gamecho for his PhD Thesis under a collaboration grant from the *Basque Government* with the *Egokituz* laboratory. His thesis involved the use of some wireless sensors and devices in conjunction with several protocols—mainly Bluetooth—to implement a context-aware ubiquitous system.

During the development of that collaboration we discovered that the resources needed to perform Bluetooth communications exceeded what was expected, not only by means of time and synchronization, but also by battery life consumption and network saturation. We found that there was a lack of a general research in this kind of environment, and as a consequence, we concluded that such a topic could be very interesting for a Degree's Final Project.

In addition to that, there was an interest to know which was the best combination of parameters like the message size and transmission rate to achieve the largest throughput while keeping the response times (*ping*) and error rates (data losses) at a minimum. Furthermore, the fact that context-aware systems are usually composed by multiple devices makes the performance decrease, and there is not a formal mechanism or tool to measure this properly.

1.3 Proposed solution

In this project a testbed[1] to measure the performance of Bluetooth based Personal Area Networks with Android under different variables and configurations is presented. The system developed focuses on the performance of an Android smartphone as the central role in a Bluetooth network formed by a set of custom Arduino-sensors, and provides metrics for battery drain, CPU usage and communications quality. This last one is given by the response times and communications' throughput of the wirelessly (through Bluetooth) connected devices.

The developed system provides multiple configurations, such as the number of Bluetooth discoveries that the smartphone should do, when should the discovered devices be connected and how, or the availability of the devices themselves. A mechanism for stressing the Bluetooth communications to their limit by sending different loads of data from the devices to the smartphone is also provided.

1.3.1 Contributions

With this new testbed the possibility of setting a formal benchmark for Bluetooth based systems centered on Android becomes available. From the set of tests carried out with this very same testbed, some suitable configurations for a Bluetooth based Android-system are presented.

1.4 Structure of the memory

Project Charter This first chapter has briefly presented the original problem or motivation of the project and the proposed solution.

Project management Plan Outlines the management of the project, the way the it will be structured and how it will be implemented. The project vision, objectives, scope and deliverables, as well as the stakeholders, their roles and responsibilities are also described.

[1]A testbed is a piece of equipment that allows for rigorous, transparent, and replicable testing of scientific theories, computational tools, and new technologies.

State of the art In this chapter the current state of the art at the beginning of the project is explained. The possible technologies around, as well as previous studies on the same research field are reviewed as well.

Architecture and technological choice This chapter gives a very general picture of the system developed in the project, and it also explains the intricacies of the technologies chosen.

Requirements capture Here the requirements for the system that has been developed are gathered. The use cases are also briefly explained.

Analysis With the requirements at hand, an analysis about the software part of the system is made.

Design This chapter deeply describes the whole framework developed in this project. The interaction between software and hardware components of the system is explained as well.

Implementation Describes the different processes of the system—particularly of the Android application—in a technical level. It also gathers general information such as the organization of the project's files relevant for whoever wants to make changes to the different parts of the system.

Testing A brief overview on the tests done in order to identify bugs and potential problems in the system.

Testbed and benchmarking This chapter explains the different benchmarking tests done with the already developed framework. It also makes a discussion regarding the results obtained.

Monitoring and control This chapter briefly reviews the work done during the project, comparing the initial plan and expectations with the final outcome.

Conclusions and future work Gathers the conclusions of the project as a whole, as well as the future work that can be done.

2. CHAPTER

Project Management Plan

This chapter focuses entirely on the scope planing of the project and the definition of how the monitoring and control will be done. Part of the planing are the identification of the phases of the project, the scheduling plan, and resource, quality, communications and risks plans. These are the topics covered in this chapter:

- Project vision and objectives
- Scope of the project
- Critical deliverables
- Organizational structure for the project
- Overall implementation schedule
- Risks, issues and assumptions
- Stakeholders and groups of interest

2.1 Project vision and objectives

In order to get started, some achievable objectives were proposed from the beginning. These were further developed and defined with time. Below are listed some of the major goals of the project:

- Main objectives:
  - Develop a full testing framework or testbed:
    - An Android *App* to perform device-discoveries, pairing and communi-

cations with several Bluetooth sensors while it measures its own performance.
    * A sensor network to perform the tests with.
    * A tool for the analysis of data logs.
  - Obtain relevant results from the tests:
    * Measure the feasibility of Bluetooth powered PANs[1].
    * Find an efficient mechanism to improve the inherent Bluetooth device discovery algorithm of the Android OS.
    * Measure the rate of communication's interruptions in multiple Bluetooth device environments.
    * Determine the optimal configuration to achieve the best performance of the communications in a ubiquitous environment with multiple Bluetooth sensors.

- Additional objectives:
  - Script or program to automatically send commands to the Bluetooth sensors (Arduino boards[2]) from a PC through a serial connection.
  - Acquire transversal skills:
    * Gain a deep understanding of the Bluetooth technology.
    * Boost autonomous working ability.
    * Develop English writing and speech skills.
    * Learn to proficiently use tools or editors like LaTeX, Eclipse & ADT[3], Arduino IDE[4], etc.
    * Acquire general electronics knowledge.

- Optional objectives:
  - Study the difference between classic Bluetooth and Bluetooth Smart.
  - Write a research paper with the discoveries and conclusions of this work.

2.2 Scope

It should be made clear that the conclusions or product that come out of this project need not be innovators or make any contribution to the state of the art on any scientific or

[1] A personal area network (PAN) is a short-range computer network used for data transmission among devices such as computers, telephones and personal digital assistants.
[2] See section 4.2.3 for information about the Arduino Boards.
[3] Android Development Tools
[4] Integrated Development Environment

research field. Since it is a Bachelor Thesis, it will be assessed as such.

In a nutshell, the main goal is to find and measure an efficient mechanism or configuration to deal with Bluetooth discovery, connection setup and data transfers in an environment where many devices are present at the same time. Therefore, the scope should be defined around that same goal. In the following subsections the requirements, boundaries and working methodology are explained.

2.2.1 Requirements

It is expected that this project follows the guidelines defined by the Faculty of Informatics of San Sebastian for end of degree projects. Some formal requirements are listed below:

- The project must be defended and assessed once every other subject of the current Study Programme is finished
- The project means 12 ECTS credits, or a minimum of 300 hours
- The director(s) of the project must be professors at the Faculty of Informatics
- The subject of the project must be developed under the specialization chosen for the Study Programme—Software Engineering in this case.

Except for the invested hours, which will not be known until the project is finished, all the other requirements are already fulfilled.

2.2.2 Boundaries

The management part should never pose a hindrance to develop the project. Therefore, micromanagement and unnecessary bureaucracy are best avoided. Only relevant information should be gathered in the report—that is, everything that takes place within the scope of the project.

On a different mater, since limited resources are available, the research will not focus on the effect of using different Bluetooth versions, nor on factors causing uncertainties or interferences and the like. At least not unless the estimated time for all the other tasks falls so short that the minimum of 300 hours required is not reached.

Bearing this in mind, the total amount of hours invested in this project should be somewhere around 400 hours, although this is just an early estimation.

2.2.3 Methodology

What am I going to do, and how? As aforementioned, one goal of this project is to perform some tests to find out how Bluetooth devices respond in different conditions. These conditions are bound to many variables, some of which are listed below:

- Hardware variables
  - Smart-phone or mobile responsible for discovery and pairing purposes (Nexus S, Galaxy Nexus, Nexus 5...)
  - Battery charge state
  - Device synchronization (will the devices be powered on simultaneously, or one by one?)
- Signal features
  - Framing size
  - Radio interferences
  - Data reception frequency
  - Serial communications Baud rates[5]
- Other variables
  - Distance between devices (variable or controlled setting)
  - Background processes running in each device
  - How and when are the devices be powered on and off? Manually or programmatically?

All these variables make for a changing environment formed by both a hardware and software framework. However, due to the limited technical resources, not all variables can be measured—and even if still being able to measure, it may not be in a very effective way. Such is the case of the radio interferences, for example. A developed explanation on these variables is done in the requirements capture (chapter 5).

Hardware infrastructure

To properly implement this project, a fully functioning Bluetooth sensor network is required. While working with Borja Gamecho, sensors for temperature, humidity, heart-rate monitoring, luminosity and the like were used. In addition to this, some other sensors were simulated after processing those same devices' data. However, all these sensors cannot not

[5]The baud rate is the rate at which information is transferred in a communication channel, measured in bits per second.

be easily toggled, since they belong to proprietary sources. Therefore, a new handful of devices should be acquired, although there is no commercial product that satisfies the specific requirements of this project. Fortunately enough, there are some Arduino boards and Bluetooth antennas at hand, so a couple of custom made sensors can be built. There are also three Android smart-phones at hand at the moment, each with a different API or OS version.

Among the different versions of the Bluetooth standard, the one that is going to be used in this work is the V2.0+EDR, since that is the version of the antennas used on a par with the Arduino boards. These antennas come from the *DF-Robot* company—a robotics and open source hardware provider—, and have flawlessly worked so far. The technologies used are properly presented and explained in chapter 4 of the *State of the Art* chapter.

Software used

Of course, being this a Software Engineering project, many programming and software development applications are going to be used. The working environment is centered on a *Microsoft Windows XP* desktop personal computer provided by the *Egokituz* laboratory. Thus, all the software must be compatible with this OS. These are the foreseen applications to be used for the execution process:

- *Eclipse SDK* with *Android Development Tools*, in order to implement the app for controlling the system with a smartphone.
- *Arduino IDE*, to program, compile and configure the simulated sensors' code.
- *Real Term*, for capturing, controlling and debugging Bluetooth data streams in Windows OS (Note: this tool will only be used during the performance tests, and optionally).
- *Anaconda*, a Python distribution for data plotting and scripting purposes.
- *Fritzing*, open source tool to design electronic systems' wiring schemes.

In addition to this, tools related to the other processes of the project will also be used. LaTeXwas chosen to write the project's report or memory, so apart from the *TEX* engine, a visual environment like *TeXstudio* is being used. For the planning process, *Microsoft Project* was considered, but knowing how big of a program it is and the lack of experience with it, simpler but equally useful alternatives such as *Gantt Project* and *Microsoft Office Excel* are going to be used. Additionally, software for diagram development such as *Draw.io*, *Dia* and *Microsoft Visio* will also be used to generate some graphic pictures and diagrams. The slides for the defense will be done with MS Power Point.

2.3 Project deliverables

The main project deliverable is the project's memory itself. This document should follow the standard guidelines of a memory in an end of degree's project on Software Engineering, in a proper and formal format, and which thoroughly gathers the work done. On a side note, a public defense of the project has to be made as well, although the material used for and during the presentation needs not be delivered.

The Bachelor Thesis is a work protected by the Intellectual Property Law, so the ownership belongs to the creator (unless the student says otherwise). Hence, all deliverables fall under this protection. However, I believe that any academic research and study should be freely available for anyone, and have therefore chosen to release the source code of the developed system as an open-source software under the Apache v2.0 license. See section 8.1 for more details.

As of the current official regulation for the Bachelor Thesis at the Faculty of Informatics of San Sebastian, the project's memory should comply with these terms:

> ARTICULO 8: PRESENTACION Y DEFENSA DEL PROYECTO
>
> 8.1. El alumno o la alumna estando matriculado deberá presentar en la Secretaría del centro el formulario y la memoria del proyecto por medio digital, y en papel si fuera necesario.
>
> El formato de la memoria deberá seguir el estándar de la Facultad. Desde secretaría se deberá hacer llegar una copia electrónica a cada uno de los miembros que constituyan el Tribunal de Evaluación además del informe del director que habrá sido requerido. Si fuera necesario material adicional (programas etc.) este se añadirá a la memoria o se pondrá accesible en la red. Si fuera necesario otro tipo de material este se entregará en secretaría.
>
> 8.2. La defensa del PFG será pública y realizada por la estudiante o el estudiante en el lugar de la facultad y en la fecha y hora en la que se le haya convocado. Estos detalles se habrán publicado con un mínimo de tres días de antelación.

Apart from the physical deliverables, a defense in the form of a presentation should also be made. As of the current official regulation, the defense of the project should comply with these terms:

> ARTICULO 8: PRESENTACION Y DEFENSA DEL PROYECTO

> 8.3. La defensa se podrá hacer en euskera, castellano o inglés, según lo previsto al inscribir el proyecto.
>
> 8.4. Cada estudiante dispondrá de un tiempo máximo de media hora para la defensa, en la que deberá exponer los objetivos, la metodología, el contenido y las conclusiones de su proyecto, contestando con posterioridad a las preguntas, aclaraciones, comentarios y sugerencias que pudieran plantearle los miembros del Tribunal.

The project's memory must be uploaded to the *ADDI* platform of the University of the Basque Country by the 5th of September, or presented in the faculty's registrar by the same date. Beforehand, however, an final version of the memory should be sent to the director of the project so he can revise the document and give it approval.

2.4 Implementation plan

With the objectives at hand, a work breakdown structure has been defined—WBS from here now on—, where high level processes of the project are split up into smaller, more precise subtasks. This scheme will later ease the planning process and organization of the working plan. Alongside this WBS, a plan has been conceived, where a rough estimation of time and work required to accomplish the main processes of the WBS is captured. In the following sub-sections the WBS and its main processes are explained, and then the plan is presented.

2.4.1 Work Breakdown Structure

Following, the main tasks that take place during the whole life of the project are identified and organized in groups. Managing the project, researching, defining and developing the system, and closing the project are the main five key types of work that have been distinguished. These constitute the high-level tasks of the project, but they all contain their own sub-tasks as well.

Manage

The management of the project encloses the planning, control and monitoring subtasks. The planning by itself is not only formed by doing estimations on the schedule, the resources needed or the quality expected. It is also formed by the risk management plan, where preventive measures and a contingency procedure are defined.

Research

In order to better understand the context in which the project is going to be developed, a proper research on the state of the art needs to be done. Papers, articles or formal studies in any other format on the subject matter would establish a good grounding. On the other hand, there is also the need to research the technologies that are going to be used during the development phase of the project. This task is classified as personal formation or training.

Architecture

Any project worth its work undergoes an analysis and design process. In this case, these two tasks are both under the architecture phase. During the analysis, components needed to meet the project's goals are identified. Later, the internal structure of each component would properly be defined by using standard, high-abstraction methods, such as UML diagrams.

Develop

It is during this phase that the real work begins, but prior to delving into programming the work environment must be set up. After installing the required software and getting the needed tools (e.g. USB wires and hubs, Bluetooth dongles, multimeters, etc.), programming would follow. In addition to this, the smartphones, Arduinos and DF-Bluetooth antennas need to be acquired. At least three different programs are going to be developed by using three types of languages: the main application in Android, the Arduino programs in C, and the plotting scripts in Python. After the whole system is finished and bug-tested, the performance tests and benchmarking would be made, from which the final conclusions of this study will be taken.

2.4.2 Close

Near to the end of the project, tasks related to writing the memory or preparing the defense will take precedence. An insight into the work done, the problems faced and the solutions found will be made, and the experience gained that could be valuable in the future will be somehow gathered as well. Additionally, an internal report containing anything needed for other researchers willing to continue with new lines that part from this project will also be made.

Finally, all the resources borrowed from the *Egokituz* laboratory should be returned in their original state.

In figure 2.1 the WBS for the project is depicted, while in table 2.1 the high- and middle-level tasks are summarized.

Code	Name	Description
1 Manage		
1.1	Plan	Define a schedule and a risk management plan.
1.2	Control	Perform periodic checks for quality, risks and changes.
1.3	Monitoring	Meet periodically with the stakeholders and update planning and requirements if necessary.
2 Research		
2.1	State of the Art	Study previous papers on the topic, similar studies and overall context of the project.
2.2	Training	Learn about Bluetooth, basic electronics, Android, Arduino and Python (among others).
3 Architecture		
3.1	Analyze	Find out what system components are needed to meet the requirements.
3.2	Design	Define the system's components and communications between each other.
4 Develop		
4.1	Setup work environment	Arrangements before actual development can start.
4.2	Build Arduino boards	Setup the DF-Bluetooth antennas and wire them to the Arduino boards.
4.3	Program	Code the Android App, the Arduinos, and Python scripts.
4.4	Test performance	Collect data while performing the different tests and study the results.
5 Close		
5.1	Write memory	Thoroughly document and describe all the work done, plus conclusions.
5.2	Internal report	Formal closure of the project.
5.3	Defense	Prepare and train for the project's defence before the court.

Table 2.1: High and middle level tasks of the Work Breakdown Structure.

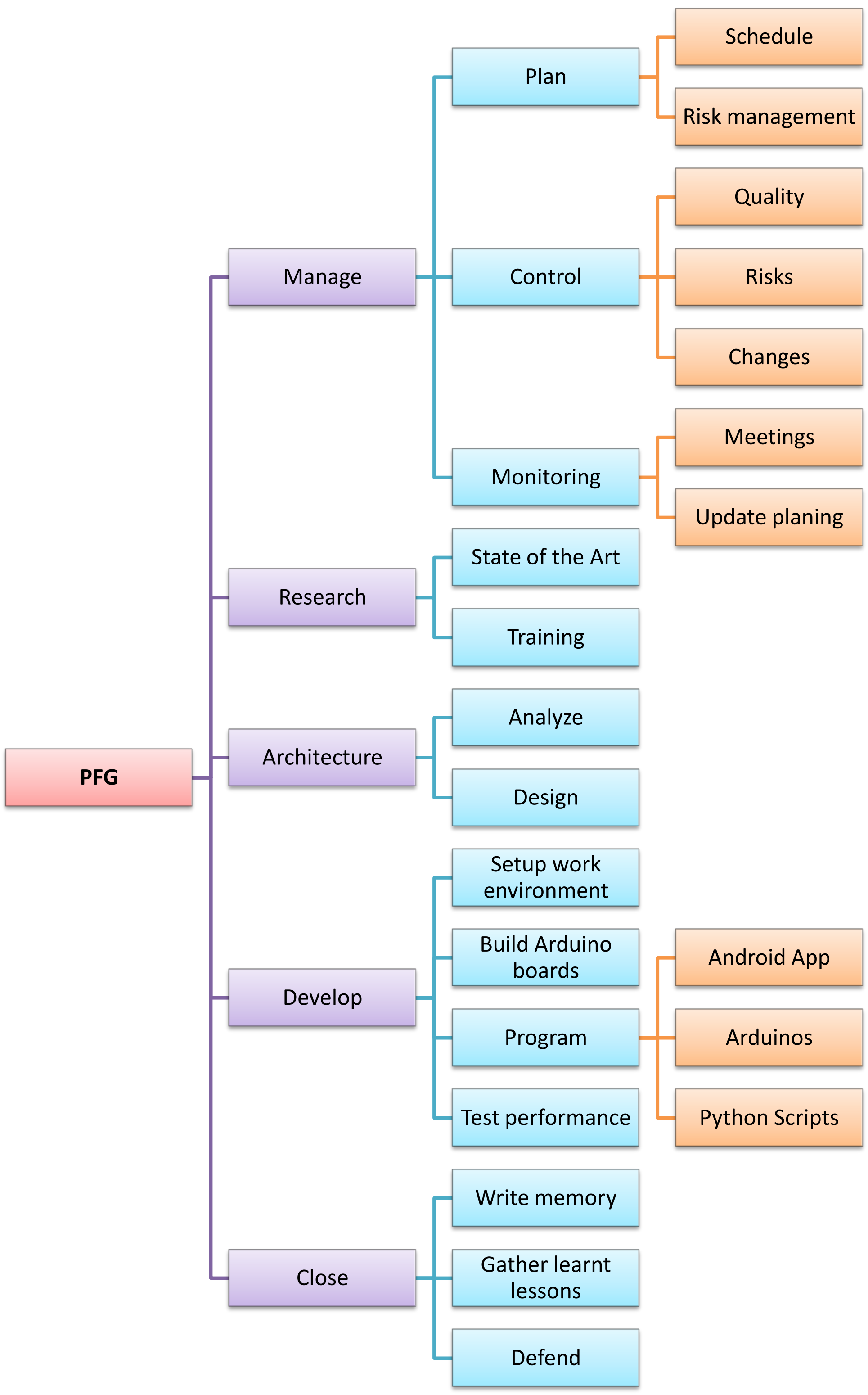


Figure 2.1: Work Breakdown Structure of the project

.

2.4.3 Management and scheduling

The final deadline to present this project's documentation is September the 5th. By that time, the project as a whole must be finished, and then in a two week's time the defense would take place. The project by itself consists of 12 ECTS credits—or 300 work hours, at least—, so there's room for error in the planning.

Taking the previous paragraph into account, the idea is to complete the project in 400 hours, spanned along 5 months. However, it should be taken into account that the learning and formation process of the involved technologies has been in progress for a couple of months now, so that task would not take much more workload.

A deeper planning and schedule is presented in the next chapter.

2.4.4 Risk Management Plan

The risks this project is exposed to are similar to any other Information Technology related project. Unexpected bugs in the software being used, version incompatibility, etc. are just the top of the iceberg. Updates that come from the management process of the project could inadvertently cause modifications to the requirements, objectives or scope of the project, risking the fulfillment of deadlines and enlarging the foreseen costs.

On the other hand, since this project requires the use of several fragile electronic hardware devices, should one or more of these show misbehavior or even breakage, the whole project could be unsettled.

Finally, since most of the work done is digitally stored, the many ways in which that information could be lost must be taken as a potential threat.

In order to keep the work done safe, software like *Dropbox* and *Google Drive* (or any other cloud storage service) that make automatic copies of files and store them on *the cloud* will be used. In addition to this, the use of *Git*[6] and *GitHub*[7] will bring yet another security layer, as well as provide help in the organization and version control process of the code. Nevertheless, weekly backups of all the project's digital assets will be manually done as well.

As for the hardware goes, there's no thing to do other than carefully and thoroughly manipulate the different devices. Training, formation and proper habits could help to minimize accidents or slip-ups.

[6] *Git* is a free and open-source distributed revision control and source code management system
[7] *GitHub* is a *Git* repository web-based hosting service

Taking these initial risks and countermeasures into account, a proper risk management plan was developed (see section 2.7.2).

2.5 Phases of the project

For the development of this project, a lineal form Life Cycle will be followed. There will be a total of four main phases:

- Initiation
- Planning
- Execution
- Closure

The total level of effort dedicated to each phase will vary as the project advances in time, and they may overlap between one another.

Initiation

At the beginning of the initiation phase, the project's main idea was conceived, and the process that gave birth to all this work started. The members of the interest group decided the whereabouts of the project, discussed the feasibility of the purposes, and a general draft was written. Later on, a more elaborated project charter was developed, where the limits within which the project must be delivered were defined. The vision, objectives, scope and implementation of the project were set out, thereby giving the stakeholders clear boundaries where the project would be developed and delivered.

Planning

During the planning phase, issues such as the identification of all the phases, activities and tasks, summing up of the effort needed to complete those tasks, or documenting all of the work's inter-dependencies are covered. Some assumptions will be made for the schedule and risks.

Execution

Once the project's concept is formally defined and an scheduled plan is made, it's time for the execution. This phase encompasses the work that should be done in order to accomplish the goals defined at the beginning. It is during this phase that physical testbed will be produced, which may as well be called the main asset or product of the project. Prior to producing anything, though, a self formation or training around the state of the art needs to take place. Control and monitoring tasks will go along with the execution at all times.

As previously stated in the Project Charter, so as to counterbalance the divergence between the work done and the initial plan, an iteration based implementation will be followed. Taking ideas from existing iterative and incremental development methods, I devised one of my own. Knowing that my working days could be long, I decided that daily iterations would keep me most motivated, since in this way I would not focus for very long hours on the same task. Even in the case of getting stuck or frustrated with some issue, by following the daily iteration I could leave it to wind down and tackle it with a new perspective the next day. Figure 2.2 depicts the iteration for each day.

Beginning with the definition of what should be done within the day, the agenda for the following hours should be organized. If necessary, a portion of the work will directly go training and learning. Otherwise, the execution process would start. Part of the execution are the development and testing of the system, meetings and any other thing that has directly something to do with the product of the project. Near the end of the day, it'll be time to document the progress done, and any updates on past work will be made if necessary. As the day's output, things left undone and possible *TO-DO*s for tomorrow should be written.

Closure

At this final stage of the project's life cycle, the final report (this document) will be delivered. Arrangements should be done for other people who would like to continue with the project.

2.6 Project implementation plan

In the following section an overall estimation of work time and effort required to fulfill this project is made. Then, a scheduling is proposed, where the different tasks defined in the WBS (figure 2.1) take a place and length in time.

2.6.1 Schedule

The estimations done for the working time in each main task are based on past experience working on related topics and projects. Nevertheless, they may not be as accurate as

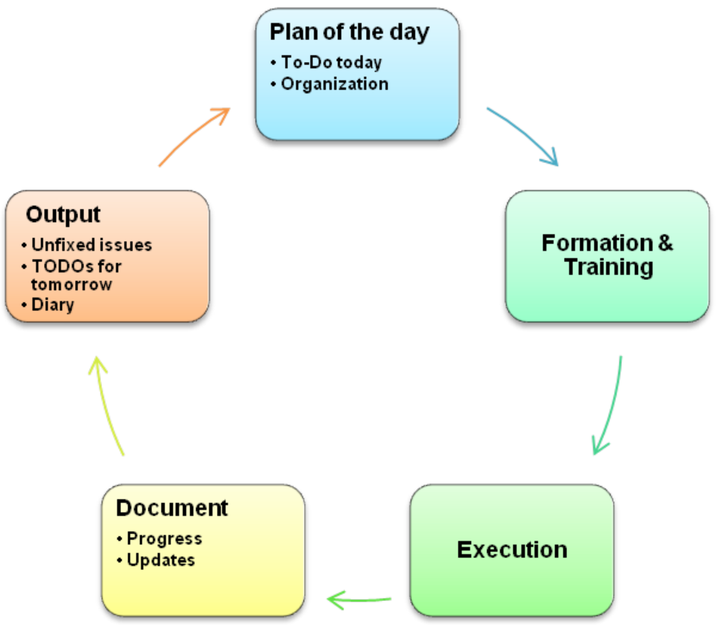


Figure 2.2: Daily iteration cycle.

desired, and it would be only natural for deviations to happen during the development of the project, since I have not ever taken part in a project of this caliber.

In figure 2.3 the calculated load of each high-level task is captured, while in the figure 2.4 a more detailed load-balance with time estimations and spans (expected starting and ending dates) for each task is shown. As it can bee seen, a total of 434 hours have been estimated from March to September.

In order to gather as most information about the work done, and as accurate as possible, an Excel sheet has been designed, where a tracking of every worked hour will be captured. From all that information, some graphics are later generated. These offer a fast overview of the work done, visually representing the tasks' fulfillment level, and enabling a fast mechanism to spot inconsistencies.

NOTE: A review over the final number of hours worked in comparison to the estimations can be found in chapter 11.

2.6.2 GANTT diagram

In figure 2.5 the Gantt diagram for the project is depicted.

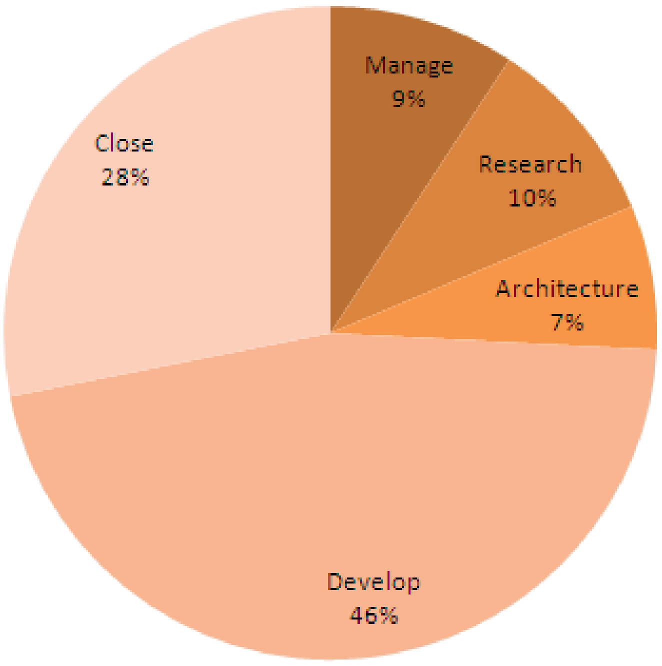


Figure 2.3: Load percentage of high-level tasks.

2.7 Quality plan

So as to guarantee that a minimum level of quality is ensured, a checklist containing different features of the system has been created (see appendix A). In the control and monitoring process during the project, this list will be progressively completed. In addition to this, both a risk management and a contingency plan are also defined.

2.7.1 Key Performance Indicators

To measure the level of fulfillment of the goals on a par with the quality of the work done, some key performance indicators have been defined. These can also be used as a mechanism to measure the success level of the project.

Quantifiable indicators:

- Schedule keep-up: the desired result for the project is to finish it within the foreseen schedule, meeting all the deadlines.
- Hours worked: the target number of total hours invested on the project is estimated on the project's plan, but a deviation of up to a 10% will be accepted as valid.
- Lines of code: even if it has nothing to do with the final quality of the developed

system, the increasing number of lines from week to week can be a good indicator that work is being done.

- Efficiency of the framework: the application developed must not have any freeze-up or halt that could taint the tests' readings with unwanted lags.

Qualitative indicators:

- Clearness of the code: the applications developed should be written as clearly as possible, being properly commented and using common sense variable names and structures.
- Satisfaction with the results obtained: clear conclusions should be able to be taken from the final outcome of the tests.
- Usefulness of the research: no initial questions raised at the beginning of the project remain unanswered.
- Experience gained: whatever the outcome of the project is, the experience gained in all the different areas touched should be positively valued.

Leading indicators:

- Director's approval: the project cannot be taken as finished until the director approves it. At the same time, the director's positive opinion should be taken as indicator of success.
- Peer feedback: the personal opinion of peers and friends could be indicator of failure or success. It should be borne in mind that those opinions could be biased.

Lagging indicators:

- Final project mark: the grade obtained after presenting and defending the project will be a definite *post hoc* indicator of success (or failure). The target mark is 10 out of 10.
- Impact of the study in the scientific field: should somebody carry on with new lines of this project, or who knows, end someday contributing in its research field, it could be interpreted as the height of success.

2.7.2 Risk management plan

Even if the size and requirements of this project are rather small, it suffers from threats as any other project. Thus, a risk management plan is required so as to minimize the impact of otherwise unexpected issues that could threaten the correct fulfillment of this work in time.

One of the primary risks is the malfunctioning of the hardware in use. On the one side, the computer where the majority of the work is done means that it is an essential asset. On the other hand, there also are mobile phones, Arduino boards and the rest of hardware equipment involved in the development of the project. Albeit not of vital value, those artifacts are tools with which the project is meant to be done, and if one failed, its replacement could take several days if not weeks. Therefore, a failure of one such device could threaten the fulfillment of the whole project.

However, hardware breakage is not the only kind of risk that should be taken into account. Indeed, there is another threat of equal importance: the requirements and objectives' updates. If during the development of the project a new requirement arose, it could be fatal for the completeness of the project. In order to minimize this from happening, the initial definition of goals, requirements and scope must be defined as best as possible. The analysis phase of the project will also be decisive. Of course, the later a new requirement appears, the greater the impact it will have.

Another minor but potential risk is that the APIs from Android change while developing the project. New API levels do not necessarily bring backwards incompatibility, but they can. This could mean that the code that worked until the update appeared may no longer do it. If such thing happened, two alternatives could be chosen: to ignore the update and continue programming in an ”outdated” version, or apply the update and upgrade the system's code to comply with the new API. Still, even after updating a device's Android version, it should be possible to restore the device to a previous release (in theory), but this could be a mess to accomplish.

Below are summed the potential risks (the items are not ordered by any priority):

- Hardware breakage (Arduinos, mobile phones, computer, wires and the like)
- Software updates (the need to reinstall a program) and end of licenses
- Information loss (both in paper and digital forms)
- Requirements update
- Unexpected bugs or incompatibilities
- Changes in Android's API

2.7.3 Contingency plan

Should an Arduino board or DF-Bluetooth antenna malfunction, it could be possible to borrow an extra one or two from the Egokituz laboratory's stock. However, if a component such a smart-phone showed some damage or malfunction, there would be no replacement.

If any of the aforementioned problems happened, the scope of the project could suffer some restrictive changes. In addition to this, updates on the planning ought to be done, for a failure to do so could bring up unexpected workloads to meet the deadlines.

Regardless of the risk or problem, an extraordinary meeting with the main stakeholders should be called whenever the scope, schedule or expected fulfillment level of the project is at risk of being changed.

2.8 Project stakeholders

There are four kinds of stakeholders according to their level of interest and involvement in this project. The main stakeholder—me, Xabier Gardeazabal—, whose interest is utterly justified for being the author of the project. The feasibility of the whole project depends of him, and all responsibility rests in no other person. It is his task to perform all the work by himself, although guidance from peers or the director will gratefully be accepted.

The next group of stakeholders is formed by Borja Gamecho and any of the members of the Egokituz laboratory who may be interested in the final outcome of this research. Borja Gamecho's interest is justified because the product of this project could be of some help for his PhD Thesis. Although he has no commitment regarding the project, he may provide help if necessary.

Another group of interested people is formed by the director of the project—German Rigau—. His role is that of a guide for the student at the head of the project. The director's commitment is twofold: solving doubts regarding the software engineering processes of the project, and the standing for having a student under his supervision successfully complete his project. Once the project memory is presented, he is required to present a report, which should later be taken into account for the final mark of the project by the evaluation court.

As of the current official regulations for the Bachelor Thesis (Article 4, subsec. 2):

> 4.2. El trabajo de dirección consistirá en exponer a cada estudiante las características del trabajo, de orientarlo en su desarrollo y de velar por el cumplimiento de los objetivos fijados, así como de realizar el seguimiento y elaborar un informe escrito previo a la defensa del que se dará traslado a la estudiante o al estudiante y al tribunal evaluador.

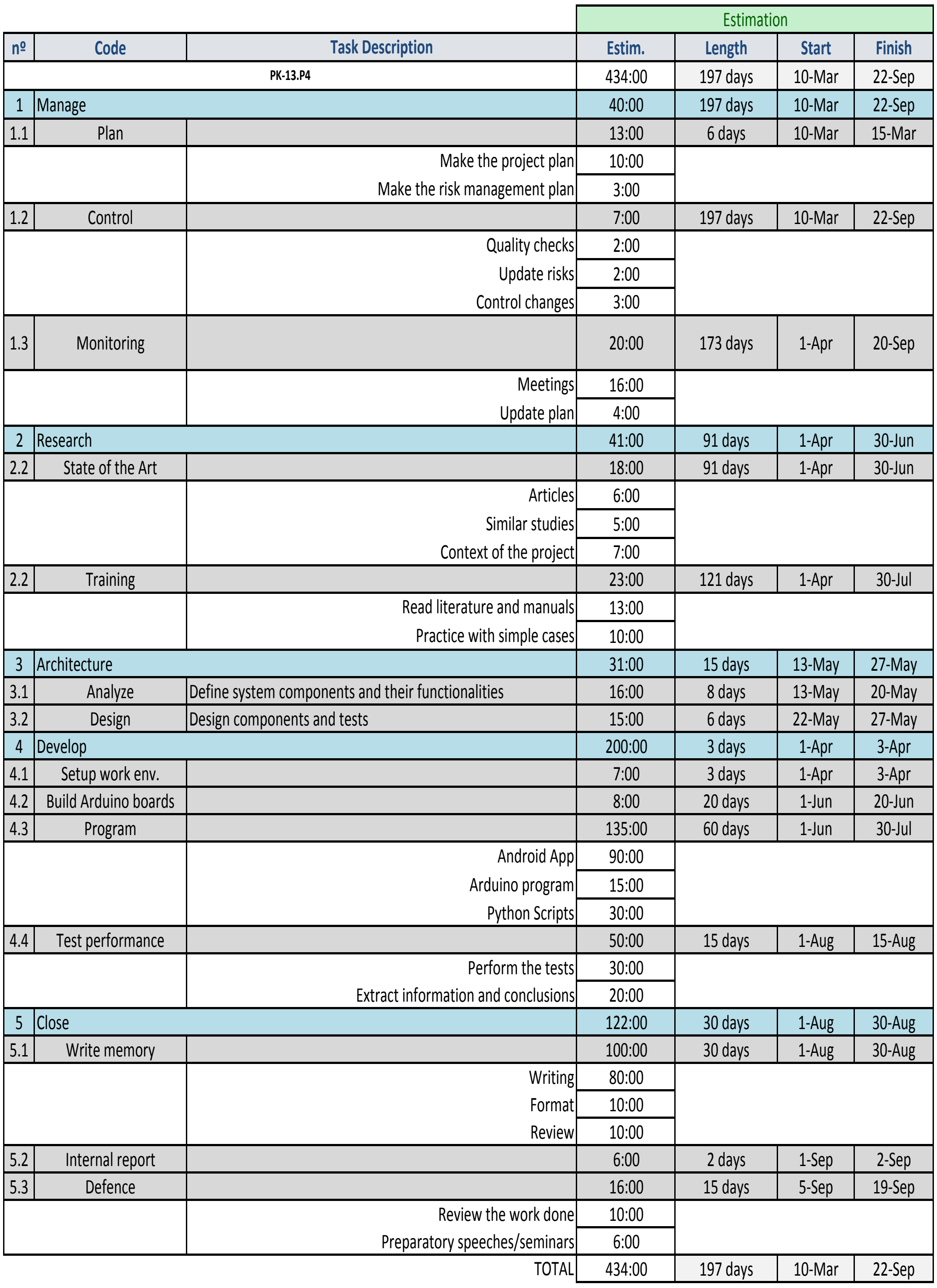
The last group of stakeholders are the members of the evaluation committee, which are known professors of the Faculty of Informatics at the same time. Their role is to test and assess the project once it is finished and presented to them. They must also be present in the defense of the project. The members of this court are internally selected by the committee of the Faculty of Informatics of San Sebastian, and have to be organized according to the same specialization of the student's thesis. In this case, since the specialization is Software Engineering, the professors must be part of the *Languages and Information Systems* department. It should be taken into account that until the defense process of the project this stakeholder group need not have any relationship with the other stakeholders.

On a side note, there is a general group of people who have no direct relationship with the project, but may find it appealing nonetheless. In addition to this, since the defense of the project is open to the public, anybody could be present at the time it takes place. Furthermore, somebody could be carrying on studies or research on a topic related to this work, and may as well be interested in getting ahold of the project's different resources.

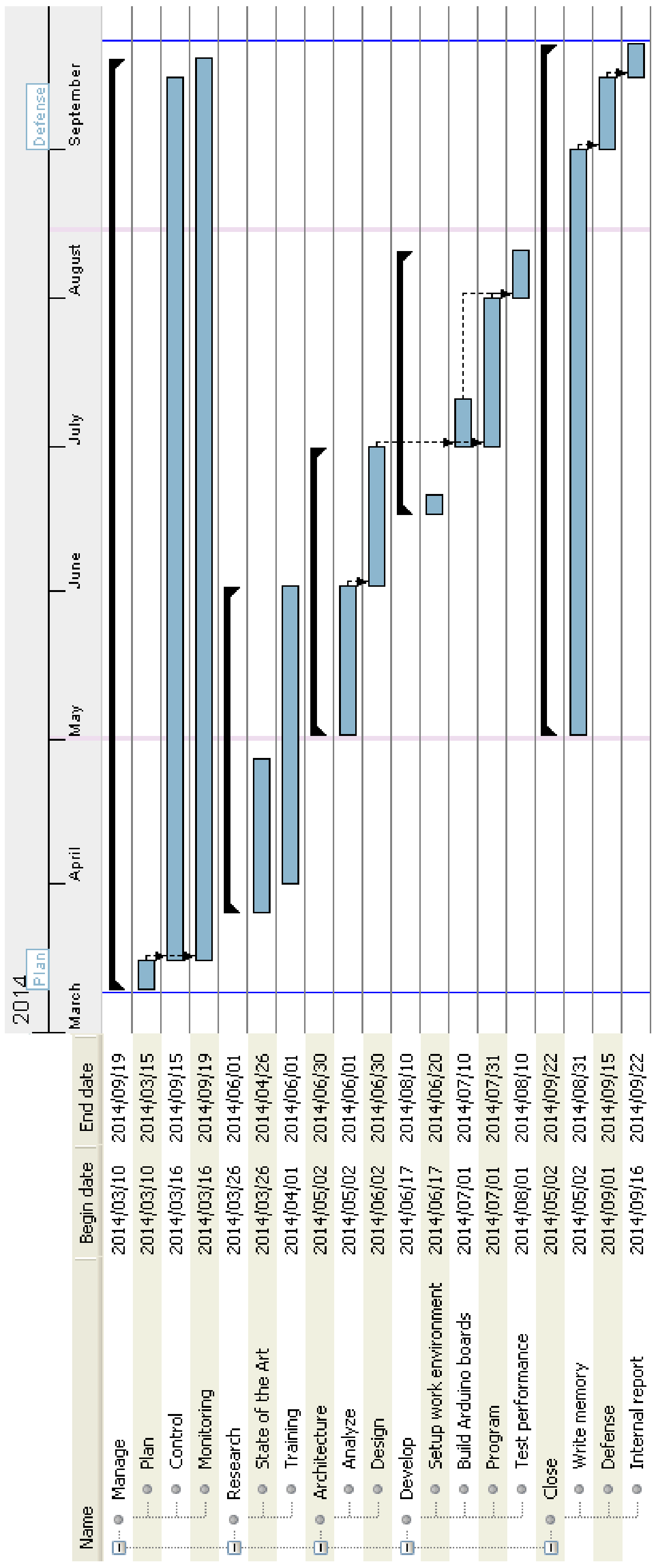
Communication plan

Three possible channels have been identified to communicate between the stakeholders: e-mail, phone-calls, and in person. Taking into account that I will spend most of the working time in the Faculty's facility, in-person communication would be the most efficient, but an e-mail or call way be equally useful for little-talk. The contact details of each main stakeholder have already been exchanged, and are not collected in this document for privacy reasons.

Figure 2.4: Initial estimation table for the different tasks.



			Estimation			
nº	Code	Task Description	Estim.	Length	Start	Finish
		PK-13.P4	434:00	197 days	10-Mar	22-Sep
1	Manage		40:00	197 days	10-Mar	22-Sep
1.1	Plan		13:00	6 days	10-Mar	15-Mar
		Make the project plan	10:00			
		Make the risk management plan	3:00			
1.2	Control		7:00	197 days	10-Mar	22-Sep
		Quality checks	2:00			
		Update risks	2:00			
		Control changes	3:00			
1.3	Monitoring		20:00	173 days	1-Apr	20-Sep
		Meetings	16:00			
		Update plan	4:00			
2	Research		41:00	91 days	1-Apr	30-Jun
2.2	State of the Art		18:00	91 days	1-Apr	30-Jun
		Articles	6:00			
		Similar studies	5:00			
		Context of the project	7:00			
2.2	Training		23:00	121 days	1-Apr	30-Jul
		Read literature and manuals	13:00			
		Practice with simple cases	10:00			
3	Architecture		31:00	15 days	13-May	27-May
3.1	Analyze	Define system components and their functionalities	16:00	8 days	13-May	20-May
3.2	Design	Design components and tests	15:00	6 days	22-May	27-May
4	Develop		200:00	3 days	1-Apr	3-Apr
4.1	Setup work env.		7:00	3 days	1-Apr	3-Apr
4.2	Build Arduino boards		8:00	20 days	1-Jun	20-Jun
4.3	Program		135:00	60 days	1-Jun	30-Jul
		Android App	90:00			
		Arduino program	15:00			
		Python Scripts	30:00			
4.4	Test performance		50:00	15 days	1-Aug	15-Aug
		Perform the tests	30:00			
		Extract information and conclusions	20:00			
5	Close		122:00	30 days	1-Aug	30-Aug
5.1	Write memory		100:00	30 days	1-Aug	30-Aug
		Writing	80:00			
		Format	10:00			
		Review	10:00			
5.2	Internal report		6:00	2 days	1-Sep	2-Sep
5.3	Defence		16:00	15 days	5-Sep	19-Sep
		Review the work done	10:00			
		Preparatory speeches/seminars	6:00			
		TOTAL	434:00	197 days	10-Mar	22-Sep



Name	Begin date	End date
Manage	2014/03/10	2014/09/19
Plan	2014/03/10	2014/03/15
Control	2014/03/16	2014/09/15
Monitoring	2014/03/16	2014/09/19
Research	2014/03/26	2014/06/01
State of the Art	2014/03/26	2014/04/26
Training	2014/04/01	2014/06/01
Architecture	2014/05/02	2014/06/30
Analyze	2014/05/02	2014/06/01
Design	2014/06/02	2014/06/30
Develop	2014/06/17	2014/08/10
Setup work environment	2014/06/17	2014/06/20
Build Arduino boards	2014/07/01	2014/07/10
Program	2014/07/01	2014/07/31
Test performance	2014/08/01	2014/08/10
Close	2014/05/02	2014/09/22
Write memory	2014/05/02	2014/08/31
Defense	2014/09/01	2014/09/15
Internal report	2014/09/16	2014/09/22

Figure 2.5: Gantt diagram of the project.

3. CHAPTER

State of the art

In this chapter the current state of the art at the beginning of the project is explained. In the following sections a brief overview on similar researches or studies can be found. An analysis on the available technologies used alternatives is made as well. But first, a review on the original context that this project comes from is done.

3.1 Background of the project

3.1.1 Pervasive computing

We call pervasive or ubiquitous to those technologies we tend to use without actively thinking about the tool. By focusing on the task, the technology is almost invisible to the user. Pervasive technologies are often wireless, mobile, and networked. As Lyytinen and Yoo said back in the day, ”computers will be embedded in our natural movements and interactions with our environments—both physical and social” [Lyytinen and Yoo, 2002].

Pervasive computing combines current network technologies with wireless computing, voice recognition, Internet capability and artificial intelligence, and its goal is to create an environment where the connectivity of the devices is embedded in an unobtrusive way.

One of the core concepts of pervasive systems is its grounding, assembled by small, inexpensive and robust networked processing devices, distributed at all scales throughout everyday life.

3.1.2 Context-Aware Computing

Context-aware computing is a sub-field of pervasive computing. As Robles and Kim (2010) says, it refers to a general class of mobile systems that can ”sense” their physical environment, and adapt their behavior accordingly [Robles and Kim, 2010].

The reason why context-aware computing is relevant for this project comes from the collaborative work done with Borja Gamecho and his PhD Thesis, as explained in the motivation of this project (section 1.2).

In [Gamecho et al., 2013], the benefits of using many different sensors and combining them to obtain higher-level context information are explained. However, it is also noted the problems that sensor heterogeneity brings with it, such as performance drops. The findings of this project may help to clarify the convenience of Bluetooth for context-awareness.

3.1.3 Personal Area Networks

A Personal Area Network (PAN) is formed by interconnected devices around a person or within their range. These networks are typically composed of regular personal devices such as laptops or smartphones on a par with other static items like printers, while connections between one another happen transparently to the user. These types of networks are usually able to connect to the Internet—and provide access to the components of the PAN—and other networks, even without wires.

Wireless Personal Area Networks

If a PAN is a network for interconnecting devices centered on an individual person's workspace, a Wireless Personal Area Network (WPAN) is the same, with the additional feature that the connections are made wirelessly. The devices that fall under this category of PANs are smaller than usual, and therefore only work in a short-range—somewhere around 15 meters—. They were envisioned to provide high-quality real-time video and audio distribution, as well as file exchange among storage mechanisms, and are quickly becoming a replacement for home systems that used to work with cables.

WPANs are based on the IEEE 802.15 standard, which should not be mistaken with the IEEE 802.11, which corresponds to the Wireless Local Area Networks (WLANs), better known as WI-FI technology. Indeed, the difference between the two is that while PANs and WPANs tend to be centered around one person, a Local Area Network—wireless or not—is usually serving multiple users.

This does not mean that a WI-FI based network cannot be part of a PAN, but this is less common as it takes more resources than its counterpart technologies, and reduced size and increased longevity is preferable over power.

3.1.4 Body Area Networks

A Body Area Network (BAN) is a type of personal area network made up from wearable computer devices. These communicate with other nearby computers or devices and exchange digital information by using the natural electrical conductivity of the human body. BANs are based on the IEEE 802.15.6 standard for transmission via the capacitive near field of human skin, but can be made wirelessly as well.

3.1.5 Sensor Networks

Sensor networks consist of a large number of nodes spread across a geographical or urban area. The nodes can be static or mobile, and key requirements for sensor networks operating in challenging environments include low cost, low power, and multi-functionality of the devices.

There are many scenarios where sensor networks are used. Sometimes, a body area network can be made by dressing a person with sensors to obtain biometric data. In other cases, a sensor network can be formed by placing beacons in cars, and be used for collecting information about their surrounding environment.

Sensor networks are usually prone to show problems, as they have to confront many changing and uncontrollable variables. It is therefore quite challenging to identify and diagnose their accuracy and reliability. Related to this, Edith and Gunningberg (2013) defined a metric known as the Quality of Information (QoI) that measures information attributes such as precision, timeliness, completeness, and relevance of data ultimately delivered to users in sensor networks. According to Edith and Gunningberg, it is a challenge to provide the required QoI in mobile sensor networks given the large scale and complexity of the networks with heterogeneous mobile and sensing devices [Edith and Gunningberg, 2013].

3.2 Other studies on the subject matter

Since the inception and adoption of Bluetooth as a technology for Personal Area Networks, many studies and research papers have been published on topics related to it.

Below are presented a few of the most relevant publications that have an special interest to this project, as they all touch topics related to what this project is all about.

3.2.1 Power consumption and performance of Bluetooth

In the article titled ”Power characterization of a bluetooth-based wireless node for ubiquitous computing” [Cano et al., 2006], a research of the consumption of a particular Bluetooth module—the Mitsumi WML C11 class 1—is made. Now, the module used in that study has nothing to do with the one used in this project, albeit the discoveries they made may be relevant nonetheless. They concluded that the most power-saving option was to power-off the devices when they were not in use. Of course, this brought a trade-off in performance, since the access time increases when booting the devices. This study only focused on the static slave devices that were waiting for connection request, and the they did not measure the consumption of the inquirer or master device (a smartphone in this project). Therefore, the present project might complement Cano et al.'s study.

Additionally, in the article titled ”Evaluation of the trade-off between power consumption and performance in Bluetooth based systems” [Cano et al., 2007], a thorough research in the subject matter is made. Their experiments focused on evaluating the power-delay and power-throughput trade-off experimented by UDP and TCP traffic, respectively. They also performed a sensitivity analysis to evaluate how distance and packet type affected power consumption, throughput and delay.

Among the different experiments Cano et al. performed, the one in which they evaluate the impact on packet delay of varying the ACL[1] packet type and the distance between devices is of uttermost interest. They observed that regardless of the packet type, Bluetooth provides a stable packet delay up to 10 meters, and that when surpassing the 10m distance it still manages to perform quite well. From 15 meters on, performance degradation starts to become noticeable.

The other experiments and conclusions gathered in [Cano et al., 2007] are not relevant to this project, since they focus on technical aspects of the Bluetooth protocol that are far beyond the scope of this project. Nevertheless, apart from the actual content, the way they portrayed their experiments' data by the use of graphic figures was of high interest and source of inspiration.

[1]The normal type of radio link used for general data packets in Bluetooth.

3.2.2 Discovery time

In the paper "Bluetooth Discovery Time with multiple Inquirers" [Peterson et al., 2006], an interesting problem on the discovery process is presented. The authors claimed that while a device is performing a discovery, the presence of a second inquiring device can significantly delay, and even preclude, the discovery of a discoverable node. From their experiments they concluded that the presence of a second inquirer may in fact prevent a inquiring device from discovering a scanning node until the second node leaves the inquiry substate.

Apart from that, in [Chakraborty et al., 2010] a thorough analysis of Bluetooth's device discovery protocol is made. In this paper, apart from explaining the Bluetooth protocol very well, the authors demonstrate that discovery delay increases with the number of devices in the area in a logarithmic way. Additionally, the authors even propose a method to reduce discovery time, but it involves delving into the Bluetooth stack implementation, so it goes beyond this project's scope.

3.2.3 Service Discovery Protocols

Even if this project does not involve any *service discovery*, it often is an essential part of PAN-like applications. Service Discovery Protocols (SDP) allow the automatic detection of services offered by any devices connected to a network. Renowned protocols such as DHCP (Dynamic Host Configuration Protocol) or UPnP (Universal Plug and Play) are examples of Service Discovery Protocols.

In the article "Service Discovery in Pervasive Computing Environments" [Zhu et al., 2005] there's an interesting table comparing Bluetooth's SDP with other protocols—Jini, Salutation and UPnP, among others—that used to be used in ubiquitous or pervasive computing. According to the author's research, Bluetooth's SDP is quite similar to its counterparts, and has nothing to envy.

Coupled with this, in [Ververidis and Polyzos, 2008] a general survey on researches done over the topics of service advertising, discovery, and selection for mobile *ad hoc* networks is made. In this article, a thorough insight into most of the service discovery protocols that Zhu et al. review in their article is made. A similar but much more condensed study was made in [Helal, 2002] as well.

3.2.4 Power and batteries

The full degree of freedom in mobile systems heavily depends on the energy provided by the smartphone's battery. In [Perrucci et al., 2011], a comparison between the components that drain the battery of a smartphone is made. According to this study, these are the components that drain more power:

1. Downloading data using WLAN (WI-FI)
2. Sending an SMS
3. Making a voice call
4. Playing something through the loudspeakers
5. Displaying high brightness

Quite surprisingly, Bluetooth is not among the top five battery consumers.

Like many other technologies, batteries have reached a physical limit where our resources or scientific knowledge cannot help us create better batteries, and their performance curve has suffered a steady flattening lately. Something similar happened to the microprocessors, where only the transistor count keeps a fast growth, according to [Sutter, 2005].

However, unlike microprocessors, battery's price has not dropped to the same extent. As Starner (2003) said, mobile phone companies sell more batteries than phones to consumers, and they do so while trying to protect their design and utilities with patents to keep third-party vendors from competing too heavily [Starner, 2003]. Therefore, the development of better batteries is slow.

3.2.5 Interferences in wireless systems

There are many technologies that make use of the 2.4 GHz ISM frequency band apart from Bluetooth. Therefore, it is natural for interferences to appear when two or more of such protocols operate on the same vicinity. Among others, WI-FI presents itself as the protocol most likely to cause interferences to Bluetooth. In [Fainberg and Goodman, 2001], an analysis of the interference between WI-FI and Bluetooth systems is made. Additionally, in [Weng et al., 2014] and [Golmie and Mouveaux, 2001] more general studies around interferences on the 2.4 GHz spectrum are made.

Interferences, however, are not only generated by colliding protocols over the same frequency band. Other variables such as the weather or materials in the area can considerably impact the efficiency or reliability of wireless protocols like Bluetooth. In the article

"Factors Causing Uncertainties in Outdoor Wireless Wearable Communications" [Fong et al., 2003], the impact of the rain over wireless networks is presented.

3.3 Alternatives to the Bluetooth protocol

The reason why Bluetooth has been chosen over other alternatives is because it is a well-established and widespread technology. However, the field of Personal Area Networks is increasingly developing new technologies and alternatives to carry communications that are worth checking out. Below are explained some of the most relevant protocols that are similar to or compete with Bluetooth to a certain extent.

3.3.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE)—marketed as Bluetooth *Smart*—is the successor of the ”Classic Bluetooth”, and it intends to provide a considerably reduced power consumption and at a lower cost, while trying to maintain similar communication ranges.

Although it was originally named as *Wibree* and developed by *Nokia* in 2006, it was later merged into the main Bluetooth standard in 2010, along with the adoption of the Bluetooth Core Specification Version 4.0. It is due to this divergence in its origin that BLE is not backwards compatible with the ”Classic” Bluetooth. Still, the specification allows for either or both versions to coexist and be implemented in the same device. Additionally, the two versions can be run in parallel in the same physical area, since even if BLE operates in the same spectrum range as Classic Bluetooth—that is, 2.5MHz—, it uses a different set of channels. Instead of the Classic Bluetooth’s 79 channels of 1-MHz each, Bluetooth Smart has 40 2-MHz channels.

Although nominally the Bluetooth 4.0 specification provides lower power consumption with higher baud rates than its predecessor versions, it comes at the cost of reducing the maximum throughput achievable.

3.3.2 ANT

Advanced Network Tools or *ANT* is a proprietary open access multicast wireless sensor network protocol, developed by Dynastream Innovations Inc.—a subsidiary of Garmin, a major GPS equipment manufacturer—. ANT is characterized by its low computational overhead and low to medium efficiency, resulting in low power consumption by the devices supporting the protocol, and enabling low-power wireless embedded devices operate on a single coin-cell battery for a long time.

ANT has been primarily targeted at the sports sector, particularly fitness and cycling performance monitoring. The transceivers are embedded in equipment such as heart rate belts, watches, cycle power, and cadence meters, and distance and speed monitors to form Wireless Personal Area Networks (WPANs) monitoring a user's performance.

ANT+ or (ANT Plus) on the other hand, is an interoperability function that can be added to the base ANT protocol.

3.3.3 ZigBee

ZigBee is a simpler, slower, lower-power, lower-cost specification similar to Bluetooth, and it was created by Philips. It is supported by a mix of companies that are targeting the consumer and industrial markets. It may be a better fit with games, consumer electronic equipment, and home-automation applications than Bluetooth. Short-range industrial telemetry and remote control are other of its target applications.

Previously called *RF-Lite*, ZigBee is similar to Bluetooth, since it also uses the 2.4-GHz band with frequency-hopping spread-spectrum with 25 hops spaced every 4 MHz. The basic data rate is 250 kbits/s, but a slower 28-kbit rate is useful for extended range and greater reliability. With a 20-dBm power level, ZigBee can achieve a range of up to 134 meters at 28 kbits/s. It additionally allows to connect up to 254 nodes in a network.

3.3.4 Comparison

In table 3.1 a comparison between ANT, ZigBee, Bluetooth and Bluetooth Low Energy is summarized.

Market name	**ANT**	**ZigBee**	**Bluetooth**	**Bluetooth LE**
Standard	Proprietary	IEEE802.15.4	IEEE802.15.1	IEEE802.15.1
Frequency band	2.4 GHz	2.4 GHz	2.4 GHz	2.4 GHz
Data Rate	<60kbps, 1Mbps	20-250 kbps	1-3Mbps	200kbps
Latency	not specified	<5 ms	100 ms	6 ms
Range (meters)	1-30m	75m	10-100m	50m
Power	20mW	60mW	120mW	10mW
Lifetime	weeks	days, weeks	days	weeks
Max. network size (nodes)	2^32	2^64	7	2^64
Network architecture	P2P, start, tree, mesh	Multiple star, mesh	P2P, piconet	P2P, piconet

Table 3.1: Comparison table between Bluetooth, Ant and ZigBee.

3.4 Alternative Wireless Technologies

Although this project focuses solely on Bluetooth, some other protocols that also operate in the 2.4 GHz have been presented. However, there are many other wireless technologies in the field of Personal Area Networks. Below, some alternatives to Bluetooth are explained. It should be noted that these technologies are not nearly as commercialized as Bluetooth, but could become quite popular in the near future.

3.4.1 Induction Wireless

Instead of radio, this technique uses magnetic induction for close-range communications. In radio, the signal is formed by both electric and magnetic fields. In Induction Wireless, the signal is only formed up by the magnetic field. The transmitter is a radiating coil that's more like the primary winding of a transformer than an antenna.

A typical Induction Wireless unit transmits up to 204.8-kbit/s data rates via GMSK modulation on 11.5 MHz. Its key benefits are extremely low power consumption and low cost. On the other hand, the fact that this system has a 3 meter range may be a drawback. However, this can be seen as an advantage, since many more people in a given space can operate such devices without causing interference to each other.

Induction Wireless was invented and patented by Aura Communications, although the company was later acquired by *FreeLink*. This technology has been widely used in the medical industry for years, but it has not been until recently that it is being used for streaming wireless voice and audio in personal electronics—such as headsets or headphones. It seems as a promising technology that could get to play a major role in the field of PANs. As *Aura Communications* state:

> The company's technology, which goes under the trademarked name LibertyLink, is a near-field magnetic wireless communication scheme. Aura's chips create a static B field (the magnetic component of RF), killing the E field (the electrical component) so it does not radiate outward. The result is a two-meter static bubble that envelops the user and enables voice and audio transmission within it. The range is so exact that moving the headset one half inch across the bubble boundary can cause the signal to disappear.

3.4.2 IR Wireless

IR wireless is the use of wireless technology in devices or systems that convey data through infrared (IR) radiation—an electromagnetic energy at a wavelength or wavelengths somewhat longer than those of red light—. In short, IR is used for short and medium range communications and control, and it provides a physically secure data transfer with very low bit error rate.

During the early 1990s the IrDA[2] and its standard appeared thanks to Hewlett-Packard, and it is still supported by Agilent Technologies. IrDA initially provided a 115.2-kbit/s data rate over a range of up to 1 m. A 4-Mbit/s version was soon developed and has been widely incorporated in laptops and PDAs for printer connections and short-range PANs. A 16-Mbit/s version was available too.

Nowadays, infrared technology is used in local networks, and it can be found in three different forms:

- IrDA-SIR (slow speed) infrared supporting data rates up to 115 Kbps
- IrDA-MIR (medium speed) infrared supporting data rates up to 1.15 Mbps
- IrDA-FIR (fast speed) infrared supporting data rates up to 4 Mbps

However, these speeds cannot compensate for infrared's limited range and its need for a line-of-sight connection[3]. Of course, every other technology based on radio or electromagnetic fields do not need any line-of-sight, as these can even go through walls.

A more recent IR development is IrGate—produced by *Infra-Com Technologies*—, which uses arrays of high-powered IR LEDs to emit coded baseband IR in all directions. Then it relies on an array of photodetectors and super-sensitive receivers to pick up the diffused IR within the networking space. Thus, the line-of-sight problem is mitigated, and a data rate of up to 10 Mbits/s is possible.

3.4.3 Ultra Wideband

Ultra Wideband (also known as UWB or as ”digital pulse wireless”) is a wireless technology for transmitting large amounts of digital data over a wide spectrum of frequency bands with very low power for a short distance. It has been in use by military and government applications for quite a long time, and its main use—until very recently—have been

[2]IrDA (Infrared Data Association) is an industry-sponsored organization set up in 1993 to create international standards for the hardware and software used in infrared communication links.

[3]A line-of-sight connection means that the devices carrying a connection must have unobstructed visual sight of each other throughout the whole connection process.

short-range, high-resolution radar and imaging systems that penetrate walls. Nowadays, however, it has been proven to be useful for short-range LANs or PANs. UWB technology emerges as a promising physical layer candidate for WPANs, because it offers high-speed communications over short ranges, with lower cost and higher power efficiency than other technologies with a similar purpose.

In addition to its suitability for personal area networks, as Yang and Giannakis (2004) state, Ultra Wideband is also very appropriate for sensor networks:

> High data-rate UWB communication systems are well motivated for gathering and disseminating or exchanging a vast quantity of sensory data in a timely manner. Typically, energy is more limited in sensor networks than in WPANs because of the nature of the sensing devices and the difficulty in recharging their batteries. Studies have shown that current commercial Bluetooth devices are less suitable for sensor network applications because of their energy requirements and higher expected cost. In addition, exploiting the precise localization capability of UWB promises wireless sensor networks with improved positioning accuracy. This is especially useful when GPSs are not available, e.g., due to obstruction. [Yang and Giannakis, 2004]

4. CHAPTER

Architecture and technological choice

This chapter tries to explain the general architecture of the system developed in this project, and a technical review on the technologies used is also made.

4.1 Testbed architecture

In order to solve the initial problem of not knowing how does the performance of a smartphone behave with many simultaneous Bluetooth communications, a framework that provides the tools necessary to form a testbed is required. This framework should be formed by two main components: a set of smartphones and a pool of Bluetooth sensors.

While they carry on communications with each other during a testing process, the phones should perform some measurements, from which different metrics would later be extracted.

Finally, from a thorough analysis of those metrics some insightful results or conclusions should be achieved. If the whole system works flawlessly, the results obtained could be used as a benchmark or reference for future projects involving wireless Bluetooth communications. Figure 4.1 shows an abstracted conceptualization of the architecture of the testbed.

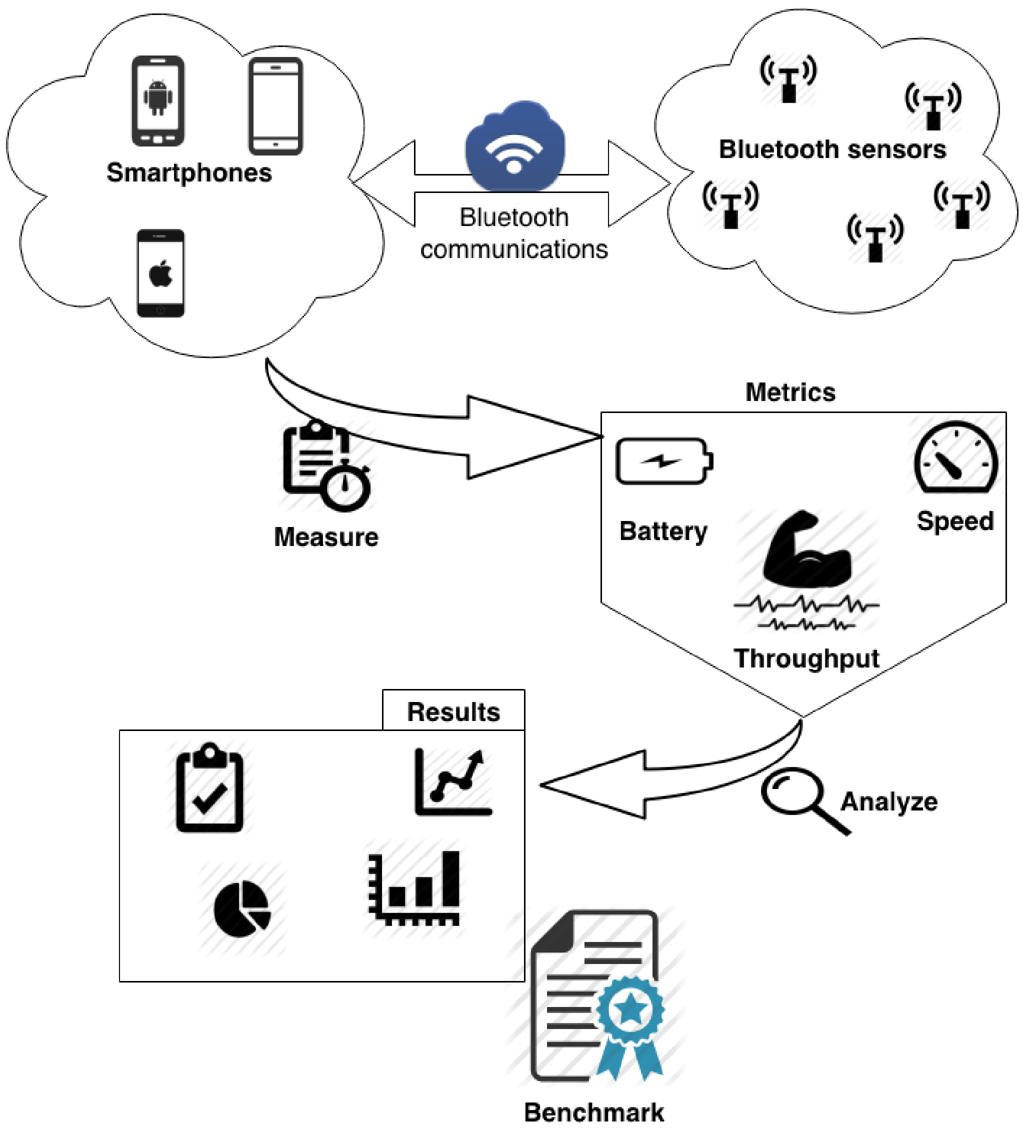


Figure 4.1: Testbed architecture.

4.2 Technologies used

4.2.1 Bluetooth

Bluetooth—the main technology used as the base of this project—is a standard wireless technology for short distance data exchanging. It has been among us for quite a long time, and since its invention in 1994 by Ericsson—a telecommunications vendor and cell phone manufacturer— it has undergone many revisions and modifications. Yet, its core features remain unchanged: a low cost wireless replacement for cables on phones, headsets and other devices.

Additionally, its low power consumption and fast connection setup make it perfect for small sensor devices and even sensor networks as well. Not surprisingly, it has been the main protocol used in the ”Personal Area Networks” for many years. Again, its simple yet useful features such as fast device discovery in the near surrounding or high rate data transfers make this protocol very suitable for such environments.

Although during the first decade since its invention it underwent a steady growth, it has not been until recently—with the introduction of Bluetooth Smart—that it has regained interest. Still, the work carried on this project focuses solely on the classic version of Bluetooth, and does not take the new and revised version into account. The reason for this is mainly owing to the original idea and motivation of this work, explained in the first two chapters of this document.

History

Although originally envisioned just as a cable-replacement technology by Ericsson in 1994, embedded Bluetooth capability is becoming widespread in numerous types of devices. These include intelligent units (PDAs, cell phones, PCs), data peripherals (mice, keyboards, joysticks, cameras, digital pens, printers, LAN access points), audio peripherals (headsets, speakers, stereo receivers), and embedded applications (automobile power locks, grocery store updates, industrial systems, MIDI musical instruments) among others [McDermott-Wells, 2004].

With time, Bluetooth has become a usual technology term of everyday life, and there probably are a very few people who have not heard of it, since almost all mobile-phones bring it inherently. Ten years ago, shortly before the second revised and improved version was adopted, this protocol did not take such a big role if compared to the many other standards existing at the time. Still, its growth was evident, and as Shepherd (2001) said, even if only 10% of mobile phones and computers were to incorporate Bluetooth chips, then the technology would achieve the critical mass necessary for chip prices to fall below $5 and for the standard to become firmly established in the market-place [Shepherd, 2001]. As time has shown, Bluetooth prevails.

In table 4.1 a comparison between the main different versions of Bluetooth throughout history is shown.

Specification version	**v 1.0**	**v1.2**	**v2.0+EDR**	**v2.1+EDR**	**v3.0 HS**	**v4.0 (BLE)**
Adoption date	2002	2003	2004	august 2007	2009	2010
Backwards compatible	—	yes	yes	yes	yes	no
Bit rate	721 kbit/s	721 kbit/s	1-2.1 Mbit/s	1-3 Mbit/s	24 Mbit/s	200kbit/s
Range	unknown	unknown	10m	10-100 m	30 m	10-50 m
Setup time	unknown	unknown	unknown	<6 s	unknown	<3 s

Table 4.1: Table comparing versions of Bluetooth in history.

Technical specifications

Bluetooth uses short-wavelength UHF radio waves in the ISM[1] band from 2.4 to 2.485 GHz. Thanks to the *frequency-hopping spread spectrum* technology[2] , Bluetooth is able to divide part of the ISM band into a total of 79 different channels—of 1 MHz each—. The data is divided into packets, and each packet goes through one channel.

Bluetooth is based on packet-switching, which means that the transmitted data is grouped into blocks of a predefined size. The protocol is defined as a layer protocol architecture, and it was conceived taking many other protocols within. There is not a single and unique implementation of the protocol, and each different adaptation is known as ”stack” (i.e. a stack is a software piece that refers to an specific implementation of the Bluetooth protocol). Every Bluetooth stack must at least make use of the *Link Management Protocol* (LMP), the *Logical Link Control and Adaptation Protocol* (L2CAP) and the *Service Discovery Protocol* (SDP). Many Bluetooth applications use RFCOMM (*Radio Frequency Communications*) as well, due to its widespread support and publicly available API on most operating systems[3] . Additionally, any given Bluetooth stack can adopt other proto-

[1]ISM is a globally unlicensed (but not unregulated) Industrial, Scientific and Medical 2.4 GHz short-range radio frequency band

[2]The repeated switching of frequencies during radio transmission to minimize the unauthorized interception or jamming of a particular frequency band slot

[3]The applications that use a serial port to communicate with each other can be quickly ported to use RFCOMM

cols such as TCP/IP, UDP, WAP (*Wireless Application Protocol*), *Point-to-Point Protocol* (PPP) and the like. Knowing all this protocols has not been necessary for the proper development of the project, but are worth mentioning nonetheless.

Bluetooth profiles: the SPP protocol

In order to use Bluetooth, the device must be compatible with a subset of Bluetooth *profiles* necessary to use the desired services[4]. The *Serial Port Profile* (SPP) is one of such profiles, and defines the requirements for Bluetooth devices for setting up emulated serial-cable connections (serial RS232 specifically) using RFCOMM.

The scenario covered by this profile is setting up virtual serial ports on two devices and connecting them with Bluetooth, so that any application may run on either device using the virtual port as if there was a real serial-cable connecting the two devices (with RS232 control signaling).

This profile requires support for one-slot packets only. This means that this profile ensures that data rates up to 128 kbps can be used. Support for higher rates is optional. Refer to [bt2, 2001] for more details.

Discovery and pairing processes

A Bluetooth Personal Area Network is also known as a *piconet*, a network with a master-slave structure where one master may communicate with up to seven slaves. Bluetooth units that are within range of each other can set up *ad hoc* connections, and two or more devices that share a channel form a *piconet* [Haartsen, 1998]. Nonetheless, Bluetooth allows the union of two or more piconets by a peculiar mechanism[5]. If a device in a piconet—whether a master or a slave—decides to serve as a slave to the master of another piconet, then this device becomes the bridge between the two piconets, connecting both networks. When two or more piconets are connected, they form a *scatternet*, where communications between more than eight devices is possible.

In order to establish a connection, however, one must look for other devices in the area. Any device may perform an inquiry (i.e. a discovery process) to find other devices to connect to, and any device can be configured to respond to such inquiries. In order for a device to be listening for connection requests, however, it has to be set in ”discoverable

[4]A Bluetooth *profile* is a specification regarding a certain aspect of a Bluetooth-based wireless communication between two devices.

[5]The basic Bluetooth protocol does not support this, so the host software of each device needs to manage it

mode”. Otherwise, any nearby device performing a discovery would not find it, and could not ask it for a connection to be established. Of course, the device that is performing the discovery needs not be in discoverable mode. Any Bluetooth device in discoverable mode will transmit the device’s name, class, its service list and some other information on demand.

Once a device is connected to another one, it cannot simultaneously establish new connections with other devices, nor will it appear in inquiries from other discoveries.

Every device has a unique 48-bit physical address, commonly known as MAC. These addresses are generally not shown in inquiries, and instead a ”friendly” name or identifier of the device is used, which can be usually set by the user.

Once a device performing a discovery has found a connectable target nearby, it will be able to start the pairing process, often referred to as ”bonding”. During this process—which usually requires some kind of user interaction—the two devices involved establish a relationship by creating a shared secret known as a ”link key”.

It is a well known fact that the discovery is a heavyweight process, both for the antenna and processor. Therefore, it is the one action where Bluetooth consumes most resources. According to [Perrucci et al., 2011], this procedure can take a longer time according to the number of the devices in the range, but the power levels remains constant and is not an exponential function of the number of Bluetooth devices around. Therefore the energy consumption only depends on the time duration of the discovery.

4.2.2 Android

Needless to say, Android is a well known open-source operating system based on the Linux kernel, and although it is available for many platforms, it is best suited for smartphones. It is developed by Google, and it currently holds the lead in terms of number of devices with a version of this OS.

Developing a basic application in Android is rather simple. The main language used is Java, but other languages such as XML are important too. The easiest way to develop an Android application is to use the official Android Development Tools (or ADT) plugin for the Eclipse IDE. This plugin contains everything needed to develop and deploy an application to any Android device connected to the computer.

It is not the purpose of this project to explain the intricacies of the Android OS and its internal architecture since there is more than enough literature on the subject and doing so would be redundant. However, a little insight into the way threads and processes are handled in Android is not in excess.

Application components and threads

Application components are the essential building blocks of an Android app. There are four types: activities, services, content providers, and broadcast receivers [Andrdoid-Developers, 2014a]. When a component is started, if the application has no other components running, Android starts an independent Linux process with a single thread. Now, what is the actual difference between a process and a thread?

A process could be defined as a program unit of execution that has some allocated resources and memory just for itself. A thread, on the other hand, is a piece of a bigger program that can run in parallel with the main application process, but it runs ”within” the process nonetheless. Therefore, processes can contain more than a thread—which is actually the way most applications function—, and these threads share the same resources and memory.

In conclusion, creating multiple threads within a process is cheaper than starting separate processes for the same purpose. Be wise, though, since overloading a process with too many threads may result in a performance decay.

Bluetooth in Android

Android first introduced Bluetooth in the API level 5, containing most of its core functionalities, but it was not until API level 18 that it was renewed with more powerful methods and fixed bugs. The first API used to work on a par with the Blue-Z Bluetooth stack—the stack for Linux kernel-based family of operating systems—, but it was changed to the *BlueDroid* stack with the API level 18. This new and ongoing stack in Android is completely different to its predecessor, so it is not an expansion on Blue-Z. According to the technical information provided by the Android source code project:

> Android provides a default Bluetooth stack, BlueDroid, that is divided into two layers: The Bluetooth Embedded System (BTE), which implements the core Bluetooth functionality and the Bluetooth Application Layer (BTA), which communicates with Android framework applications.

It seems that this new stack is quite immature still, since there are several known issues with Android’s Bluetooth API which are yet not fixed as of the day of this writing.

Many developers have claimed that since the new Bluetooth stack upgraded in API level (Android 4.2), their applications have shown many new bugs, and blame the new stack's implementation.

Even if it is out of the scope of this project, it is worth noting that the functionalities for Bluetooth Low Energy were added in API level 18 (Android 4.3), so the devices running an older OS version should be updated to be able to make use of it.

The API levels mentioned in the previous paragraphs are integer values that uniquely identify the framework API revision provided by each version of the Android platform. It is not a thing worth worrying about, but it should be taken into account when developing applications for Android, since using resources that are only available in the last API level means no backwards compatibility.

4.2.3 Arduino

As its creators state, Arduino is an open-source physical computing platform based on a simple microcontroller board, and a development environment for writing software for the board.

The Arduino programming language is an implementation of *Wiring*, a similar physical computing platform which is based on the *Processing* multimedia programming environment. There is an open-source Arduino IDE for Winfows XP that can be downloaded for free, and it comes with many example programs. In addition to this, there is a growing community on the Internet with more than enough resources for beginners.

The Arduino boards can be used to easily develop not very complex custom electronic devices, taking inputs from a variety of switches or sensors, and controlling LEDs, motors, and other physical outputs. An Arduino program can be run on its own board, standalone, or it can communicate and interact with software running on a computer or other devices through different means. The boards can be assembled by hand or purchased preassembled.

There are many Arduino boards available to choose from, and many of them perfectly fit the requirements of this project. As a representation, the UNO and MEGA models have been analyzed, two of the most popular ones. These two boards provide more than enough resources for the requirements defined in the scope. They can easily be connected to the computer by USB, and have an additional external power supply input of 9-12 Volts.

In the figure 4.2 the different elements that form an Arduino UNO board are depicted.

The anatomy of the MEGA board is quite similar, but has been left out for brevity. Instead of an ATmega-328 microcontroller, the MEGA board comes with an ATmega-2560, plus more digital and analog pins. Additionally, the MEGA board has three serial input and output (RX and TX[6]) pin pairs, in contrast to the single pair of the UNO board. This is due to the more complex processor of the MEGA.

As a last note, it should be mentioned that apart from the Arduino boards, the "*chipKIT* Max32 Prototyping Platform" developed by Digilent[7] was also considered. It is basically the same board as the Arduino MEGA, but adds the performance of the Microchip PIC32 microcontroller. It is also compatible with many Arduino shields[8]. However, at the beginning of the project there were not many of such boards available, and instead of buying new ones, the Arduinos were chosen over them. In addition to that, the Arduino community and the support that its members can provide is much larger when it comes to original Arduino boards, so any sort of problem that appears with the *chipKIT*s may take much more time to be fixed.

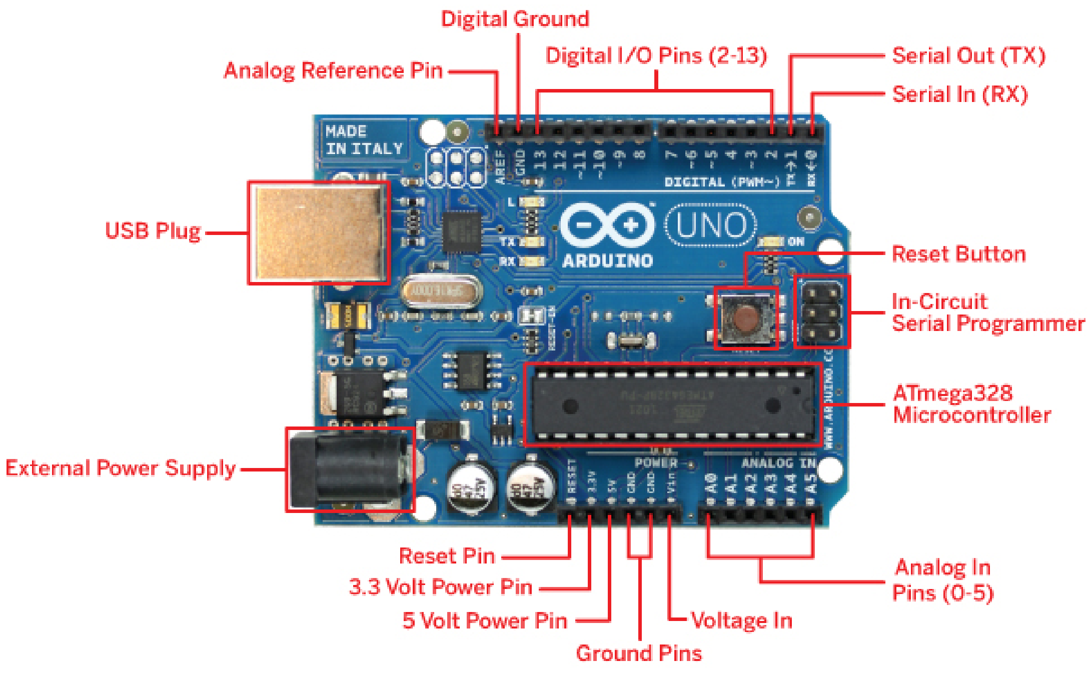


Figure 4.2: Anatomy of the Arduino UNO board (source).

[6]RX stands for receiver, and TX for transmitter.

[7]Digilent Inc. is an electrical engineering products company with technology-based educational design tools.

[8]Shields are boards that can be plugged on top of the Arduino PCB extending its capabilities.

The DF-Bluetooth module

The Arduino UNO and MEGA boards do not have any Bluetooth capabilities embedded, so an external module is needed. Again, due to their low-cost and that a handful of them were already available in the Egokituz laboratory, the DF-Bluetooth V3 modules from the DF-Robot company (www.dfrobot.com) were chosen.

This module implements the Bluetooth Specification v2.0 +EDR. According to its technical file, it is designed to prevent electrostatic damage to the module, since it has an extra integrated circuit layer which acts as a coating. This extra circuit comes with two LEDs (”STATE” and ”LINK”)which are used to display the module’s status (fast blinking means ”search” state, otherwise ”connected”) and link state (it only blinks while paired) respectively—if neither LED blinks, it means the module is not powered. In addition to this, there are two switches to toggle the module. The ”LED Off” switches the LINK LED to enter a power saving mode, and the ”AT Mode” button sets the board into the AT command mode or automatic binding transparent data mode. In the latter, it acts as a normal Bluetooth device. In AT command mode, the module can be configured by sending commands over a serial connection. The most relevant parameters that can be configured are the module’s name over Bluetooth, the baud rate, and the master or slave mode.

In figure 4.3 a DF-Bluetooth V3 module is shown.

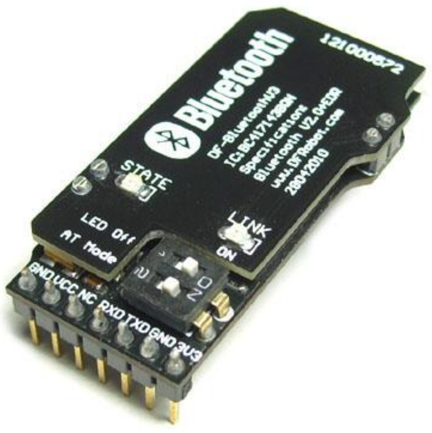


Figure 4.3: A DF-Bluetooth V3 module (source).

Tri-state buffers

In this project, some tri-state buffers have been used to switch the Bluetooth modules’ power programmatically.

In digital electronics, a buffer[9] simply lets its input pass unchanged to its output— the opposite behavior of a NOT gate—. A tri-state buffer, however, adds an additional ”enable” input pin-out that controls whether the primary input is passed to its output or not. If the ”enable” pin-out’s value is true, the tri-state buffer behaves like a normal buffer. If the "enable" pin-out’s value is false, the tri-state buffer passes a high impedance signal[10], which effectively disconnects its output from the circuit (i.e. it lets no current through).

A very comprehensible explanation about tri-state buffers can be found in [Lin, 2003], where the logical components that form these devices are also analyzed.

Table 4.2 shows the truth table for a tri-state buffer.

Enable Input	**Input A**	**Output**
false	false	hi-Z
false	true	hi-Z
true	false	false
true	true	true

Table 4.2: Truth table for a tri-state buffer.

Figure 4.4 shows the pin-out diagram of the L293D tri-state buffer used in this project. Note that technically, the L293D is not just a tri-state buffer, but a ”power driver” with several other capabilities.

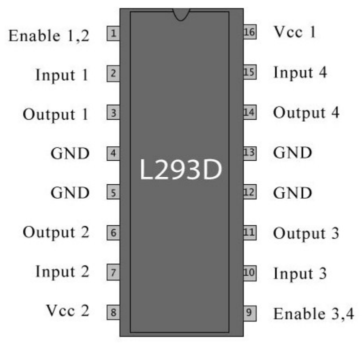


Figure 4.4: Pin-out of the L293D tri-state buffer (source).

[9]A buffer (or buffer amplifier) provides electrical impedance transformation from its input to its output.
[10]In electronics, high impedance means that a point in a circuit (a node) allows a relatively small amount of current through.

4.2.4 Python

Even if it is just a helper tool, a proper explanation on Python should not be left over. It is a cross-platform and general-purpose programming language created by Guido van Rossum. One of Python's philosophies is that its core should remain small, and any other desired functionality should be achieved through extensions. There is therefore a huge standard library ready to be used by the developers. Unlike Java, Python uses dynamic type checking—the type safety of the programs is verified at runtime, and not when compiling—, and can also be used as a command line interpreter.

So far, two major versions of Python remain in use: Python 2.7 and Python 3.3. While the latter contains new features and possibly fixes some bugs of the former, it is not backwards compatible. This could be a problem, since not all the libraries from Python 2.7 and earlier releases are completely available for the last versions. And even if they are, the syntax can vary.

For this project, the *Anaconda* Python distribution has been chosen. It is a completely free to use and distributable software from the *Continuum Analytics* company, and runs under the Windows XP operating system (among others). Its default install comes with Python 2.7, so no previous nor further setup would be needed to start working with it. Anaconda aims to simplify the package management and deployment of the Python libraries, and provides a visual interface and a command prompt to easily interact with. In addition to this, Anaconda comes with common libraries installed and additional packages can be easily added.

Python offers a wide variety of plotting packages, and *NumPy*, *MatPlotLib* are of uttermost interest for this project, as they both enable the use of Python in scientific computing. NumPy allows for multidimensional arrays and matrices, and it also contains a large library of high-level mathematical functions. MatPlotLib, on the other hand, provides an object-oriented API for creating and embedding plots into any program. These two libraries alone make Python perfect for the logged data analysis required for the testing process of this project. A comprehensible starting guide can be found for each of these libraries in [Numpy-Developers, 2013] and [John Hunter, 2013], respectively.

5. CHAPTER

Requirements capture

This whole project comes from the idea that Bluetooth based communications consume too many resources in Android. That is just a hypothesis though, and a framework that provides a controlled environment to prove this theory—in other words, a testbed—is needed. Therefore, two things must be clearly defined: what is going to be measured, and how.

5.1 Performance tests

In order to obtain objective and accurate results, the performance tests should be carefully designed. There should be as little "noise" (e.g. phone applications running in the background, consuming extra resources) as possible, so the whole system must be built accordingly. In addition to this, the tests should be executed with precise parameters, modes, states and the like, not randomly. Taking all this into account, a detailed testbed must be defined, where the possible ”scenarios” and ”discovery plans” are adequately identified and parameterized.

5.1.1 Scenarios

An scenario refers to the environment that a specific test is going to expect. In essence, it defines whether the number of active Bluetooth nodes is going to be static or dynamic during a given test. In a static scenario, it is not expected that new devices will appear (nor the current ones disappear) during a test. Logically, the exact opposite situation will not be unexpected in a dynamic scenario.

If the scenario is dynamic, the devices could appear and disappear randomly or all at the same time. It has just been said that the tests should not be random, but evaluating a changing environment in a moderately controlled way could be of uttermost interest. Therefore, the devices in a dynamic scenario should be able to appear and disappear at will, but in a synchronized manner. Hence, instead of letting each device appear at its own time, they should do so progressively. For the progressive timing, the appearances should be treated as units, so while the potential devices are appearing, none of the currently reachable ones should disappear (and the same would apply to the disappearances).

In addition to this, an special case can occur in the progressive mode, exactly when the time between the appearances is near to null. In order not to mess up the situation more, progressive appearances will have a minimum time of fifteen seconds between each other.

So as to keep the number of potential tests simple, a few fixed scenarios have been defined:

Stable scenario When the test begins, no new Bluetooth devices will appear nor disappear in the nearby area. This considers two possibilities: either no device is nearby (”void” scenario), or at least one device is reachable (”n-devices” scenario) at the beginning of the test.

Progressive scenarios When the test begins, there can be more than one device, or none at all. If at the beginning there is no device reachable, a number of devices will progressively begin to appear eventually. Once the situation has established, it will keep itself like that for some time (or even indefinitely), until a progressive disappearance of the devices occurs. This cycle may repeat itself.

5.1.2 Discovery plan

Since every Bluetooth connection starts with a discovery process[1] by the master device (i.e. the phone), a ”discovery plan” is crucial. From the outset, three key possibilities or ”modes” have been identified for Bluetooth’s discovery process:

- One single or ”initial” discovery
- A ”continuous” discovery
- ”Periodic” opportunistic discoveries

Additional discovery plans could have been included, such as ”chaotic” discoveries which would be ran at random periods of time, or a ”logarithmic” plan which would

[1]See section 4.2.1 for more information on the discovery and pairing process.

increase the period between the discoveries over time. However, they are not required for the testbed, and have been left out for simplicity.

When the discovery has found a device, two paths can be followed. Either nothing is done until the discovery finishes, or a connection request is sent to the target device. Now, this choice entails some trickiness. If we choose to wait until the discovery finishes to dispatch the connection, but the discovery ”mode” is continuous, what have we been waiting for? The answer is void. But why should we wait in the first place? Well, reviewing Bluetooth’s discovery process (particularly in Android), it seems that it is quite a heavyweight procedure, and trying to open a connection while discovering may be source of unexpected problems.

This brings up several issues to consider. If when a device has been found, an immediate connection wants to be dispatched, we can either keep on with it—regardless of the potential problems that may appear due to trying to open a connection while discovering—, or abort the ongoing discovery and then try to open a connection. If the second option is chosen, it should be considered whether a discovery should be instantly started right after the connection setup is finished or not.

Back to square one, if we chose to wait until the discovery is finished to fetch a connection, the possible paths fork again. What if more than one device has been found during the last discovery? Should they be connected at the same time, or progressively? After researching the possible outcomes of setting up multiple connections simultaneously, no preceding cases where more than one connection had to be done at the same time have been found. However, taking into account that the discovery is a heavy procedure, it is not absurd to believe that simultaneous connection set-ups could show unexpected problems. Therefore, and seeing it as yet another potential source for a valuable finding by this project, the testbed should consider the two connection-timing possibilities: connecting all devices together—knowing that something could go wrong—, or progressively, fetching each connection just after the previous has finished.

In a few words, these are the discovery plan parameters:

- Discovery mode:
  - Initial.
  - Continuous.
  - Periodic.
- Connection timing:

  - Immediate, while discovering.
  - Immediate, stopping the ongoing discovery.
  - Delayed, after finishing the ongoing discovery.

- Connection mode:
  - Progressive, connecting one device after another.
  - All-together, connecting all devices at the same time.

It is clear that the number of tests that could be made just by combining those three parameters is too large. To cap it all, taking into account that there is also more than one scenario, it could take a lifetime to carry through all the performance tests and later analyze the results. Therefore, a smart selection of the most relevant test-cases must be done. In table 5.1 some potentially interesting discovery plans have been portrayed. The final pool of discovery plans chosen—along with their corresponding scenarios—is left to be decided during the design execution of the project.

	Initial Discovery	**Continuous Discovery**	**Periodic Discovery**
Immediate conn. while disc.	—	Progressive, All-together	Parameterized
Immediate conn. stopping disc.	—	Progressive	Progressive
Delayed conn.	Progressive, All-together	Progressive, All-together	Progressive, All-together

Table 5.1: Discovery plan parameters.

5.2 Performance testing variables

In order to prove that Bluetooth consumes too many resources, certain variables should be monitored. In section 2.2.3 of the first chapter, some test conditions were introduced. Following, those variables are explained in more detail.

5.2.1 Hardware variables

The first changeable item or condition is the smartphone with which the test is going to be carried on. At this moment, there are three possible candidates to try the tests with:

- Nexus S
- Galaxy Nexus

- Nexus 5

Table 5.2 sums up the most relevant technical details of the devices at the time they were used for the tests. Each phone has a different OS version, battery, CPU and Bluetooth support, so it will not be possible to say how much each individual variable affects the tests' results. In the coming section 5.3, a more insightful analysis over the phones is made.

Device name	**Nexus S**	**Galaxy Nexus**	**Nexus 5**
Brand	Samsung	Samsung	LG
Version Name	Jelly Bean	Jelly Bean	KitKat
OS version	4.1.2	4.3	4.4.4
API version	16	18	19
CPU	1 GHz single-core	1.2 GHz dual-core	2.26 GHz quad-core
Battery	1,500 mAh [2]	1,750 mAh	2300 mAh
Bluetooth	v2.1	v3.0	v4.0

Table 5.2: Devices used for the performance tests.

5.2.2 Signal features

There are several parameters in which the signals can vary:

- Data reception frequency
- Data sending frequency
- Serial baud-rate
- Framing/packet size
- Radio interferences

Among those variables, the radio interferences are the only that cannot be toggled, due to its nature. Nonetheless, the tests should be carried as far as possible from any interfering wireless source (like a WI-FI antenna). The data input and output frequencies cannot be toggled in a physical level, as they are dependent of the underlaying hardware. The same happens with the packet size. Still, it is possible to customize the message size, as well as the message sending and reading times in the application layer. The format for the messages is defined in section 7.4.

The value of the serial baud rate is not usually easy to change (if possible). Unless otherwise required, this value should be set or left to its default.

[2]An ampere hour (abbreviated as Ah) is the amount of energy charge in a battery that will allow one ampere of current to flow for one hour.

5.2.3 Other variables

In the recently mentioned section 2.2.3, some other variables such as the distance between the devices, or the number of threads per device were identified. Taking into account that the Bluetooth protocol can be guaranteed to function at a minimum distance of a meter, the testbed should be defined within that radius. As for the number of threads goes, it cannot yet be specified, but a balance seeking the best performance should be found.

The sensors may be powered on and off by hand, but a programmatic way could ease the benchmarking process considerably.

5.2.4 Desired data measures

The testbed should easily provide measurements for both CPU and battery consumption rates for at least the Android device. The throughput[3] and response times for each and every sensor should also be available at any moment—that is, if they are connected.

The measurements need not be displayed in a live format, since doing so could consume more resources than needed, and the benchmark's values would not therefore be entirely valid.

5.3 Physical testing-framework

Taking the aforementioned in mind, the system that is going to be developed requires at least an Android powered smart-phone and a pool of sensors that transmit data through Bluetooth. The Android application would uninterruptedly measure different variables such as the throughput of the communications with the sensors, the battery consumption, and the like. This measures must be stored in some way for later use—e.g. in a text file or an FTP server. A computer or some sort of terminal is also required to view and study the communications' results.

5.3.1 Android device

It is required that the Android device (or devices) used has Bluetooth capabilities. In addition to this, the device's implementation of Bluetooth must comply with the v2.1+EDR

[3]In communication networks, throughput is the measure for the amount of data that can be transferred from one unit to another in a given amount of time, usually given in bits per second

specification of said protocol, since that's the DF-Bluetooth module's version. Finding such characteristics is not a problem at all, considering that almost every smartphone in the world has Bluetooth as a built-in feature. As for the version goes, it is also task of the OS to give support to the different Bluetooth specifications out there. In the case of Android, API level 5 or higher should be enough, but due to the deep changes in its implementation of the Bluetooth stack, it is a far better choice to use the API level 18 or higher—that is, Android 4.3 *Jelly Bean* or subsequent versions. More information about Android's implementation of Bluetooth can be found in section 4.2.2.

5.3.2 Bluetooth sensor pool

The devices that play the role of Bluetooth sensors need to be cheap and easily built. Therefore, they are going to be entirely made from Arduino boards and DF-Bluetooth modules.

These custom ”Arduino sensors” must meet some specific requirements. First, since the tests focus on the phone's side, and *a priori* it is still unknown how much time the phone's battery will last, it is compulsory that the devices' batteries outlive the phone's. This means that the sensors must have an uninterrupted source of power. Secondly, they must provide a mechanism to receive commands and act accordingly.

5.4 Use casess

A total of three use cases have been identified. It should be borne in mind that for the time being, this application is not targeted at a big group of people, as just serves as a tool for setting a benchmark in Bluetooth communications. Therefore, the use cases explained below could as well be treated as a unitary ”perform a benchmarking test” use case. Figure 5.1 shows the three use cases identified, and the following sub-sections explain them in detail, with their own sequence diagrams.

5.4.1 Start a performance test

The first use case would be the one in which the tester sets some parameters in the Android phone and starts a new test. The test then begins and no more interaction is needed. The application will run until the phone's battery ends, or the user kills the application. Figure 5.2 shows the sequence diagram for this use case.

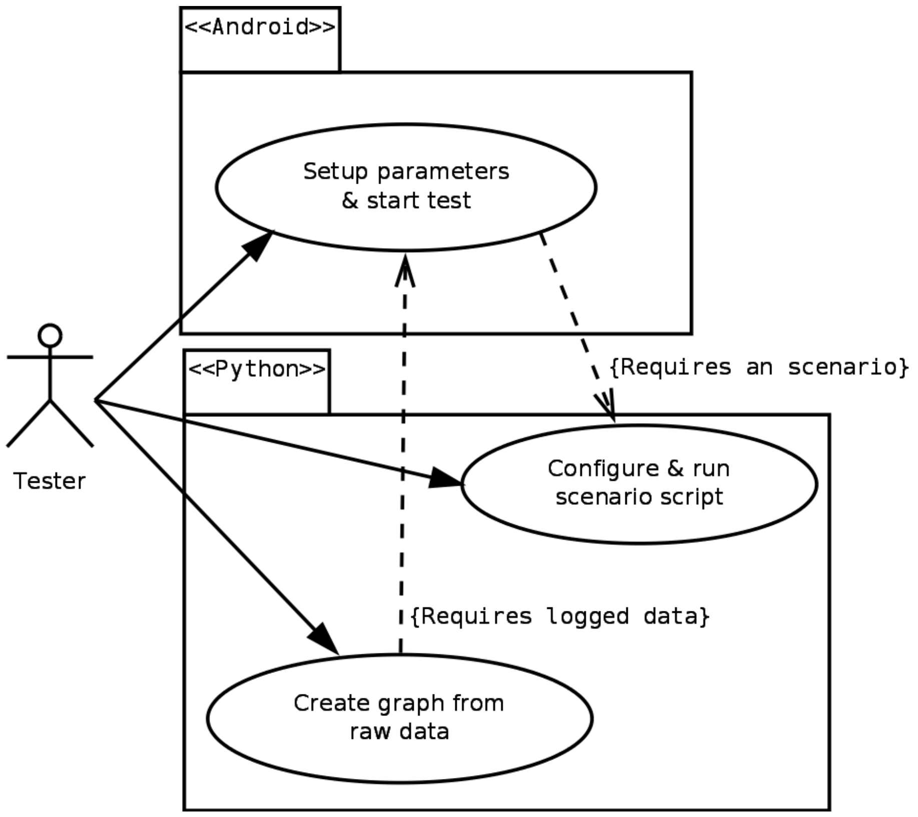

<<Android>>
Setup parameters & start test
{Requires an scenario}
<<Python>>
Tester
Configure & run scenario script
{Requires logged data}
Create graph from raw data

Figure 5.1: Use cases of the system.

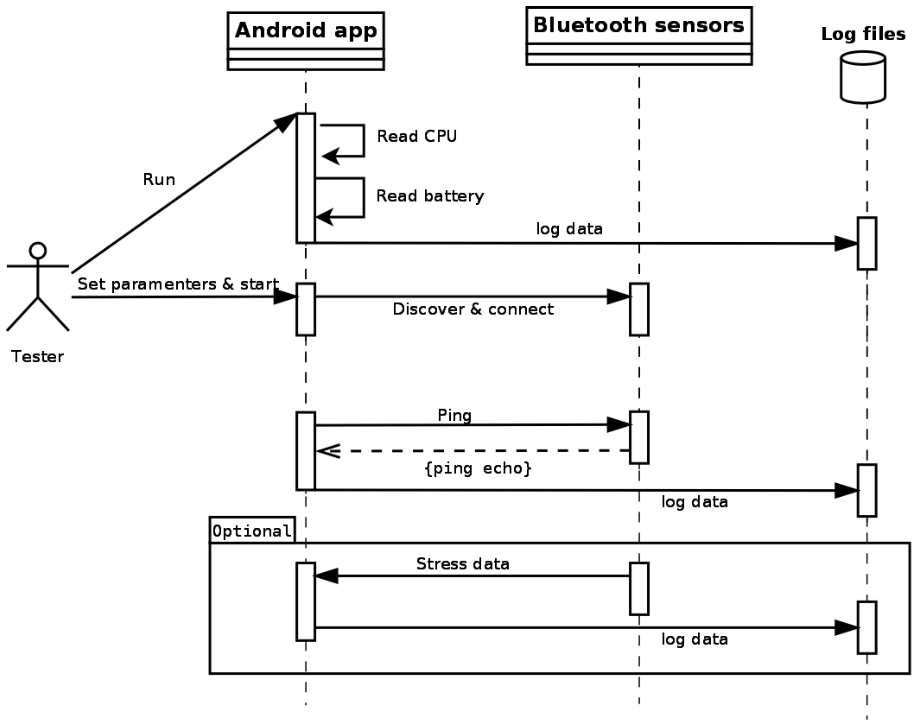


Figure 5.2: Sequence diagram for starting a test in the Android application.

When the application is run, it immediately starts to perform CPU usage and battery readings, and it also stores them in some log files. Then, right when the test parameters are set, if the scenario is set with one or more Bluetooth sensors, the application may find and connect to them. From then on, it will continuously send ping signals and store the obtained values. Note that if there is no scenario set (by the tester), the outcome of this test cannot be foreseen.

5.4.2 Set a scenario

The next use case is the one in which the user or tester sets a scenario. In a few words, setting a new scenario means tweaking the Python script that communicates with the serial ports of the sensors (Arduino boards) so that they perform some specific actions, and running it. Among other things, the serial-script can decide when to connect to the boards, and when to send some particular commands to them, so that they behave one way or another. The serial script may or may not run indefinitely, depending on the tester's chosen configuration. Figure 5.3 shows the sequence diagram for setting a scenario.

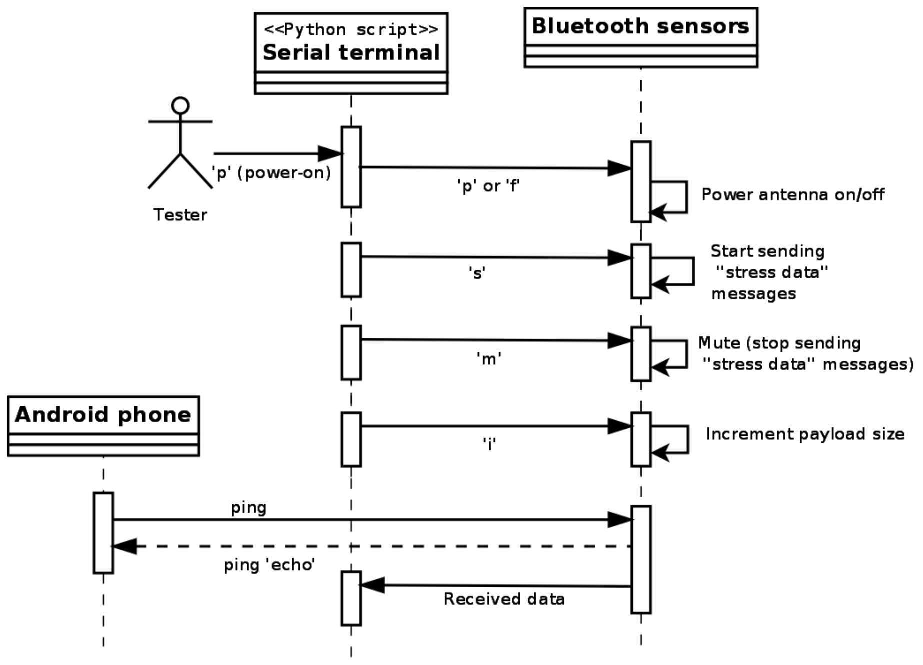


Figure 5.3: Sequence diagram for setting a test scenario through a Python script.

5.4.3 Create a graph from raw data

The last use case corresponds to the process of taking the raw data from the smartphone and processing and plotting it through a Python script. The plotting process requires that the log files have been copied from the smartphone to a specific folder in the computer in which the Python plotting script is going to be run. Figure 5.4 shows the sequence diagram for this process.

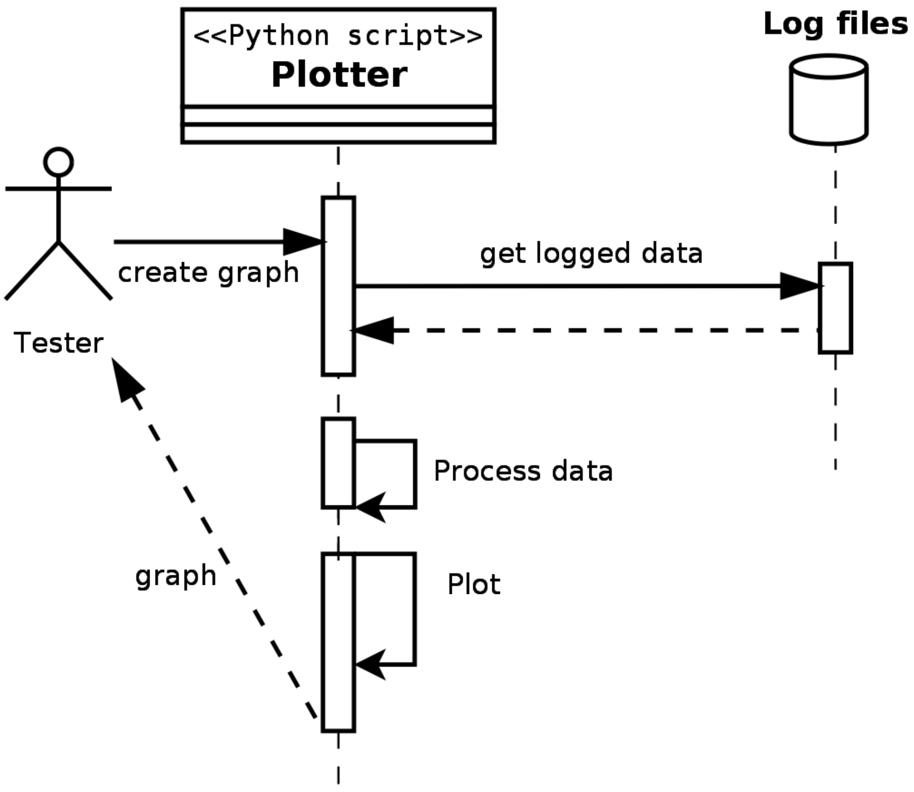


Figure 5.4: Sequence diagram for creating a graph through a Python plotting script.

6. CHAPTER

Analysis

The purpose of this chapter is to explain the thought process done before designing the system developed. Many things have already been explained in the previous chapters, but here some extra considerations are made.

6.1 Bluetooth

6.1.1 Why Bluetooth?

Bluetooth encompasses several key points that facilitate its widespread adoption. As stated in *"What is Bluetooth?"* [McDermott-Wells, 2004]: 1) Bluetooth is an open specification that is publicly available and free; 2) its short-range wireless capability allows peripheral devices to communicate over a single air-interface, replacing cables that use connectors with a multitude of shapes, sizes and numbers of pins; 3) Bluetooth supports both voice and data, making it an ideal technology to enable many types of devices to communicate; and 4) Bluetooth uses an unregulated frequency band available anywhere in the world.

These features make Bluetooth an outstanding resource worth trying to work with.

6.1.2 Technical specifications

The Bluetooth specification version that is used in this project is v2.1+EDR[1], although it is not the purpose of this document to explain the specifications and technology affairs

[1]EDR stands for Enhanced Data Rate.

of the Bluetooth protocol—there's more than a thousand pages for the V2.1+EDR core version, and almost three thousand for the last core version (4.1) of the protocol available in the official Bluetooth website for whoever is interested—, a few particular features should be recalled.

For instance, the maximum theoretical throughput that can be achieved with Bluetooth V2.1+EDR is 3Mbit/s, but the practical data transfer rate is 2.1 Mbit/s [Kewney, 2004].

Another important thing to bear in mind is that there cannot be more that seven "slave" devices simultaneously connected to a "master" device in a piconet. Moreover, not all "master" devices may even be able to connect with seven slaves. Still, the Bluetooth protocol enables the formation of *scatternets*[2] to connect more than seven devices with a master. However, this strays from the scope of the project, and will not be taken into account while developing the system. This could possibly be a good future work that stems from this research.

Finally, it should be noted that the Bluetooth protocol has always suffered from signal interferences. As Bluetooth operates in the 2.4 to 2.4835 GHz electromagnetic band, problems may appear if a Zig-Bee[3] or any other IEEE 802.15.4[4] based protocol is running nearby. In addition to that, in [Intel, 2012] it is claimed that USB 3.0 devices, ports and cables have been proven to interfere with Bluetooth devices due to the electronic noise they release falling over the same operating band as Bluetooth. This issue should be considered when designing and performing the tests of the present project.

6.2 Hardware used

6.2.1 Android Smartphones

There are a total of three different available Android devices with which to test the application (as previously shown in table 5.2). From older to newer, a Galaxy Nexus, a Nexus S, and a Nexus 5. All these three devices are incidentally some the official smartphones for which Android is developed. What this implies is that Google's staff developers actively use these very same models to develop their applications—or even the Android OS itself—. As a consequence, these devices should in theory give less software problems than any other models in the market.

[2] A scatternet is an ad-hoc network formed by two or more piconets.

[3] ZigBee is a specification for a suite of high-level communication protocols used to create personal area networks based on IEEE 802.15.4. See section 3.3.3 for more information.

[4] IEEE 802.15.4 is a standard which specifies the physical layer and media access control for low-rate wireless personal area networks

As for the Android versions of each device, there's a wide variety as well. The Nexus S has the 4.1.2 version of Android (API level 16); the Galaxy Nexus has the 4.3 version (API level 18); finally, the Nexus 5 has the last current version available (4.4.4, and API level 19, as of the time of this writing). Except for the Nexus 5, the other two are in their respective top updates possible, so they will remain in those versions indefinitely.

It should be noted that the devices have been previously used for other research purposes, some of which may have exhausted the battery life hope considerably.

As stated in the review done over *Android & Bluetooth* in the state of the art (see section 4.2.2), Android has a completely new Bluetooth stack since API level 18. This means that the Nexus S has a different underlaying Bluetooth infrastructure and API-calls to the other two phones, which have the newer stack. This could bring up a potential problem, and prevent us from performing trustworthy performance tests. What's worse is that Android 4.1.2 appears nowhere on the map.

Additional research has brought some light on the matter. Even if there is not a single mention of Android 4.1.2 in the official pages, and other sources do not provide any meaningful information, the upgrades made in API 18 do not seem to alter the Bluetooth's normal API, according to [Andrdoid-Developers, 2014b]. The updated API only adds a handful of classes for Bluetooth Low Energy, so there will be no problem at all by using the Nexus S.

It should be noted that the three phones may be used for the tests, but not all of them will be equally available. Only the Galaxy Nexus is guaranteed to be at hand anytime during the project. The other two may be requested by other members of the Egokituz laboratory.

6.2.2 Bluetooth sensors

As explained in an earlier chapter, the Bluetooth devices developed in this project are made of Arduino UNO and MEGA boards plus a DF-Bluetooth module. However, even if they have been referred to as ”sensors” throughout the whole project, they are not that much like it. Actually, they should be called Bluetooth beacons or stations transmitting radio signals, since they do not actually sense anything.

The reason why Arduino was chosen over existing Bluetooth sensors or other chip-sets is twofold. On the one hand, there is no sensor in the market that can be easily configured to send data the way we want, when we want (in other words, Arduino lets us control the test conditions better than anyone else). On the other hand, Arduino has demonstrated to be one of the most popular microcontroller platforms. Finally, only an affordable Bluetooth module compatible with Arduino had to be appointed. Due to its availability and reduced cost, the modules from *DF-Robot* were chosen.

As previously mentioned in the requirements capture, the Bluetooth sensors must be able to be switched on and off. The Arduino boards offer two possibilities for this: DC adapter plug—specifically 9 to 12V DC, 250mA or more, 2.1mm plug, center pin positive adapters—and/or USB connection. Batteries could as well be connected to the Arduino's DC plug, but since batteries are limited by time, they are of no use for our purposes. The DC adapters do offer what is needed, but there are not as much as 8 of them at hand, so that's a no-go as well. Still, even if more were available, most DC adapters are quite bulky, and the expansion sockets needed to plug the adapters would take quite a lot of space as well. Therefore, no other choice than USB is left. Luckily enough, there are many unused USB-2.0 wires with standard A and Standard B plugs available. Additionally, there's a *D-Link* expansion hub that outputs four extra USB-sockets.

The Arduino boards provide more than enough *pin-outs*[5] for what is needed in this project. The most interesting features of the UNO boards for our requirements are:

- 13 configurable digital pinouts, two of which provide serial capabilities (one for inputs and another for outputs).
- One 5V and one 3.3V output power-supply pinouts.
- Three ground pinouts.
- A reset button.
- One USB plug.

The MEGA boards, on the other hand, offer a few more pinouts. Instead of the 11 digital pins of the UNOs, the MEGAs provide 25. Of those, there are 3 pairs (pin-outs 14 to 19, specifically) which provide capabilities for serial communications.

As a last remark, it should be noted that the output power supplies of the Arduino boards cannot be toggled, so they always provide power—that is, if the board itself is powered—. In addition to this, the DF-Bluetooth modules do not have any mechanism to toggle them ON or OFF either. If the modules are going to be switchable, a mechanism must be designed that enables this feature[6].

[5] A *pin-out* or *pinout* refers to the electrical connection points of an electronic device.
[6] The solution given consists of the *tri-state buffers* exlpained in section 4.2.3.

6.2.3 Environment

Apart from the system's components themselves, the environment in which the tests will be carried on needs to comply with the requirements identified in the previous chapter.

Since the tests can take several hours, or even days, they should be performed in a place were the sensors and the smpartphone can stay still, and do not disturb. Additionally, the tests should be performed under as less interferences as possible. Since no interference detector is available, not much can be done other than trying to find a place away from any electrical power source and WI-FI antenna. The Egokituz laboratory in the university where this project is being developed should provide a good place that meets all these requirements evenly. The most appropriate hours may be at night, when there should be nobody working, as the possible interferences will be the lowest and most homogeneous.

Finally, it will be taken for granted that all the sensors with which to make the tests have already been paired (i.e. introduced the security PIN code) with the smartphone being used during each test, so no "artificial" time is lost while the user enters the required pins manually.

6.3 Software applications

Even if the applications developed are fully explained in the design chapter, some questions had to be considered before designing them.

6.3.1 Android application

The Android application that is going to be developed needs to be as efficient as possible, so that the tests' measurements are not tainted in the slightest. In addition to this, the overall application's components should be arranged keeping a balance between modularization and performance. However, performance will be chosen over modularization or other design concerns, for obvious reasons.

The application should present the user with a simple interface where some test parameters can be chosen. A button or any other artifact to instruct the program to start the test with the selected parameters will also be accessible.

Accessibility concerns, or multi-language possibilities and even layout traits should be overlooked, unless the scope of the project is modified. The reason behind this is simple: the work overhead does not pay off, and it does not practically add more value to a system which is not supposed to be interacted with.

The identification of the Bluetooth devices for the tests can be done either by checking each device's friendly name or MAC address. Additional mechanisms might be used as well, but are not necessary.

6.3.2 Arduino Application

The Arduino application should follow the same guidelines of the Android application regarding its performance. The program must be able to work on its own while powered on, but there should be a way in which to receive commands and act accordingly.

The program should have at least two main behaviors or states: in the ”PING” state, all the received Bluetooth signals are echoed to its source. In the other behavior, the ”STRESS” state, the board would uninterruptedly send a message or packet of a fixed size in periods of really short time, also fixed. Note that both states should be able to work overlapped at the same time.

The messages sent between the Android and Arduino application should follow a pre-defined message format, with some headers and then the payload (actual content). This should boost the control possibilities over the message count, error rate, etc. See 7.4 for a detailed description of the message format.

6.3.3 Python Scripts

Two types of scripts are going to be needed: one type will serve for plotting purposes, and the other for sending commands to the Arduino boards through serial communications. It has to be borne in mind that these scripts serve the purpose of simplifying a couple of tasks of the benchmarking process for the overall system. Should the fulfillment of the project be at risk, these scripts would be the first dispensable components to be cut back.

Plotting

Taking the tests' raw results in the form of text logs, there should be a script that generates a graph for each identified variable. From the outset, it is expected to get graphical representations for the battery percentage, CPU usage, PING time and throughput values.

The script may automatically generate and export the graph to a common image file format, or let the user customize the layout and other features of the project and let them decide to export the or discard the graph. This will be decided conveniently at a later time.

Serial communications

It is still unknown how much time will each test case take, as it all depends on the battery of the phone—that's what this project wants to find out—. However, a script that lets the user quickly define some commands to be sent over a given time span or at specific time intervals could be of most help. The script should be able to automatically open serial COM ports for each Arduino connected to the computer, and should also have the possibility of sending commands to different devices simultaneously.

Developing a graphic interface with which to interact with goes beyond the goals of the project, so a command-line window must provide all the script's functionalities. Developing a GUI could be an interesting future work.

6.4 Google, Android and Bluetooth

During the *Google IO*[7] in 2013, a presentation titled "Best Practices for Bluetooth Development" took place. During their speech, Sara Sinclair Brody, Rich Hyndman, Matthew Xie not only explained the best practices for Bluetooth development in Android, but they also presented the new version of Bluetooth, the so called BLE (Bluetooth Low Energy) or Bluetooth Smart.

According to the hosts, it is a good practice to check if the Android device supports Bluetooth, even if the vast majority of existing Android devices do so. They also note that as of the current API version—18 at the moment of their presentation—, there is no way of switching on Bluetooth without the user's interaction or awareness. Supposedly, the reason behind this decision is a security concern, so that no external actors can get access to Android's inherent Bluetooth adapter.

Later on in the same speech, a secure way to establish connections with Bluetooth devices is presented. What's curious about this is that the presenters themselves believe that such way of establishing secure connections is not worth the effort that it supposes, since the probability of a ”Man In The Middle” attack is almost null.

Additionally, during the 2013 IO presentation, the hosts responded to several questions regarding Bluetooth Classic and BLE. They stated that as of API 18, Bluetooth LE does not support broadcasting capabilities. This means that Android devices can only run the ”Central Role”, or in other words, are not discoverable by other devices. At least not

[7]Google I/O is an annual developer-focused conference held by Google in San Francisco, California

while Bluetooth LE is being used. This should be taken into account if future work on this project is going to be made.

For whoever that deals with Android and Bluetooth, it is worth checking the whole IO presentation (which can be found in [Google-IO, 2013]).

7. CHAPTER

Design

In this chapter the overall design of the system is gathered. This will give an insight of the way in which each individual component of the system interacts with the rest.

7.1 Overview

The system developed consists of several pieces or components. On the one hand, there's the hardware side, which lies in the Android smartphone and the Arduino boards, plus at least one computer or terminal to send commands to the Arduinos via USB and analyze the test results. On the software side, an Android application is needed for the smart-phones, an Arduino program to echo incoming packets and periodically send ”junk” messages, and some Python scripts. It is taken for granted that in order to compile the programs and manage the peripheral devices, a computer (needs not be the same as the terminal sending serial commands) is also being used, but this will not be taken as part of the system. The data logged by the Android application may also be copied to a computer (again, this could be done elsewhere, in an extra machine, or in the same one) and the plotting scripts would generate some graphs for later analyzing them. Figure 7.1 depicts an overview of the system, where the most meaningful information exchange between the components is shown.

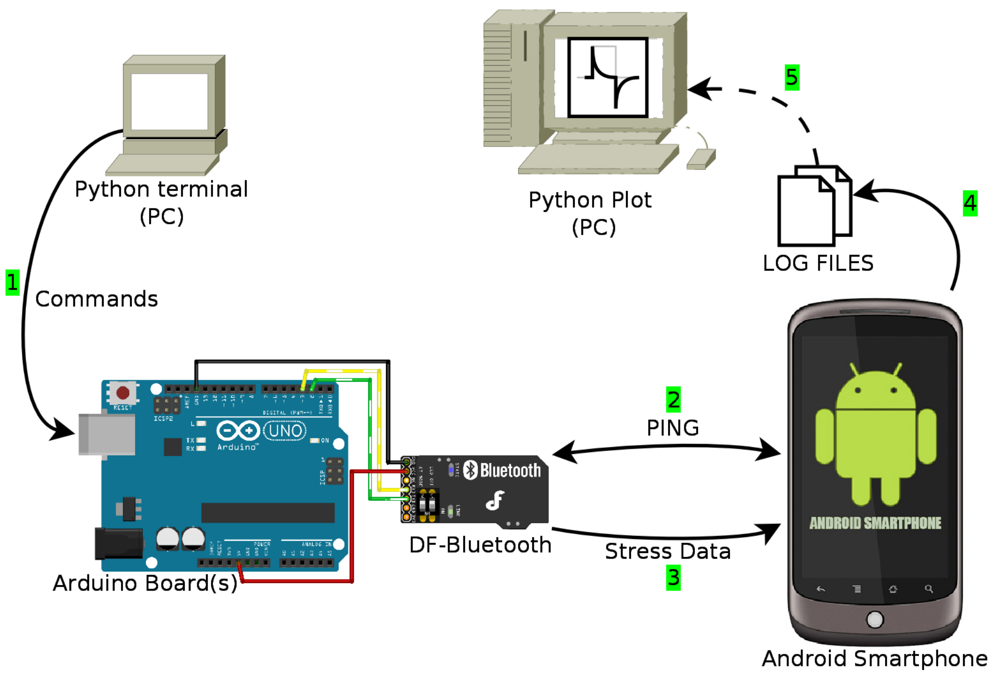


Figure 7.1: Overview of the testbed designed and inter-component communication.

7.2 Android Application

The Android application takes a central role in the system. It is responsible for managing the Bluetooth antenna of the phone to find nearby Arduino devices, connect to them, and carry on read/write communications. It is also responsible for logging different data to the phone's external storage or SD memory card, as well as dealing with layout control, processing external input and responding to requests. If this component misbehaves, the system's integrity in null. Hence, it is mandatory to precisely design this application's structure.

7.2.1 Application structure

The Android application consists of six main classes, but one of them contains two additional ones "nested" in it. They can all be divided into three groups according to the three-tier architecture[1]:

- The *MainActivity* makes up for all the layout control, in the presentation tier.

[1]In software engineering, the three-tier architecture is a client–server architecture in which presentation, application processing, and data management functions are physically separated, so that they can be developed and maintained as independent modules.

- The *BTManagerThread*, *ArduinoThread*, *BatteryMonitorThread* and *CPUMonitorThread* classes, which constitute the core of the application, are all part of the business logic or middle tier.
- The *LoggerThread* class would make up for the data tier.

This architecture allows an easier modularization, and should a part of the application be changed, no incompatibility issues would appear. Additionally, there are some extra resources for the presentation layer, as Android's layouts are easier defined in non-Java code. The *activity_main.xml* file contains the *view* object definitions for the main (and only) layout, and additional XML files define the icons and GUI elements' text values, among other things.

Additionally, there's a group of less important classes that are just mere tools. That is the case of *ArduinoMessage*, a class used to build well formed messages[2] and check if the received ones are valid (i.e. the message contains no corrupt data).

In the following pages, a thorough explanation of the classes that conform the application is made. A complete class diagram is shown in appendix B.

Main Activity

The *MainActivity* presents itself with some input controls in the form of radio buttons to select the test-plan parameters:

- Discovery mode: initial, continuous or periodic.
- Device connection mode: progressive, connecting one device at a time, or all together and at the same time.
- Connection timing: immediate, while discovering; immediate, but stopping the discovery in course, or delayed (connection starts once the discovery is finished)

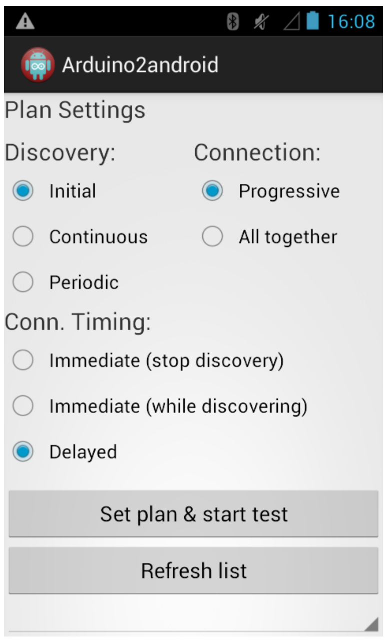
To tell the program that a new test should begin with the selected parameters, a ”Set plan & start test” button is presented below parameter control radio-buttons.

Whenever a relevant event occurs, such as a discovery start or finish, or a successful connection to a device, a so-called *toast* is presented in the screen, containing a timestamped message representing the event in question. This *toast* will only appear for a short period of time in the screen, and of course, it won't be seen if the screen is locked.

[2] A predefined message format for the communications between Android and the Bluetooth sensors is explained later in this chapter, in section 7.4

It is clear that the graphic interface is minimal, and so it should remain to consume as few resources as possible. In case some feedback is needed to see whether communications are being carried on with a given device, this could be done simply by watching the RX and TX built-in LEDs of the device's board. In fact, the DF-Bluetooth antennas' LEDs themselves inform about the connection state (paired or unpaired) by default.

In figure 7.2 the layout of the main interface can be seen.



16:08

Arduino2android

Plan Settings

Discovery: [✓ Initial] Continuous Periodic

Connection: [✓ Progressive] All together

Conn. Timing: Immediate (stop discovery) Immediate (while discovering) [✓ Delayed]

Figure 7.2: "Main Activity" *View* interface

Bluetooth Device Manager

The *BTManagerThread* is the cornerstone of the whole Application. It is responsible for handling Android's BluetoothAdapter, as this is the starting point for all Bluetooth actions in the Android OS. This adapter gives control over the discovery process, and it can also be used to retrieve the list of devices that are already bonded with the phone. At the same time, this thread listens to some broadcast[3] *Intents*[4] related to Bluetooth events that are raised when a device is found, a connection /disconnection is made, or a discovery is started or finished.

The logic behind this class is the core of the whole system, but understanding how

[3]A broadcast is a message that any app can receive.

[4]An *Intent* is a messaging object used to request an action from another app, and facilitates communication between components.

it works is not easy. There are a total of three varying parameters that this class must control. Firstly, it has to manage when to start each discovery (if more than one is to be done). Three are the possibilities for discovery timing: just an initial one, a continuous discovery, and a periodic one. Controlling all these cases can seem tough at first, but it is not if the commonalities between them are found. For instance, the initial discovery must always happen, so there's no need to control whether an initial discovery has to be done or not. It is only necessary to know what action should be made once the first discovery has finished.

At this point, one could argue that the continuous discovery is a particular case of the periodic, with a null delay time between each period. However, due to Android's way of handling delayed tasks for the future, if the discovery had to be truly continuous, these two modes could not be achieved in the same way. Therefore, when the broadcast *Intent* saying that the discovery is finished, the plan mode set is checked, and the program acts according to it. In the case the initial discovery, nothing more is done, as no more discovery requests will be done. In case a continuous discovery ought to be done, it is requested immediately through the Bluetooth-adapter. Should the discovery mode be periodic, however, a post-delayed runnable containing the discovery request is queued into the OS, which will be later executed after the given delay has passed.

Another parameter of the planner is the connection-timing mode, which can be either immediate, while a discovery is in process (or stopping it), or delayed. Unlike in the previous case, there is no need to periodically run a post-delayed method. On the contrary, it is mandatory to control-check the parameter at two different times: first when a new device is discovered, and later, when the discovery is finished. When the system finds a device, a receiver in *BTManagerThread* gets ahold of the newly found *BluetoothDevice* object, and it immediately stores it in a temporal array. Then, the connection timing mode is checked and, if it is immediate, it will stop the ongoing discovery (or not). It will also request the device to be connected through an auxiliary or helper thread (more on that later). In case the connection timing is set to be delayed, nothing is done at this point. As aforementioned, however, the timing-mode is also checked when a discovery is finished, so in case it is set to be delayed, a connection request is fetched after the discovery has finished.

The last piece of the logic control is the connection mode of the devices found. Since the possible cases are only two, progressive and all-together connect, the control is much simpler than the other cases. As this parameter is not bound to time, its control is not done in the same place as the other two. In this case, it is done whenever a connection is requested. If the mode is set to be progressive, only one connection will be processed at a time, and nothing will be done until that process has finished (be it successful or not). Then the next connection will be fetched. If the connection with the available devices is to be made at the same time, however, concurrent and non-blocking processes will be run to perform the connections (more on this later).

As it can be seen, the logic implemented in this class is quite intricate. Further explanations on this control-process can be found in the chapter dedicated to the development process (section 8.3).

Planner Thread and Background Thread Dispatcher

Even if it was not foreseen to use nested classes, in order to maximize the performance while keeping the structure of the application as intact as possible, two additional classes where appended at the end of the *BTManagerThread* class: *ArduinoPlannerThread* and *BackgroundThreadDispatcher*. As the names themselves point out, both extend the *Thread* class, and again, the reason behind this is performance, as this enables performing particular time-consuming tasks in the background that would otherwise block the proper flow of the Bluetooth Manager, and therefore spoil the whole system whatsoever.

The *ArduinoPlannerThread* is used to check if the new devices found are of any interest to us (i.e. they are Arduino devices), and to start the threads that would manage the communications with each of them. At first, the *ArduinoPlannerThread* was in charge of dynamically creating and starting each *ArduinoThread* as well. Since this process took a while to end, it was mandatory for the *ArduinoPlannerThread* to be running on a thread of its own.

Later, though, during the development process of the project, a new and more elegant way of creating those threads was found through the *AsyncTask* class of Android. According to Android's reference, this class is in essence mostly used by the UI threads to run heavyweight procedures or blocking calls in the background. However, there is no limitation for using it the way we are. Thus, the *BackgroundThreadDispatcher* class was created, as this significantly eased the way of launching many threads at the same time.

Arduino Threads

The *ArduinoThread* classes are the only ”dynamic” threads of the application, so to speak, as their life-cycle depends entirely on external factors to the application. Each of these threads are created and started whenever a new Arduino device is detected nearby (that is, with a reachable Bluetooth antenna), and destroyed when the connectivity is lost between the smart-phone and the Arduino.

Each *ArduinoThread* contains its own socket object, as well the corresponding input and output streams associated to each. Three key processes work around these objects. Firstly, the connection process, which takes place when the class is instantiated, opens a socket level connection with the newly found Bluetooth device. If successful, the thread keeps running until otherwise told or connection is lost. While running, each *ArduinoThread* uninterruptedly listens to the input stream of its socket. When a well formed message[5] is received, a notification is sent to the main thread, so that the message can be processed or logged. At the same time, an asynchronous function that writes any given data to the output stream of the Bluetooth socket can be called. Logically, it is this function that will send the *ping* calls to the connected device.

Had it been a smaller group of Bluetooth devices, the most efficient way to deal with the incoming and outgoing messages would have been to keep dedicated threads for each read & write task. However, taking into account the overhead that the inter-process communications bring with them, a different decision was taken. From the experience gained, it seemed that serial communications over Bluetooth were fast enough to avoid building a blocking queue of either incoming or outgoing messages, so simplicity was chosen over the possibility of creating more threads and using the resources they would have required.

As a side note, there's a peculiarity when dealing with incoming messages that was not foreseen at first. The problem in question was that while carrying simultaneous communications with multiple devices at the same time, although each dedicated *ArduinoThread* worked flawlessly, the main thread acted as a bottleneck, since all the incoming messages where passing through it. In order to minimize the unavoidable effect of this drawback, two modifications were made: on the one hand, the handler of the *MainActivity* was pulled out of it, and embedded in a dedicated thread so as to not block the GUI and the whole system. On the other hand, in order to decrease the number of messages arriving at the main thread, each *ArduinoThread* would send the incoming messages in sets of a hundred. Thus, the bottleneck problem was greatly minimized.

[5]See section 7.4, where the message format is explained.

Battery Monitor

The *BatteryMonitorThread* is a class that runs on its own thread, and its sole purpose is to listen for battery state changes. Sporadically, when the Arduino OS sends a broadcast *Intent* with the battery state, this monitor reads the data, calculates the percentage, and sends it in a message to the main process, so that it may later be logged or displayed in the GUI.

CPU Monitor

Much like the battery monitor, the *CPUMonitorThread* also runs on its own thread and looks for changes in the CPU usage. However, the way this is achieved has nothing to do with *Intent* broadcasts. Indeed, this thread checks periodically (every second) the */proc/stat* file in the Android device, and reads some specific data-fields, with which the usage is inferred. The format and content of this file, as well as the process of obtaining the CPU usage, is explained in section 8.3.6 of the development chapter. But in essence, it is the same as in most UNIX operating systems: it contains various pieces of information about kernel activity that are automatically updated by the OS.

Logger

The *LoggerThread.java* is the only component in the third tier of the architecture of the program. Unlike in most other systems where both data persistence mechanisms and data access layers are implemented, in this application only the storage option (persistence) is given, as the logic tier needs not any past data.

This class runs in the background doing nothing else than to wait until a message from another thread is received. Each of these messages will bring with them a string or array of strings that need to be written to a log file.

For performance reasons, the logger will write the data to the corresponding files without returning any response to the thread that made the logging request, unless an exception while writing has happened. In that case, a message to the main thread (*MainActivity*) will be sent.

7.2.2 Data flow

As aforementioned, the gist of the program resides in the *BTManagerThread*, but deeming it as the core of the program wouldn't be wise, since all information passes through the *MainActivity*, and then the *LoggerThread*. Therefore, it could be said that there is no class or part of the program above another. However, as the *MainActivity* holds the GUI, killing this thread will start a cascade effect, stopping every ”worker” threads[6].

When the application is first run, the user is presented with the interface in figure 7.2. After choosing the parameters (or leaving the default mode) and pressing the ”Set plan & start test” button, a *toast*[7] saying that discovery has started will appear in the screen. If the selected mode is no other than ”initial discovery”, the screen will not show more information, unless new devices are connected by the current discovery.

Unbeknownst to someone watching the screen, however, several processes have been running in the background since the inception of the application, and even more after the ”Set plan & start test” button has been pressed. The battery and CPU monitoring threads have uninterruptedly been waiting for battery-change broadcasts and reading the /proc/stat file respectively. Messages from those two threads are already arriving to the main process, where they are being redirected to the logger thread. Therefore, even if no item in the screen has been touched, the log files in the storage of the smart-phone are already being populated.

Right after the plan is set, the *BTManagerThread* starts one or more discoveries (not simultaneously, of course) through the device’s Bluetooth adapter, and informs to the *MainActivity* whenever it does so. When the adapter finds a new device, a broadcast is caught by the *BTManagerThread*, and this sends it to the *ArduinoPlannerThread*. Depending on the ”connection time” chosen, the *BackgroundThreadDispatcher* will be called instantly or at a later time. Either way, if the connection is opened successfully, the *BTManagerThread* will receive a message containing the newly started thread’s instance.

At this time, there could possibly be a pool of *ArduinoThread*s running and carrying communications of each own, and simultaneously be sending the received messages to the *MainActivity* directly. Should one of these threads lose connection with its Arduino device, a notification would arrive at the *BTManagerThread*, and this would act accordingly (i.e. kill the disconnected *ArduinoThread*).

There are three possible messages that each *ArduinoThread* can send to the *MainActivity*: the ping response times, the ”STRESS DATA” sizes, and errors (corrupt messages).

In the figure 7.3 the data flow between classes is depicted.

[6]Android refers to the GUI as the main thread (the first that is run), while the other threads are taken as workers.

[7]In Android, a toast provides simple feedback about an operation in a small popup for a short period of

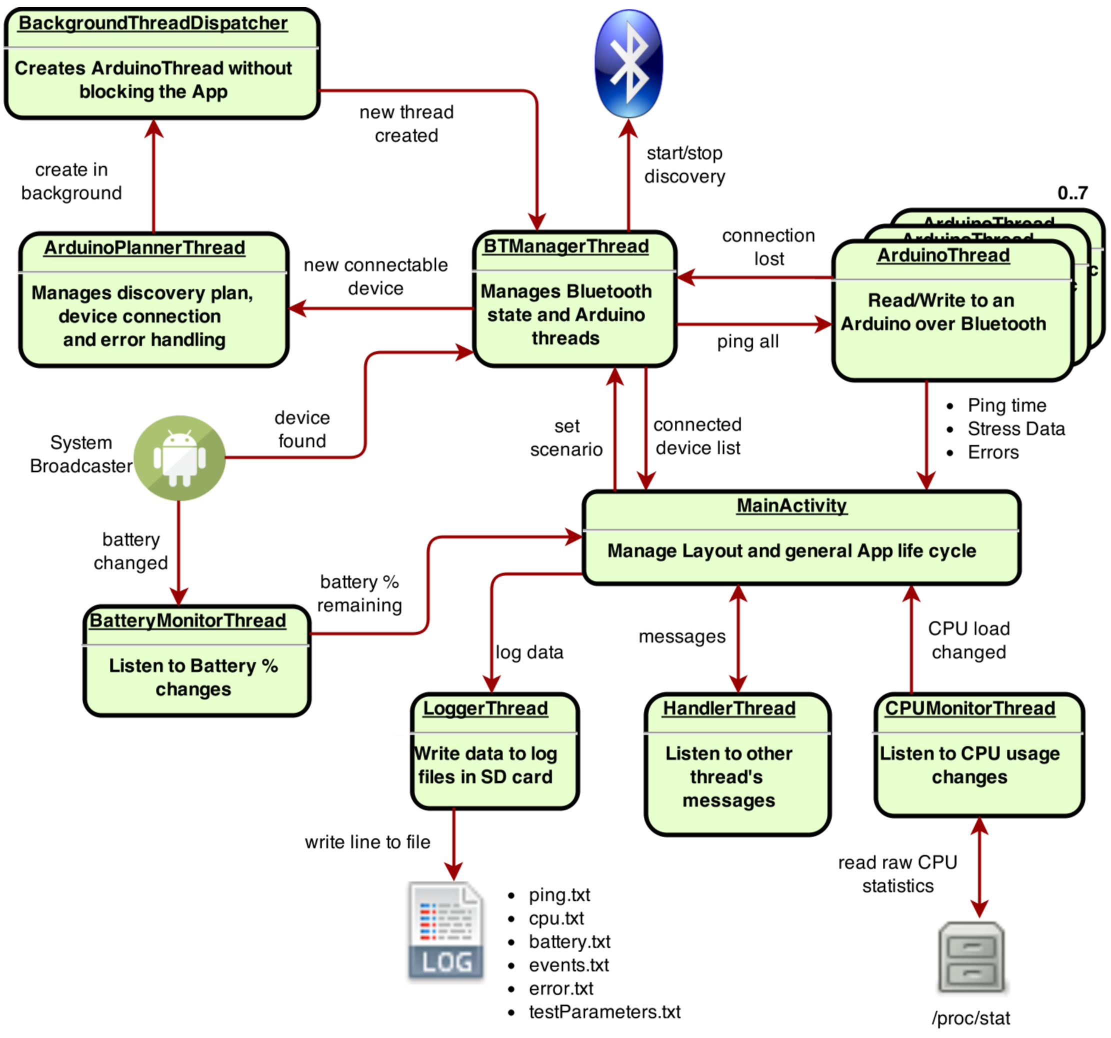


Figure 7.3: Data flow diagram.

7.3 Arduino

The Arduino application is dependent of the wiring between the Arduino board itself and the DF-Bluetooth module. Therefore, the wiring design should be made prior to the program.

7.3.1 Wiring

The way to make use of the DF-Bluetooth module through Android is simple, since only two things have to be made: the first is to power the module, and the second is to connect the output of the module to the input of the Arduino board, and vice-versa with the input of the module. As previously mentioned in section 4.2.3, the Arduino UNO boards do not have any extra dedicated serial pins—other than the RX and TX pins, which are shared with USB serial connections—. Therefore, the wiring scheme will be slightly different for the UNO and MEGA boards. Note that this difference implies different programs for each type as well.

In the case of the Arduino UNO board, digital pins 2 and 3 have been selected as complementary RX and TX pins, so these will be the input and output pins, respectively. As for the MEGA board, the RX1 and TX1 pins (digital pins 19 and 18, respectively) have been selected as input and output.

From the two possible power outputs in the Arduino boards (3 and 5 Volts), the 5V pin is going to be connected to the VCC pin of the DF-Bluetooth module. In addition to this, one of the GND (ground) pins is going to be connected to the GND pin of the module.

Figure 7.4 shows the wiring scheme for the Arduino UNO board with the DF-Bluetooth module, while figure 7.5 shows the equivalent wiring for the Arduino MEGA. The colour of the cables have the following meaning:

- Red: power (5V)
- Black: ground (*gnd*)
- Yellow: from the output (TX) of Arduino to the input (RX) of the Bluetooth module (data goes from the Arduino to the module).
- Green: from the output (TX) of the module to the input (RX) of Arduino (data goes from the module to the Arduino)

The wiring setup explained and designed until now should be sufficient for our purposes. However, knowing what has previously been stated about the Arduino boards not

time.

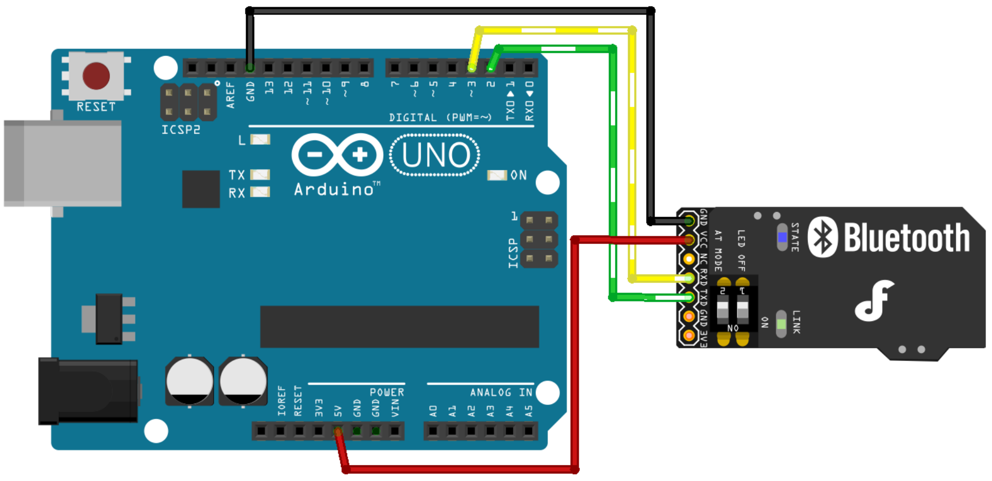


Figure 7.4: Wiring of the Arduino UNO board and DF-Bluetooth module.

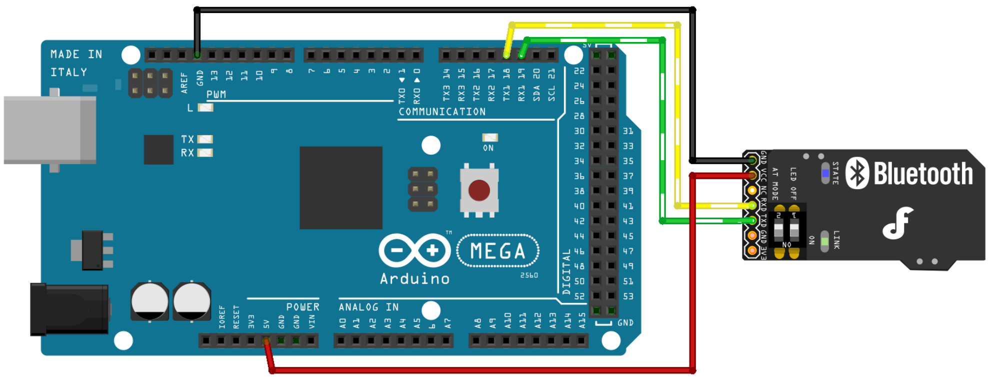


Figure 7.5: Wiring of the Arduino MEGA board and DF-Bluetooth module.

having any built-in capability to cut the power off of their 5V supply[8]—nor do the DF-Bluetooth modules have a toggling option for powering them ON & OFF—, there may be a problem with the performance tests' "progressive" scenarios.

Taking into account that the Bluetooth modules **should** be able to be automatically turned ON & OFF (see the performance tests' *other variables* section in 5.2), an external mechanism is required.

The solution to this problem is solved by using a *tri-state buffer*[9] (technically, it's a "power driver"). In essence, this component acts as a power switch. The precise way to use it on a par with an Arduino and a DF-Bluetooth module is explained in the development chapter (section 8.4). The inclusion of this module in the wiring is depicted in pictures 8.3a and 8.3b for the UNO and MEGA boards, respectively.

7.3.2 Application

The Arduino application consists of a single file, but since different Arduino boards are being used, two separate but almost identical files are required.

The fact that two different Arduino boards are being used would not matter if the Arduino UNO had some extra dedicated serial RX and TX pins. The ones it has could be perfectly be used to communicate the board with the DF-Bluetooth antenna, but then the board could not be connected to the computer by the USB, as previously mentioned. To work around this problem, a library from Arduino that simulates the serial pins' behavior by software was used[10]. Indeed, this is the reason why the code for the Arduinos is divided into two files, since everything but the Bluetooth-serial related code is identical.

As any other Arduino program, two main parts can be identified: the setup block and the looping loop. The setup block is called only once, just when the board starts or is rebooted[11]. During this block, the board is configured to open the required serial ports at a specific baud-rate (the DF-Bluetooth module's default, 57600 bits/s). Additional configurations like setting the Bluetooth module's power to null, or initializing some clock parameters are also done in the setup block.

In the looping block, the program will be waiting for incoming data from two sources: the message-bytes coming from the Bluetooth module, and the serial-commands that are

[8] See section 6.2.2 in the analysis chapter for more information.

[9] In section 4.2.3, a detailed explanation about tri-state buffers can be found.

[10] The library in question is *SoftwareSerial*.

[11] Any Arduino board can be physically rebooted by pressing the "RESET" button in the board. When a serial connection is established with a board, it automatically reboots itself.

sent to the board directly by serial (through the USB cable). In addition to this, if a special flag is activated, the board will write the same ”STRESS-DATA” message to both the serial port (USB) and the DF-Bluetooth’s RX port.

The program’s state is managed by means of some control flags, which are activated when specific commands are sent to the board by serial (USB). Below are explained the implications of those flags, and figure 7.6 shows the state machine for the Arduino application.

”p” (activates the POWER flag): The DF-Bluetooth module will be powered on, and the program will echo all incoming data from it.

”f” (activates the FINALIZE flag): The DF-Bluetooth module will be powered off, and the program will cease to send any data. This command sets the device on its initial state, but stores the size of the ”STRESS-DATA” message.

”s” (activates the STRESS-DATA flag): The program will send a ”STRESS-DATA” message in each loop.

”m” (activates the MUTE flag): The program will cease to send ”STRESS DATA” messages, but will keep echoing all incoming data from the DF-Bluetooth.

”i” (briefly activates the INCREMENT flag): The size of the ”STRESS-DATA” message is incremented in one unit (to be defined, but most surely 1, 10, or 100 bytes)

7.4 Arduino to Android communication

The design for the message format explained below is inspired on [hxm, 2010] by *Zepyr Technology*.

Figure 7.7 shows the format of the messages sent between the Android application and Arduino devices. Below are explained the meanings of each field, as well as the possible values for some of them[12].

1. The STX field (Start of Text) is a standard ASCII control character (0x02) and denotes the start of the message. Although it is not guaranteed that this will not appear again within a message, it gives some delimiting to the frame and therefore a start character to search for when receiving data.
2. The Message ID uniquely identifies each message type and is in binary format. For PING messages, the ID is 0x26, and for STRESS-DATA messages, 0x27.
3. The sequence number is an integer that identifies the ordinal number of the message, and it depends on the message ID. Valid values range from 1 to 100.

[12]Note that hexadecimal numbers are prefixed by ’0x’.

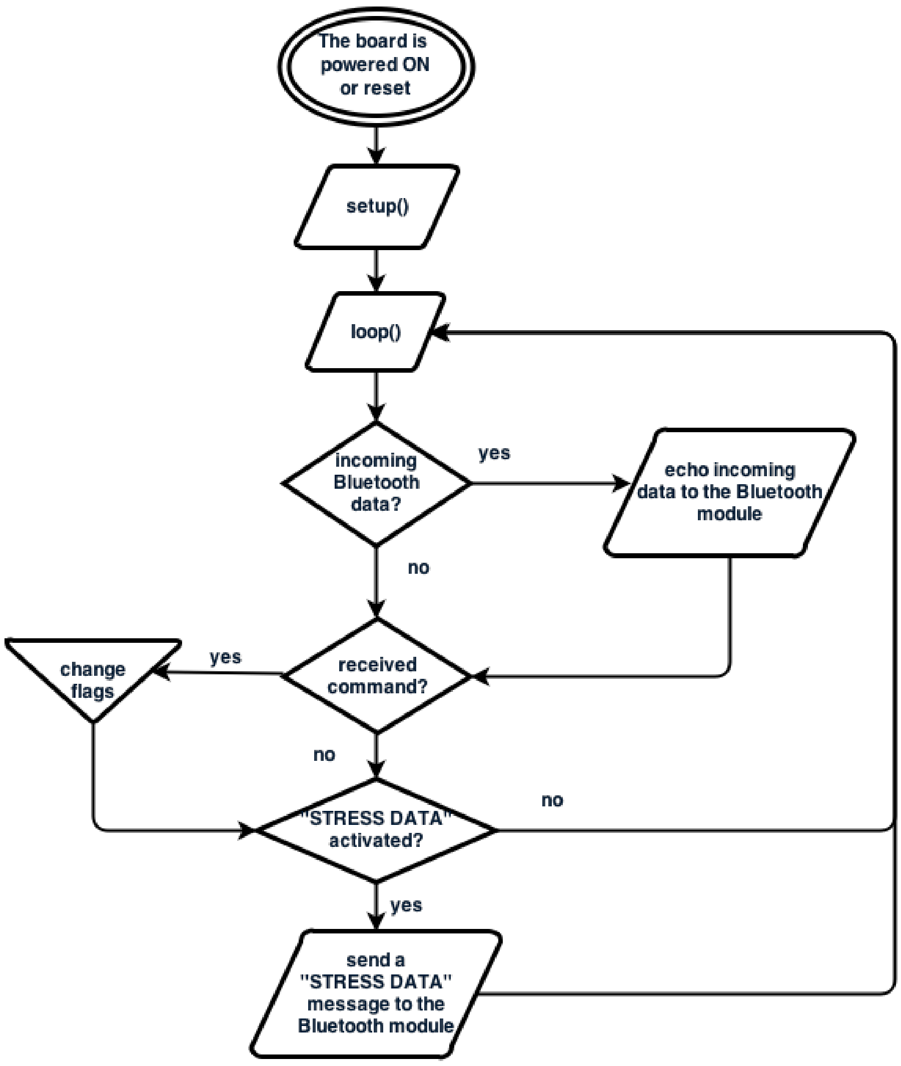

The board is powered ON or reset
setup()
loop()
incoming Bluetooth data?
yes
echo incoming data to the Bluetooth module
no
received command?
yes
change flags
no
"STRESS DATA" activated?
no
yes
send a "STRESS DATA" message to the Bluetooth module

Figure 7.6: State machine for the Arduino devices.

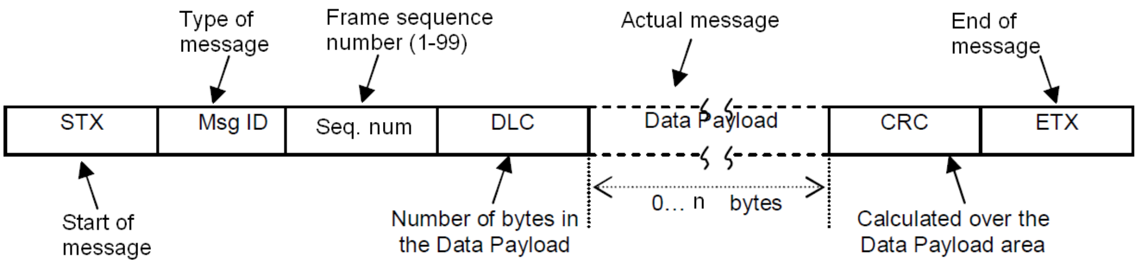


Figure 7.7: Format of the messages sent between the Android application and Arduino devices.

4. The Data Length Code is used to specify the number of bytes within the data payload field of the message. It is a *long* number formed by four bytes. Valid values range between zero and 2^{32} (inclusive). For PING messages it should be 0x00.
5. This field contains the actual data sent between the local and remote units and can contain anywhere between zero and 2^{32} bytes of data. The number of bytes in this field is dictated by the DLC field.
6. CRC stands for Cyclic Redundancy Check[13]. A 4 byte CRC is used, and its value is based on the remainder of a polynomial division (CRC-32) of the payload field.
7. The ETX field (End of Text) is a standard ASCII control character (0x03) and denotes the end of the message.

The minimum number of bytes for a message is:

$$1_{STX} + 1_{Msg.ID} + 1_{Seq.Num.} + 4_{DLC} + DLC_{payload} + 4_{CRC} + 1_{ETX} = 12 + DLC$$

In the Android application a message is instantiated whenever a PING is sent to a target device, or whenever a message (regardless of its type) is received. For both purposes, the *ArduinoMessage* class is used. On the other hand, in an Arduino device a message is only created when sending a ”STRESS-DATA” message. This is so because when an Arduino receives a PING message, it is echoed immediately and no more attention is given to it.

7.5 Python Scripts

The reasons for using Python are twofold. On the one hand, it is a clear and easy to learn language, yet it provides powerful plotting capabilities. Having no previous experience with other languages, it appeared as the most convenient tool to plot the tests' results. On the other hand, if provides the opportunity of automating the changes in the scenario (that is, the switching of the Bluetooth antennas connected to the Arduinos, among other things) through serial port protocol communications.

For these reasons, and the fact that I already had some experience with this language, regardless of the context, Python was chosen.

Plotting

The motivation behind using the plots comes from a need to simplify the process of examination for the tests' results. The visual representation would not only help to inter-

[13] A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data.

pret the results in a way other than reading the logged data, but also to identify unexpected anomalies in the glimpse of an eye.

In a few words, the plotting scripts take data from the log files, make calculations if needed (e.g. to obtain the throughput), and show the results in a graphic diagram window. This window lets the user customize some properties of the diagram, and offers the possibility of exporting it to an image file. A static image could be automatically generated and sent to the computer by code also, but this would not give the opportunity to change the zoom-scale or other parameters in real-time.

The raw data is logged in a way which eases readability to the python scripts. Each data type is stored in a separate file, each containing different number of values. However, all the log files share something in common: every entry or line of the logs is preceded by a time-stamp of when the reading was done. These stamps are the time in when the value of the data is first known, and is given in Linux or *epoch* time[14]. There are log files for battery percentage changes, CPU usage, ping times, received data, and events. The contents and structure of each data type is explained below:

- Battery percentage: the readings are logged in *battery.txt*. Each entry consists of a timestamped float value, ranging from 0 to 100. A value reading is done when the Android application receives a broadcast intent from the OS. These broadcasts are not periodic, so the time-space between readings can vary significantly.
- CPU usage: the readings are stored in *cpu.txt*. Each entry consists of a timestamped float value, ranging from 0.0 to 1.0. Readings are performed once every second.
- Events: an event is a string value indicating that a relevant phenomenon has happened, such as "discovery-started".
- Received data: the readings are stored in *data.txt*. Each entry is preceded by a timestamp, followed by a string value that indicates the target sensor from which a ”stress” data message was read, and then by an integer indicating the number of bytes that were read.
- Ping values: the readings are stored in *ping.txt*. It is very similar in structure and content to the received data, but with another integer value at the end of each entry. This last value is the ping or elapsed time since the packet was sent and received.

During the process of storing the data, no information is lost nor filtered. What’s more, in order not to taint the readings’ timestamps, the Android application does not perform

[14]Unix time (aka POSIX time or Epoch time), is a system for describing instants in time, defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970

any calculation over the data values, since this would consume more resources than required.

Therefore, the data in the log files is precise in the sense that it has been collected in the most efficient way possible with the resources at hand. This does not mean that the values are a 100% reliable. The battery level, for instance, depends on the accuracy of the phone's internal sensors and Android's algorithm to calculate it. It would be much more appropriate to measure the electrical current that drains the battery with a multimeter, but there is no easy way of logging the readings for later use. In addition to this, depending on the smartphone the battery is not always easily accessible, as it is sometimes fixed or soldered.

The ping times may be tainted as well, since there is no way of measuring the exact real time in which the Android device sends or receives the packet through the Bluetooth antenna. If the device is busy performing other tasks, the packet may be stored in a buffer for some time, and alter the final ping value. However, this is not a problem, since what this research is measuring is not the real value of round-trip delay time but rather the smart-phone's rate of attainment in the communications.

Serial Communications

The script presents the user with a multiple-choice menu (all within a command-line terminal), with three options:

1. See available serial ports: shows the serial ports available at this moment (most will be USB devices).
2. Open serial ports: tries to connect to all the available serial ports belonging to an Arduino. Once a connection to a port that has been established, command ”f” is sent to it, to reset its state.
3. Begin test-script: executes a block of code that sends the commands explained in section 7.3.2 to the connected serial ports at specific time intervals.

The logic behind the script will be kept as simple as possible, and no error-checking will be necessary. The block of code that sends commands to the connected serial ports may or may not be an infinite loop. If the program ends in an unexpected way, no special commands needs to be sent to the connected serial ports.

The block of code containing the test-script will be slightly changed for each test case.

8. CHAPTER

Implementation

This chapter explains some technical details about the system developed. Some minimum or general knowledge on Eclipse, Git, Android, Arduino, Python and other technologies is taken for granted.

8.1 Apache 2.0 license

The source code of this project is developed and published under the *Apache License, Version 2.0* by the Apache Software Foundation. The project's source contains a copy of this license, and the original Apache license template can be found in the link below.

https://www.apache.org/licenses/LICENSE-2.0.html

8.2 The Git repository

Although the development of the project has been done entirely in a PC workstation, all the files required for the system to be developed, compiled and executed are within a local Git repository. At the same time, this local repository is connected to another one in GitHub. This means that all[1] the source files of the local Git repository are stored on the cloud.

Originally, this repository was used for a sample application made by Borja Gamecho and I during our previous collaboration, and was later used as the basis for the whole

[1]*All* the files that have been committed or uploaded

system hereby developed. The owner of the account linked to the repository is Borja, while I have full access for contributing to it as well.

The reasons for keeping the project in this repository and not a new one are twofold: first, because at an early state of the project, moving and reconfiguring this setup would have taken much time, as I did not have much experience with Git and did not want to mess things up. Doing so would have taken an overhead of work for a purpose that was not necessary. Second, and more importantly, it was decided that in order to guarantee that Borja could have access to it in the future, I would keep a copy for myself, while updating his repository.

The project's entire source code is available in a public repository at *GitHub.com*, and it can be easily accessed following the link below:

https://github.com/bgamecho/arduino2android

8.3 The Android application

The development of the Android application is entirely done through the Eclipse-ADT framework. The project's workspace directory is located inside a *Dropbox* folder to guarantee that the files are automatically backed up to a secure location. Apart form this, the whole project is also part of a Git repository within *GitHub.com*, as aforementioned.

The project is structured as any other Android program. Apart from the auto-generated files and directories, there are three main folders in which the source code for different components can be found:

src Contains the main source code, where all the *java* files are gathered.

res Contains extra resources, such as layout descriptions (in *XML*), or general *String* values.

Assets Contains files related to the overall system. This folder is special, as no files required for the Android application are within it.

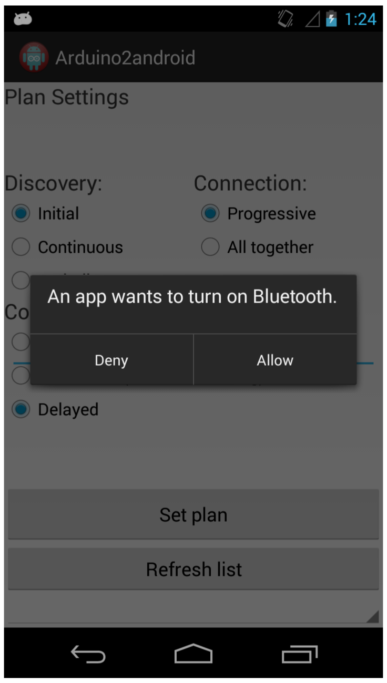
The Assets folder contains the source code of the Arduino programs, as well as some python scripts. Some tests' results used for debugging purposes can also be found in here.

In the following subsections, some of the most meaningful or interesting pieces of the application are explained, as well as the process of developing them.

8.3.1 Starting the application

When executing the application, *MainActivity* is started. During its creation, it sets the GUI layout and initializes the other threads (*BTManagerThread*, *BatteryMonitorThread*, *CPUMonitorThread* and *LoggerThread*), and then starts them all. When those classes are instantiated, a reference to the *Handler* of the main thread (*MainActivity*) is passed along to each one of them. This will give them the possibility of sending messages to the *MainActivity*.

When *BTManagerThread* is created, it checks whether the Android device has a Bluetooth adapter or not—i.e. checks if the phone has a Bluetooth antenna—. If it does, it checks if it is currently enabled. If it is not, it requests the user to enable it, and Android will automatically present the user with a pop-up (an Android *Intent* produced by the call in listing 8.1) as shown in figure 8.1 requesting for permission. Once the user has granted (or denied) permission, the *MainActivity* will receive the *Intent*'s result. At this point, if Bluetooth is disabled (the user has not granted the permission), the application is immediately finished.



Discovery: [✓ Initial] Continuous

Connection: [✓ Progressive] All together

An app wants to turn on Bluetooth.

[✓ Delayed]

Figure 8.1: Bluetooth turning on request.

Due to Android's usability security and policy, there is no possible mechanism to turn on Bluetooth without the user's knowledge and permission. In case this changes in the future (in an updated Android API), this procedure could be automated.

Listing 8.1: Bluetooth enable request.

```
Intent enableBtIntent = new
   Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

((Activity) mainCtx).startActivityForResult(
    enableBtIntent,
    REQUEST_ENABLE_BT);
```

8.3.2 Communicating with *BTManagerThread*

The *BTManagerThread* class contains two objects with which it gets information from other classes or components of the Android application. The first one is *btHandler*, and instance of Android's *Handler* class. This object listens for messages sent explicitly (and only) to the *BTManagerThread* from other classes. There are a total of five different messages that can arrive, each with a unique identifier, and possibly served with additional parameters. The messages may only serve to inform about an incident, or to request for something to be done. The possible message identifiers and their implications are:

- MESSAGE_SET_SCENARIO: Set the scenario and start a test.
- MESSAGE_BT_THREAD_CREATED: An *ArduinoThread* thread has been initialized and is running.
- MESSAGE_ERROR_CREATING_BT_THREAD: There's been an error while instantiating an *ArduinoThread* thread.
- MESSAGE_CONNECTION_LOST: Connection was lost with an Arduino device.
- MESSAGE_SEND_COMMAND: Send a command to an Arduino device.

The other object that enables communicating with other components is *myReceiver*, an instance of Android's *BroadcastReceiver* class. This object listens to broadcasted *Intents* from Android (all the *Intents* that are received have automatically been created and sent by the OS) and acts accordingly. There are two *Intent* types, according to their origin: *BluetoothDevice* and *BluetoothAdapter*. For the former, all received *Intents* contain an instance of a *BluetoothDevice* object that corresponds to the message's inception. There's a total of five *Intents* that the receiver listens to:

BluetoothDevice

- ACTION_FOUND: It means that a Bluetooth device has been found by a discovery, and an instance of it (*BluetoothDevice* object) is attached to this *Intent*.

- ACTION_ACL_CONNECTED: A connection has been established with a Bluetooth device.
- ACTION_ACL_DISCONNECTED: The connection with a Bluetooth device has been lost, and an instance of it (*BluetoothDevice* object) is attached to this *Intent*.

BluetoothAdapter

- ACTION_DISCOVERY_STARTED: A discovery has started.
- ACTION_DISCOVERY_FINISHED: The ongoing Bluetooth discovery has finished.

8.3.3 The control logic

As previously explained in section 7.2.1, the control logic implemented inside *BTManagerThread* is quite intricate. Basically, the requests for discovery can done by four different triggers:

- MESSAGE_BT_THREAD_CREATED: triggered when an *ArduinoThread* has successfully been created and a connection with a Bluetooth sensor has been established.
- MESSAGE_ERROR_CREATING_BT_THREAD: triggered when the creation of an *ArduinoThread* has encountered a problem, probably for not being able to establish a connection with a target Bluetooth sensor.
- ACTION_DISCOVERY_FINISHED: triggered when an ongoing Bluetooth discovery has finished.

Additionally, when the time since the last discovery exceeds a predefined time interval, a discovery is also requested just in case one of the previous triggers did not work properly.

All discovery requests are sent to a function where the plan parameters are checked, and where it is decided whether to carry the discovery out or not. Listing 8.2 shows the contents of that function.

8.3.4 Communications with the Bluetooth devices

When the *BTManagerThread* finds a connectable Arduino—by checking its name, identifier or MAC address—, an *ArduinoThread* is created for that device alone. The very first thing this thread/class does when instantiated is to open a socket with the target device it intends to connect to. Listing 8.3 shows the function for this process.

Listing 8.2: Function to check the plan decide to start discovering.

```java
private final BluetoothAdapter _BluetoothAdapter;
private final long discoveryInterval = 30000; //milliseconds
private long lastDiscoveryTimestamp;
public Handler btHandler = new Handler() {...}
// List of devices waiting to be connected:
private Map<String,BluetoothDevice> waitingToBeConnected;
//Test parameters: (instanciated when this class is created)
private int discoveryPlan, connectionMode, connectionTiming;

private void startDiscoveryIfPossible() {
    switch (discoveryPlan) {
        case INITIAL_DISCOVERY:
            // Do nothing (an initial discovery has already been done)
            break;
        case CONTINUOUS_DISCOVERY:
            if(connectionTiming != IMMEDIATE_WHILE_DISCOVERING_CONNECT){
                if(!(waitingToBeConnected.size() > 0)){
                    _BluetoothAdapter.startDiscovery();
                    lastDiscoveryTimestamp = System.currentTimeMillis();
                }
            }else{
            _BluetoothAdapter.startDiscovery();
            lastDiscoveryTimestamp = System.currentTimeMillis();
            }
            break;
        case PERIODIC_DISCOVERY:
            Runnable postRunnable = new Runnable() {
            @Override
            public void run() {
                _BluetoothAdapter.startDiscovery();
                lastDiscoveryTimestamp = System.currentTimeMillis();
            }};
            btHandler.postDelayed(postRunnable, discoveryInterval);
            break;
    }
}
```

Listing 8.3: Function to open a socket with a Bluetooth device.

```
private BluetoothDevice myBluetoothDevice; //Instanciated previously
private BluetoothSocket _socket;
private InputStream _inStream;
private OutputStream _outStream;

public void openConnection() throws Exception{
    Method m = myBluetoothDevice.getClass().getMethod(
        "createRfcommSocket", new Class[ ] {int.class});
    _socket = (BluetoothSocket) m.invoke(myBluetoothDevice, 1);
    _socket.connect();
    _inStream = _socket.getInputStream();
    _outStream = _socket.getOutputStream();
}
```

Once the socket is successfully created, the thread will enter in an infinite loop, and in each iteration it will check for new data arriving to the input stream (*_inStream*).

Figure 8.2 shows the dialog for entering a device's PIN code when requesting a connection establishment with it.

8.3.5 Battery monitoring

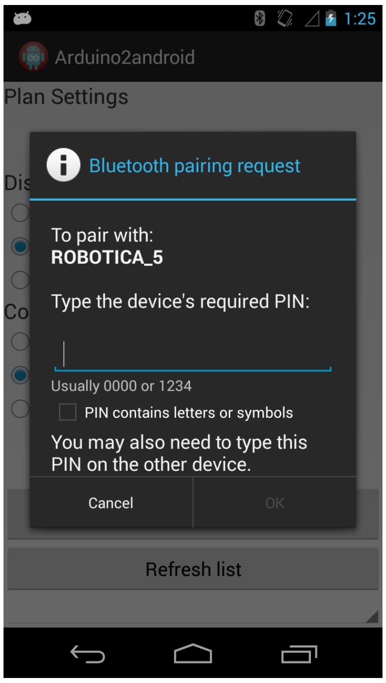
The *BatteryMonitorThread* is kept as simple as possible. Its only commitment is to listen for the general purpose ACTION_BATTERY_CHANGED *Intent* thrown by the OS (Android) whenever it detects a change in the Battery level. In order to receive such messages, a registration like the one in listing 8.4 needs to be made. Listing 8.5 shows the code for retrieving the Battery level value from said *Intent*, and how the *MainActivity* is notified through its *Handler*.

8.3.6 CPU usage monitoring

The mechanism to calculate the CPU usage is inspired on a question from the *StackOverflow.com* forums[2].

The Android operating system is based in Linux Kernel, although it is not identical (Google has made several modifications). Still, there is plenty of features that both Android and Linux share in common, such as the *proc* file-system, which provides an inter-

[2]The original forum-thread can be found here.



To pair with:
ROBOTICA_5

Type the device's required PIN:

Usually 0000 or 1234

☐ PIN contains letters or symbols

You may also need to type this PIN on the other device.

Figure 8.2: Pairing request for a Bluetooth device.

Listing 8.4: Registration for ACTION_BATTERY_CHANGED *Intent*.

```java
private Context mainCtx; //Instanciated previously

IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
batteryStatus = mainCtx.registerReceiver(myReceiver, ifilter);
```

Listing 8.5: Calculus of the Battery level.

```java
private Handler mainHandler; //Instantiated previously

private final BroadcastReceiver myReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        switch (action) {
            case Intent.ACTION_BATTERY_CHANGED:
                int level = intent.getIntExtra(
                    BatteryManager.EXTRA_LEVEL,
                    -1);
                int scale = intent.getIntExtra(
                    BatteryManager.EXTRA_SCALE,
                    100);

                float batteryPct = (100*level) / (float)scale;
                long timestamp = System.currentTimeMillis();

                Message sendMsg = mainHandler.obtainMessage(
                    MainActivity.MESSAGE_BATTERY_STATE_CHANGED,
                    batteryPct);
                Bundle myDataBundle = new Bundle();
                myDataBundle.putLong("TIMESTAMP", timestamp);
                sendMsg.setData(myDataBundle);
                sendMsg.sendToTarget();
            break;
        }
    }
};
```

Listing 8.6: Content of the */proc/stat* file.

```
cpu  3965 760 6647 1920 4138 0 96 0 0 0
cpu0 3965 760 6647 1920 4138 0 96 0 0 0
intr 58877 123 236 0 0 157 0 3 0 1 0 0 0 4328 0 0 284 1926 13349  [...]
ctxt 173935
btime 1408899808
processes 2754
procs_running 4
procs_blocked 0
softirq 73912 0 27372 334 989 12857 0 198 0 22 32140
```

face to the kernel's data structures [lin, 2014]. Commonly mounted at */proc*, the */proc/stat* file under this file-system is of most relevance when dealing with CPU measurements.

As Nitsch (2003) thoroughly explains, in the */proc/stat* file there are various pieces of information about kernel activity, and all the numbers that appear in this file are aggregates since the system first booted [Nitsch, 2003]. Listing 8.6 shows the content of the */proc/stat* file for a specific *Linux* machine[3].

The numbers at each line correspond to the identifier at the beginning of the line. For example, the numbers at the first line ”cpu” identify the amount of time the CPU has spent performing different types of work, while the number at the ”procs_running” line means the number of processes running right now in the different CPUs. Time units are in USER_HZ or Jiffies[4].

The only relevant data to calculate the CPU usage is within the ”cpu” line (the first line). There may be one or more ”cpu#” lines, which correspond to each individual core of the processor. The first line is the sum of them all. The meanings of the columns for Android's particular *proc* file-architecture are as follows (from left to right):

user Normal processes executing in user space (time spent in user mode).
nice Niced processes executing in user space (time spent in user mode with low priority).
system Processes executing in kernel space (time spent in system mode).
idle Twiddling thumbs (time spent in the idle task).
iowait Time waiting for I/O to complete.
irq Time servicing interrupts.

[3]The content of */proc/stat* varies with architecture.
[4]In a Linux operating system, HZ (”hertz”) is a variable that measures how often the CPU can switch between tasks.

softirq Time servicing softIRQs.

steal Stolen time (the time spent in other operating systems when running in a virtualized environment).

guest Time spent running a virtual CPU for guest operating systems under the control of the Linux kernel.

gest_nice Time spent running a niced guest (virtual CPU for guest operating systems under the control of the Linux kernel).

Note that **Steal**, **Guest** and **Guest Nice** times are all spent in virtualized guest operating systems, and that usually these values are zero in Android [Bui, 2012].

Now, as Jose Martinez (2009) states, every time */proc/stat* is read it gives the number of slices each processor has passed doing what. As aforementioned, however, all values are counted since the OS is running. Therefore, a single reading will not tell us the current CPU usage, but the total absolute usage of the CPU in each type of task. In order to know the ”current” CPU usage, we have to make an initial reading, store it, and after an elapsed time interval, make another. The difference between the two readings (calculated as the subtraction for each number or column, or in other words, the last reading minus the previous one) will give us the CPU's usage during that time interval [Martinez, 2009].

Still, that difference is raw data, and in order for us to interpret it, a formula needs to be applied. If we divide the ”real” CPU value's difference (ignoring the idle task's slices) by the total (taking idle slices into account), this will give us a number between 0 and 1 (0 being no CPU usage and 1 being a 100%).

The formal calculation is done as follows (bear in mind that ”1” stands for the first reading of the */proc/stat* file, while ”2” for the second):

$$CPU_1 = user_1 + nice_1 + system_1 + iowait_1 + irq_1 + softirq_1$$
$$CPU_2 = user_2 + nice_2 + system_2 + iowait_2 + irq_2 + softirq_2$$
$$CPU_{TOTAL} = \frac{CPU_2 - CPU_1}{(CPU_2 + idle_2) - (CPU_1 + idle_1)}$$

Of course, the time interval's length affects the reading, and hence the final calculated value. Reading spans less than 0'5 seconds have shown to be quite chaotic, while intervals near a second seem to be quite accurate. Listing 8.7 shows the function in *CPUMonitorThread* class that calculates the CPU usage. Note that if there has been an error, ”-1” is returned, and the calling process should decide what to do.

Note: there is a possible bug in Android's kernel that may slightly corrupt or even break any kind of load/CPU usage monitoring based on this technique. While developing the application in this project, the function that calculates the CPU usage yielded negative

Listing 8.7: Calculating the CPU usage from the */proc/stat* file.

```
private long cpu1, idle1; //Values from a previous reading

private float readUsage() {
    try {
        RandomAccessFile r = new RandomAccessFile("/proc/stat", "r");
        String load = r.readLine();
        load = .readLine();
        r.close();

        String[] toks = load.split(" ");

        long idle2 = Long.parseLong(toks[4]);
        long cpu2 = Long.parseLong(toks[2]) + Long.parseLong(toks[3])
                    + Long.parseLong(toks[5]) + Long.parseLong(toks[6])
                    + Long.parseLong(toks[7]) + Long.parseLong(toks[8]);

        float result = (float)(cpu2 - cpu1)
                        / ((cpu2 + idle2) - (cpu1 + idle1));

        idle1 = idle2;
        cpu1 = cpu2;
        return result;

    } catch (IOException ex) {
        ex.printStackTrace();
    }

    return -1;
}
```

values, which by logic can't be possible. At the time of this writing, this issue has been identified at least once in the ”Issue Tracker” of the *Android Open Source Project* (see `https://code.google.com/p/android/issues/detail?id=41630`). So far, the only measure that can be taken is to discard those illogical readings.

8.4 Building the Arduino boards

Once the design was done, the process of building or ”wiring” the Arduinos with the DF-Bluetooth modules was pretty straightforward. There were plenty colored, short wires in the laboratory, and enough breadboards[5], so no soldering was needed at all—an advantage, since learning how to do it properly, and acquiring the resources could have taken some extra time—.

As previously stated during the design chapter (see section 7.3.1), the initial wiring scheme was quite simple. However, there was the lack of a mechanism to programmatically switch the Bluetooth modules' power. The proposed solution was the use of a *tri-state buffer*, an component that can provide a functionality—among others—similar to a switch. The tri-state buffer used in this project is an ”L293D”, the ones built by *ST Microelectronics* (www.st.com) specifically[6]. This model provides capabilities far beyond than what is needed for this project, and there probably are others better suited. However, it was the best choice due to its current availability in the laboratory, and its reduced cost.

As explained in section 4.2.3, the way to use the L293D is not that complex. According to the pin-out configuration shown in figure 4.4 and the truth table from the device's manual recreated in table 8.1 (refer to [st2, 2003] for the whole specifications), a total of six particular pins need to be connected with the Arduino board and the DF-Bluetooth module. Always taking in mind which pin-out is which, the simplest way to achieve a switching mechanism is as follows:

VCC-1 5 volt supply for internal power of the L293D.

VCC-2 5 volt supply for powering the outputs 1 to 4 of the L293D.

GND Ground supply for internal power of the L293D.

Input-1 Normal input to the buffer (5V supply in our case).

Enable-1 Enabling digital signal. If high, **output-1** will give 5V. If low, **output 1** will be off (0V).

[5]A breadboard (or protoboard) is a construction base for prototyping of electronics.

[6]Credit for choosing this device goes to Borja Gamecho, whose knowledge on electronics is greater than mine

INPUTS		OUTPUT
1A	**EN**	**1Y**
H	H	H
L	H	L
X	L	Z

H = high level, L = low level, X = irrelevant, Z = high impedance (off)

Table 8.1: Truth table.

Output-1 Outputs 5V if **enable-1** is high). Else, outputs 0V.

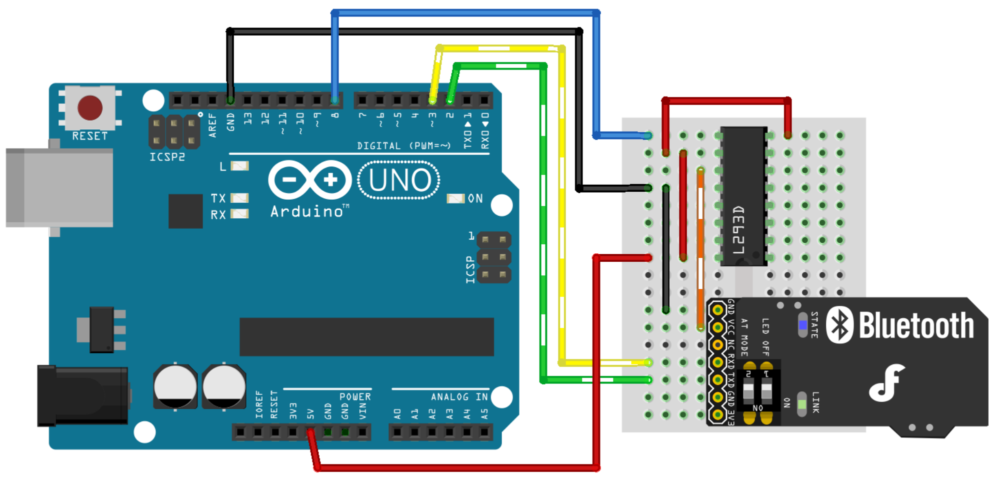
The enabling signal can be achieved with any of the digital pin-outs of the Arduino boards that are not being used for other purposes. In this project, digital pin 8 has been chosen, but it could have been any other. Figure 8.3a shows the final wiring scheme for the Arduino UNO boards, while 8.3b does the same for the MEGAs.

8.5 Plotting scripts

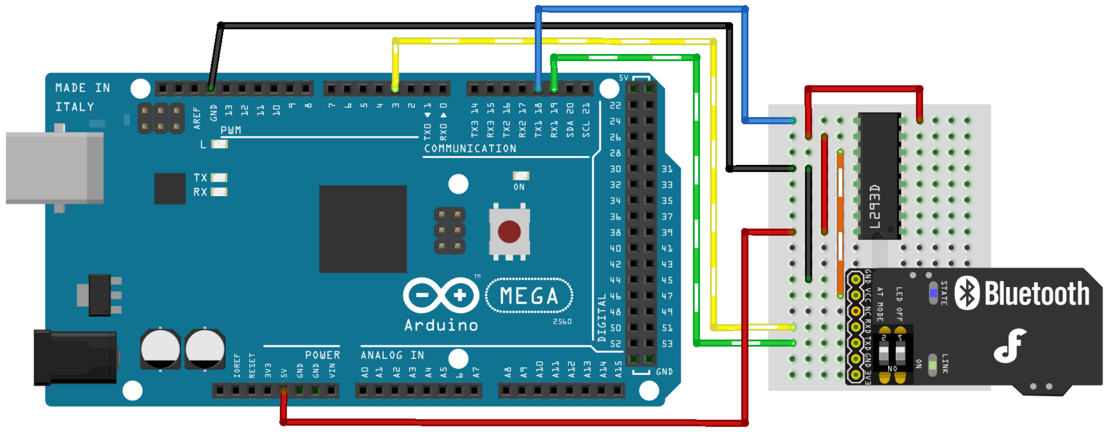
The graphical representation of the tests' results is not easy to predefine, as different tests may show diverse graphical outputs. The length of a test-log can vary in the time span the test was carried, as well as in the total number of entries in the log. For example, a test in which no Bluetooth devices have been connected will have no "ping" and "throughput" data to display. Additionally, a test done with just one sensor will be much smaller (in total entries) than another test done with seven sensors in a similar time span.

According to the number of entries in the log, these are from lower to bigger the possible cases:

1. Events
   (a) Initial discovery: less than 10 entries.
   (b) Periodic (30 seconds between): $\tilde{4}$ entries per minute.
   (c) Continuous: $\tilde{1}0$ entries per minute.
2. CPU values: 60 entries per minute
3. Battery values: 1 to 10 entries per each % point (~100 entries for the whole battery).
4. Ping: ~10 entries per second, per connected sensor.
5. "Stress data": ~10 entries per second, per connected sensor in "Stress" mode.



(a) Wiring of Arduino UNO and DF-Bluetooth with the L293D buffer



(b) Wiring of Arduino MEGA and DF-Bluetooth with the L293D buffer.

Figure 8.3: Final wiring schematics for the Arduinos, DF-Bluetooth module and the L293D buffer.

When plotting the results, the ranges of the different variables will need to be taken into account:

- Battery load (percentage): from 0 to 100%.
- CPU usage (float point): from 0.0 to 1.0 points.
- Ping time (seconds): from 0 seconds on (limitless in theory).
- Throughput (bytes/s): from 0 b/s to a theoretical maximum of 262.5 kB/s.

The plotting scripts are kept simple, and their execution is straightforward (no parameters need to be passed). There are four ”template” plotting scripts, one for each variable (battery, CPU, ping and throughput). These templates can be used to take a part of them and plot composed graphs. The path of the files to be processed is indicated in the code, so the script should be changed for each different log-file that wants to be plotted.

9. CHAPTER

Testing

9.1 Debugging

Instead of designing and performing formal unitary tests for the different components of the testbed—something that would have taken a big overhead—, a thorough debugging of the applications has been made. The programs for the Arduino boards have been the easiest to verify, as their code is quite small. Since the Python scripts are interpreted at runtime, the method followed to check their proper realization has been to perform individualized proves with a reduced set of preliminary log-files or data. The debugging process of the Android application has taken the most effort.

9.1.1 Android debugging

With the use of *adb*[1], the debugging has been made live (with no virtualization). Everything from the creation of threads to the reading of incoming Bluetooth messages has been thoroughly debugged and tested.

Thanks to the tools provided by the Android development environment it was possible to identify the threads or classes that were less memory efficient. This enabled restructuring them so that the overall performance of the application was at its maximum.

[1] *adb* acts as a middleman between an Android device and the development system (Eclipse-ADT in this particular project).

9.2 Problems found

A few major problems found during the execution phase of the project are explained below.

9.2.1 The Bluetooth callback

Among the main issues found, the misbehavior of the DF-BluetoothV3 module is at the top. As previously explained in section 4.2.3, these antennas provide the Arduino boards with a Bluetooth powered serial communication capability. The problem was that the communications stopped at a given moment under no particular circumstances.

At the beginning of the development process, the Arduino sensors had a very simple behavior. By default, the antennas would not send any data, unless a serial ”start” signal was sent from a PC-terminal connected by USB. Once this ”start” command was received, the Arduino then sent some randomly generated data through Bluetooth, which could then be read by the Android application. Among other commands, the ”finalize” command stopped the sending of data. Up to this time everything worked like a charm. However, after receiving the ”finalize” command, and once the Bluetooth antenna ceased to send ”stress” data the, if the ”start” command was received, nothing happened.

I discovered that when many consecutive commands were sent from the Android application to the Arduino via Bluetooth, at first they did not arrive to the board, until the reset button of the board itself was pressed. This meant that the antenna stored the data somewhere in a buffer and did not release it until later. The same effect appeared if instead of resetting the board a random signal was sent to the Arduino board directly through a serial USB connection from a Computer.

The origin and solution to the problem could be manyfold. At first I thought it was an error induced by wrongly closing the stream sockets in the Android application, since every time the connection came to an end, a warning exception appeared. However, this warning seemed to be minor and known bug in the Bluetooth stack of Android.

After many trial and errors, the origin of the problem was not any clearer. Thinking that the origin was in the hardware, a workmate from the Egokituz laboratory lent his hand to find the issue. We got to use a *Hewlett Packard 1651A Logic Analyzer* and a *Fluke 175 True RMS* multimeter, and we discovered that one of the two (Arduino board or Bluetooth antenna) might malfunction due to the short voltage difference of the digital pin-out ports.

One of the possible solutions was to put the mode of the input pin on the Arduino board to high, since apparently it was at a lower voltage (3.5V) than the output pin (4.7V). Since this did not solve the problem, another approach I tried was to heighten the voltage of the Bluetooth module to put it on a par with the Arduino's. To achieve this, a CMOS device was tried, to no avail.

I found out that Android provided a sample ”Bluetooth-Chat” application among other examples in their repositories. I gave it a chance, and designed a prototype program for the Arduinos to communicate with this ”Bluetooth-Chat” application. To my surprise, Android's application worked flawlessly, even with the original wiring setup of the Arduino UNO and MEGA as well. This made it clear it was all a coding problem withing the Android application.

I analyzed the specific way in which Android's ”Bluetooth-Chat” application performed the connection with the target device, and how the input and output streams were retrieved. Architecture and class design differences aside, Android's application and mine were quite similar, except for one thing:.I was performing unsynchronized readings and writings from the I/O streams, and had not thought about the possibility of multiple threads trying to read from the same socket at the same time. After synchronizing the use of the socket, the problem was solved.

9.2.2 Opening the socket

As I myself struggled with it at the start, it seems that one of the most common issues that developers find when working with the Bluetooth protocol is how to create the connection and the socket. It seems that Android has made a clean break in order to keep security leaks happening because of Bluetooth.

As the developers of Zephyr HxM [hxm, 2010] say:

> There are some ’issues’ with the Bluetooth issues with some versions of Android, and with some specific devices. Ordinarily all you would have to do to create the Bluetooth socket is use the create call, specifying the UID of the Bluetooth service profile that will be used for the connection.

Initially, the workaround I used was to perform the so called "java reflection", as explained here. However, after further research on the matter, I found out that there was a better method (the one described in listing 8.3).

9.3 Graph plotting in Python

The first Python script for plotting was written without many problems. Some initial tests were done to verify that the output was the expected one, and corrections were made where needed (e.g. showing grid-lines, or scaling the X- and Y-axis). However, when the real benchmarking process begun, the size of the logged data was so large that the plotting scripts took much longer time than expected. Some improvements were done, but still, the time required for plotting larger log-files with one to two billion of entries is not acceptable. The fact that the computer used for the plotting purposes does not have much memory does not help either.

However, a possible solution to reduce the time required to plot a single graph has been identified. As stated in [mssaxm, 2013], there is a common error done by many programmers when reading large files, coined as ”slurping”. In essence, ”slurping” is the action of loading all the contents of a file in memory to have access to all the lines at the same time. The problem of course is that when a file is large enough, the machine’s free memory gets fully used. ”Slurping” is a wrong way of reading files for two reasons:

- It’s quite memory inefficient
- It’s slower than processing data as it is read, because it defers any processing done on read data until after all data has been read into memory, rather than processing as data is read.

If the processing and plotting of the logged files were done line by line, the Python script would surely create the graphs much faster.

10. CHAPTER

Testbed and benchmarking

The data and graphs showed in this chapter are specific to the present project, and have not been corroborated by any other testing framework or research. The explanations that go with the results are as objective as possible, but are personal interpretations nonetheless.

10.1 Testing environment

All the tests have been done in the laboratory, without exception, so a minimum level of homogeneity between tests should be guaranteed. However, the equipments of some workmates may produce interferences during the day (mostly because some of them actively use the wireless WIFI and ZigBee protocols, or even Bluetooth), so the preferred time to carry on the tests has been during night hours, starting sometime in the evening and terminating them early in the morning.

10.2 Selection of relevant tests

From the different test scenarios and plans that were identified during the analysis process, a few have been selected to be analyzed and discussed. The other cases have been either excluded due to inconsistencies in the data, or they have not been done with the final version of the testbed (i.e. they have been done during the development phase in a ”beta” fashion, as the test conditions were not as homogeneous or controlled as they should).

Below, the tests that have been done are listed and classified in different iterations. Technically, performing each of them would have required a whole day (or even more), but so as to reduce the testing process, some of them have been cut out to a few hours. Furthermore, no tests could be carried on at the same time and place due to the interferences, and the phones' batteries needed to bee fully recharged before starting a new test.

- *Ground tests*: measure the *de facto* battery drain by the phone.
  - No discovery or sensors connected (the device is in stand-by).
- *Only discovery tests*: measure the battery drain of a continuous discovery.
  - Each phone isolated.
  - One phone in continuous discovery and the others nearby (just in standby).
  - All phones nearby, and all in continuous discovery.
- *Ping tests*:
  - Initial discovery with one sensor-device. Measures the ping drain.
  - Initial discovery with seven sensor-devices. Measures the 7x ping drain.
- *Discovery & ping tests*:
  - Continuous discovery with one sensor-device. Measures the ping+discovery drain.
  - Continuous discovery with seven sensor-devices. Measures the 7x ping+discovery drain.
- *Stress tests*:
  - Initial discovery with one sensor-device in STRESS mode. Measures the ping+stress drain.
  - Continuous discovery with seven sensor-devices in STRESS mode. Measures the 7x ping+stress drain.

10.3 Test iterations

To perform the tests in an organized way, a ”testing iteration” method or procedure has been followed. Each iteration encompasses a set of tests that have the same purpose of measuring a certain behavior under some specific conditions.

Ground tests

First, a preliminary assessment has been done, which sets the ”grounding” or reference for the tests that would follow. This tests consisted in doing some isolated proves for the smartphones to measure the power drain each of them had in standby. These proves were done twice for each phone: for the first one, the testing Android application was run without further ado. Prior to the second, however, some tweaks were done to the phones, where background services and applications that had nothing to do with the testing were stopped. The second row of tests showed slightly improved battery performances, so these ”tweaks” have not been applied in subsequent tests.

After this preliminary assessment, it seemed that in standby mode the *Galaxy Nexus* was the most efficient than its peers, very closely followed by the *Nexus 5*.

Figure 10.1 shows the graphs for the battery drain over a time-span of ~19 hours, while table 10.1 shows the mean battery drain rates (assuming that the consumption stays constant over the whole battery life).

As the graphs clearly show, the Nexus S has a different battery-reading rate than the other two phones. The reason behind this is probably due to the hardware and software version differences.

Device	battery % per min.
Nexus S	0.00844
Galaxy Nexus	0.00515
Nexus 5	0.00523

Table 10.1: Battery drain rates in standby.

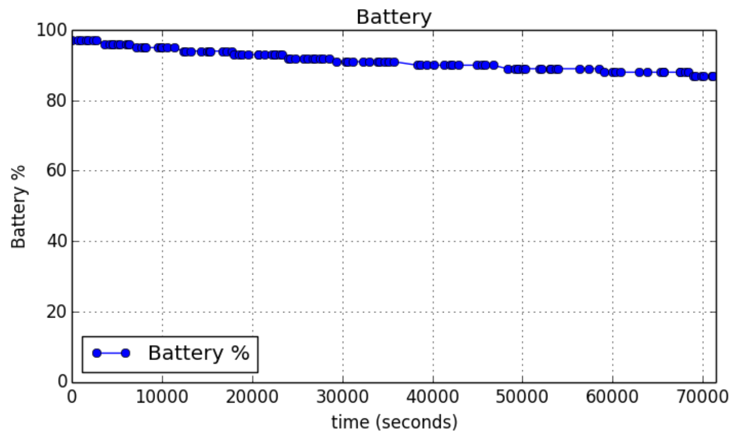
Only discovery tests

The next iteration of tests intended to measure the battery drain of the discovery process in each smartphone. For this purpose, each phone was isolated and left in a continuous discovering state. As expected, the results showed a noticeable increase in battery drain.

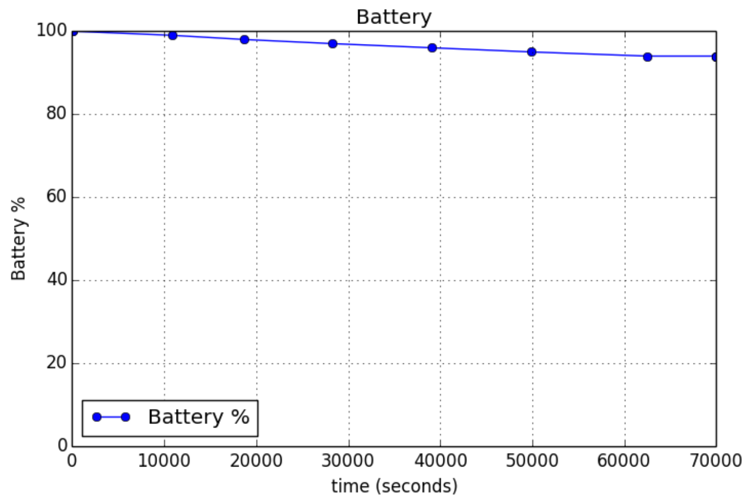
Figure 10.2 shows the graphs for the battery drain over a time-span of ~16 hours, while table 10.2 shows the mean battery drain rates for each device.

A second iteration of these same tests (discovery only) was done with a Bluetooth ”decoy” (another phone with Bluetooth activated and set in discoverable mode) was left

(a) Nexus S



(b) Galaxy Nexus



(c) Nexus 5

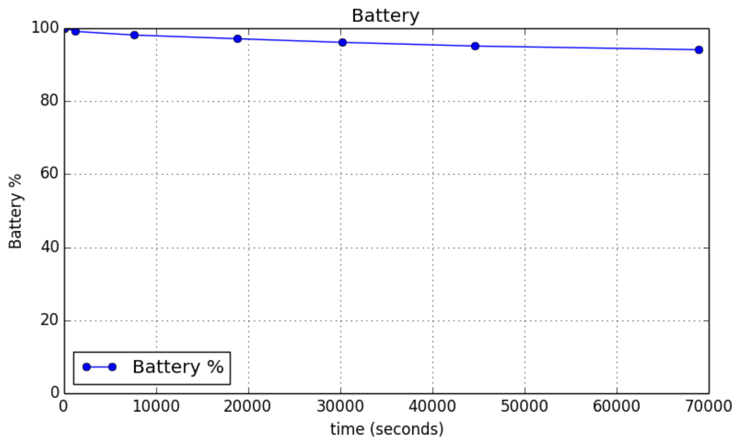
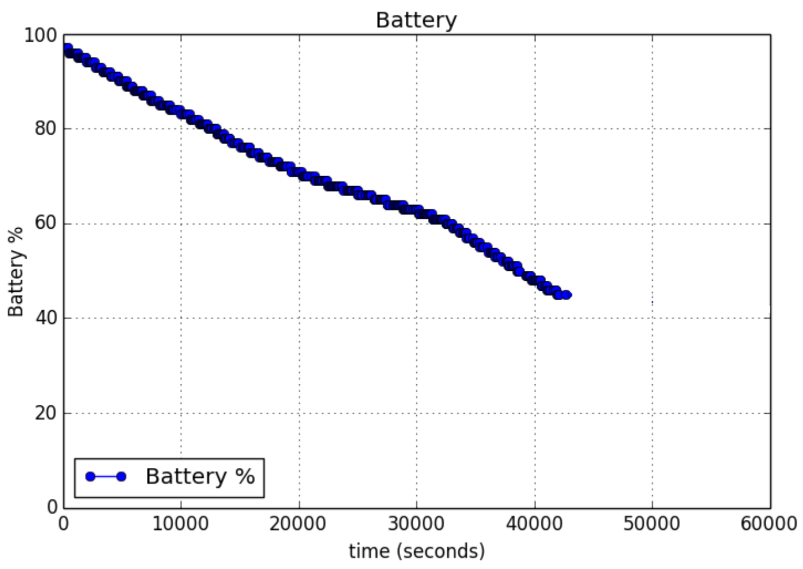
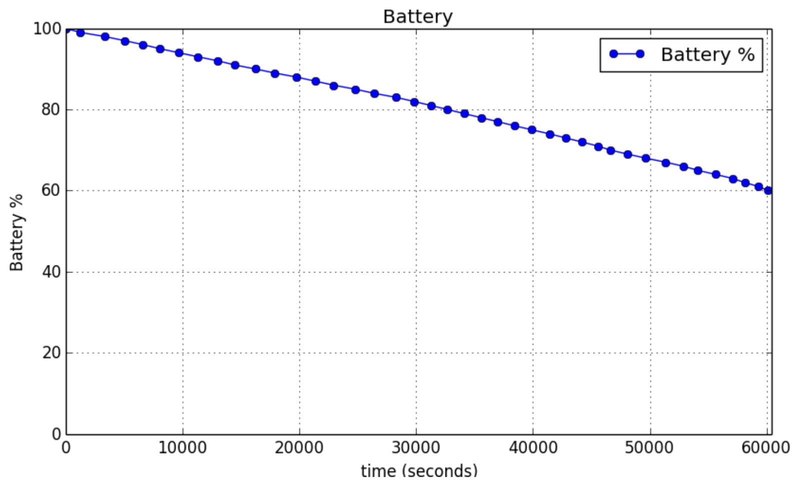


Figure 10.1: Ground tests: standby without Bluetooth (~19 hours).

(a) Nexus S



(b) Galaxy Nexus



(c) Nexus 5

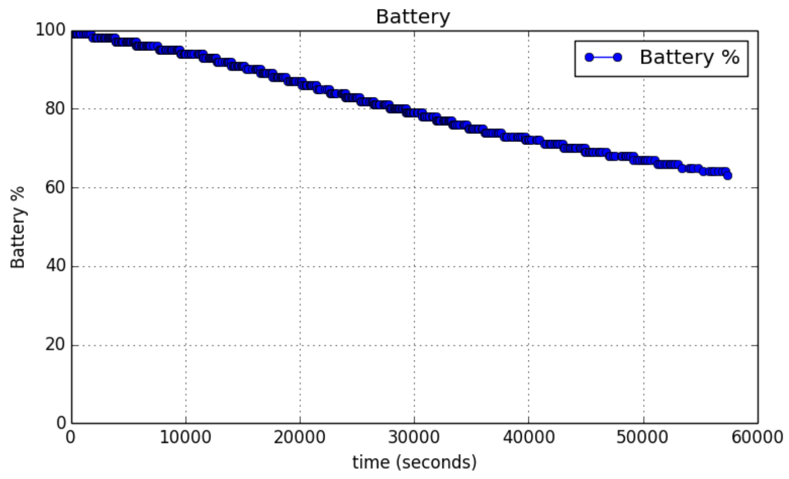


Figure 10.2: ”Only-discovery” tests (isolated).

Device	battery % per min.
Nexus S	0.0451
Galaxy Nexus	0.04105
Nexus 5	0.04454

Table 10.2: Battery drain rates in continuous discovery.

nearby the smartphone that was performing the discoveries. This sensor was not bonded to the phone, and each discovery found it over and over again. Therefore, even if no connection was being established with this 'decoy", the discoveries found it. The results showed an important increase in battery drain compared to the discoveries done in complete isolation.

Figure 10.3 shows the graphs for the battery drain over a time-span of ~16 hours, while table 10.3 shows the mean battery drain rates for each device.

Sub-figure 10.3b shows that the *Galaxy Nexus* has suffered the biggest battery drain increase when discovering with a "decoy" nearby. Note that during the first 12000 seconds (~3h 30min), the battery drain rate is similar to the other two smartphones, but it suddenly plummets faster than before. It is still not very clear why this has happened.

Ping tests

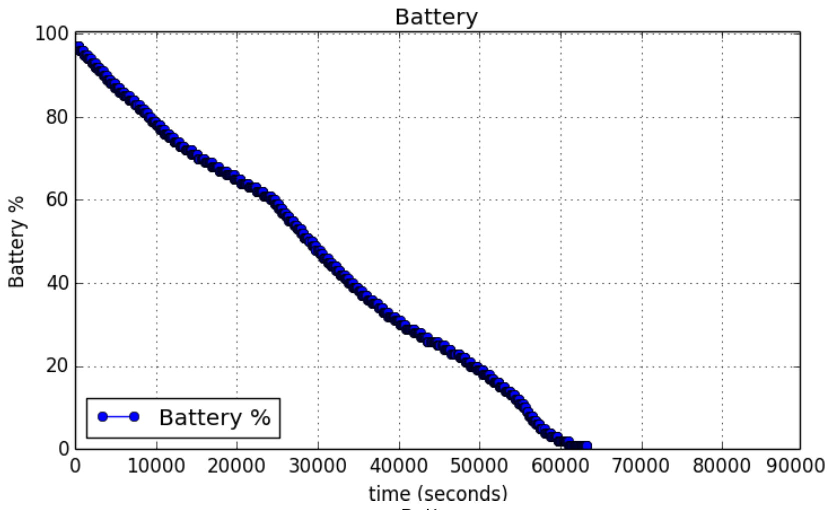
During this testing iteration, the ping or response time for the sensor devices in different conditions wanted to be measured. First, a simple scenario composed by just a single Arduino with an initial discovery plan was carried. Then, the same test was done, but with six Arduinos at the same time.

The results show a slight increase in the ping times (from a mean time of 50 ms to ~75), while the throughput increases quite more (from a mean of 125 B/s to 700 B/s).

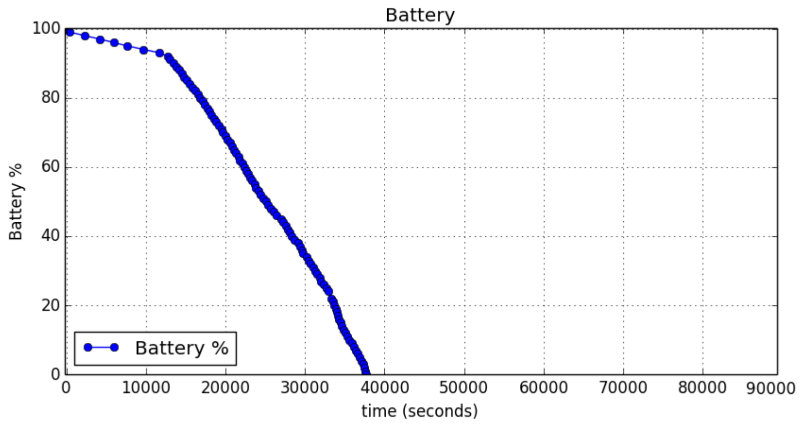
Figure 10.4 shows the graphs for the ping and throughput values with one sensor only (sub-figure 10.4a) and then with six simultaneously (sub-figure 10.4b). As it can be seen, the mean throughput increases six times, while the mean ping time remains almost unchanged.

Later, a similar test was done first with one and later with six sensors, while the phone performed periodic discoveries every 30 seconds. The results clearly showed an increase in the ping values when a discovery was in process. Figure 10.5 shows a graph where it can be seen the impact of a discovery in the ping and throughput for communications with

(a) Nexus S



(b) Galaxy Nexus



(c) Nexus 5

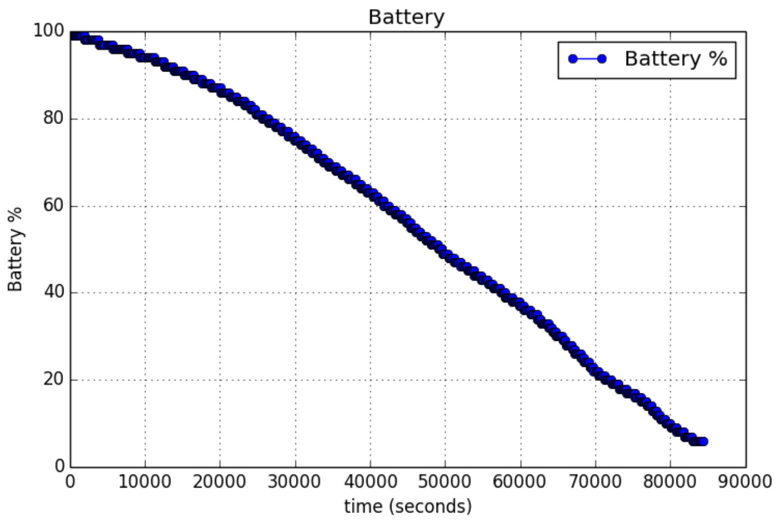
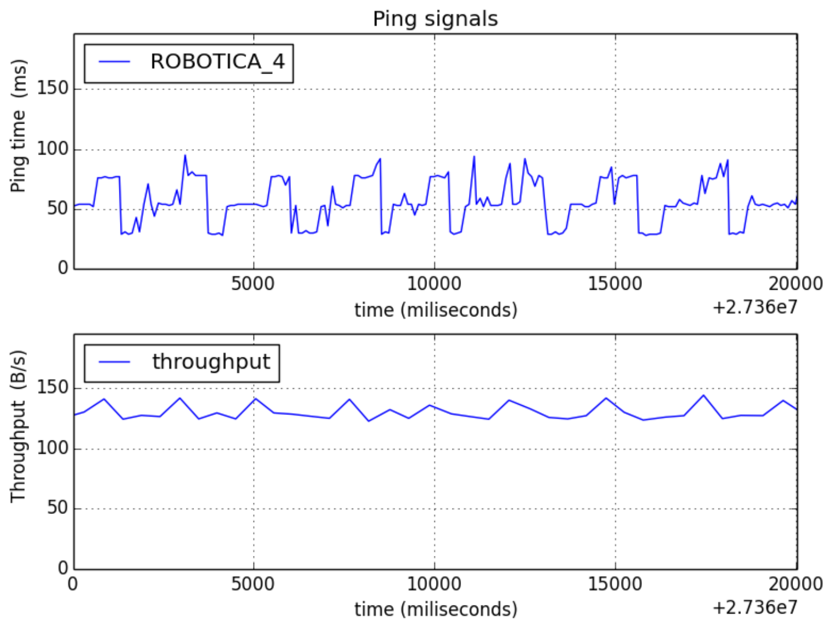


Figure 10.3: ”Only-discovery” tests (not isolated).

(a) One sensor



(b) Six sensors

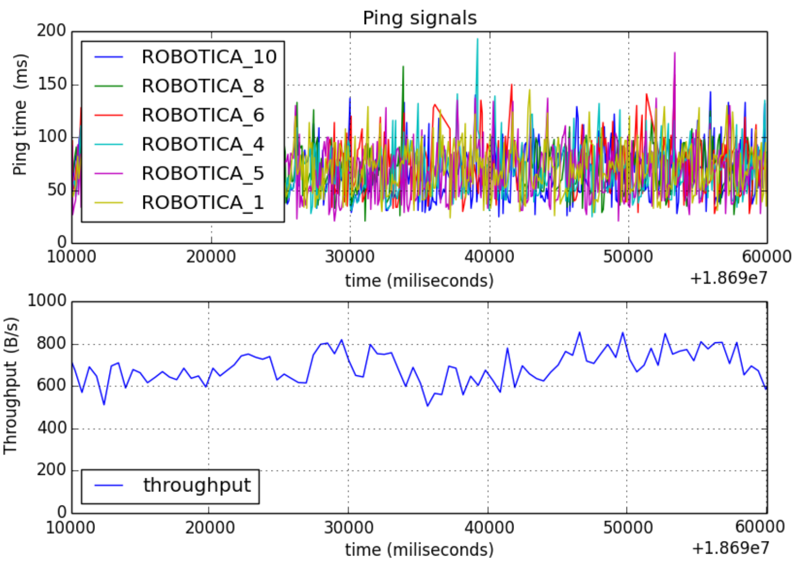


Figure 10.4: Ping tests (with one and six sensors).

Device	battery % per min.
Nexus S	0.0920
Galaxy Nexus	0.16
Nexus 5	0.06684

Table 10.3: Battery drain rates in continuous discovery.

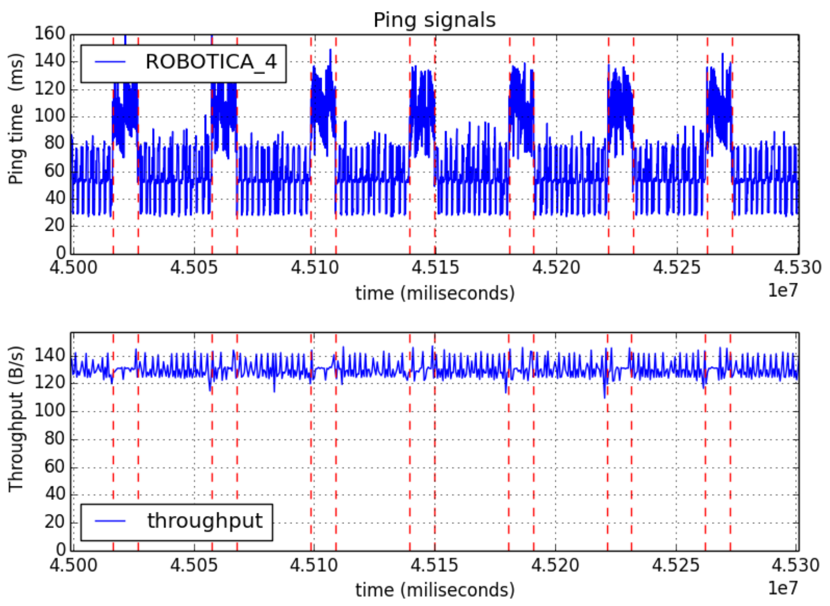
a single sensor (sub-figure 10.5a) and with six (10.5b). Looking closer to figure 10.5b, it can be seen that not all ping times are altered equally. Indeed, three of the signals suffer an increased delay of up to 200-800 milliseconds, while the other four barely get to the 200 ms during each discovery. It is no coincidence that those four signals correspond to the Arduino UNO boards. My conclusion for this is that the simulated serial connection of the Arduino UNOs with the DF-Bluetooth modules is faster than their MEGA peers.

Stress tests

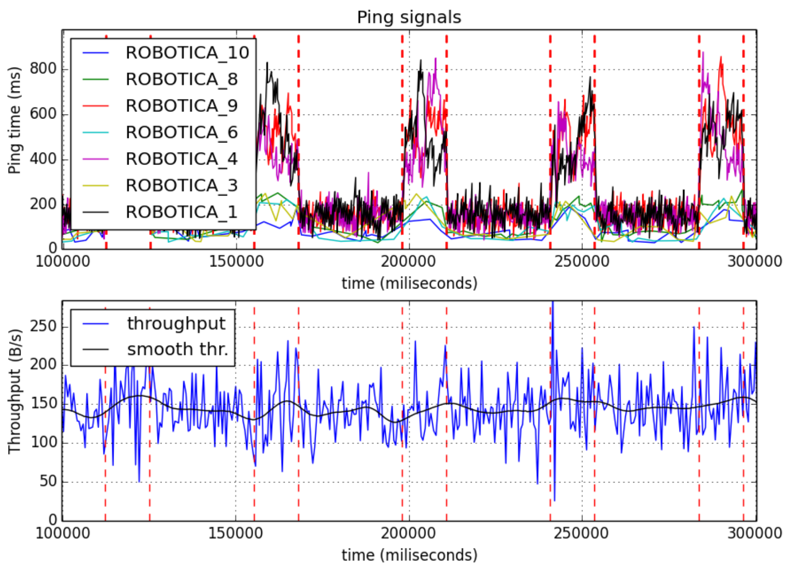
During this phase, a set of ”stress tests” were done with seven Bluetooth sensors transmitting loads of data to a single smartphone (the *Nexus 5*). The phone was subjected to different stress levels in each test-iteration (i.e. in each iteration the sensors sent more data than in the previous). The results were really interesting.

Figure 10.6 shows the results for one ”stress test” in which seven Arduino sensors sent data increasingly over time. Starting at 10 Bytes per second, every 10 minutes the message size for all of them grows in 100 Bytes. In the second sub-graph, the throughput is calculated by the actual ”useful” messages received, so all the messages that contain errors do not count for the ”valid” throughput. The third sub-graph shows the size of the incoming messages. It seems that at a transmission rate of 1.5 kB/s, the communications reach a limit, and from then on the ”stress messages” begin to have errors, until all of them seem to be corrupt. This is logical, since the larger a message is, the more possibilities of having a corrupt byte it has. Therefore, it looks as if the throughput falls, and the size of the ”stress messages” is null, when in reality, all incoming messages are discarded due to errors in them. That’s why the time for the ping messages keeps growing as well.

It should be noted that much like it happens with the increased ping times when a discovery takes place (see section 10.3), it seems that some of the Bluetooth sensors do not suffer from increased ping times during the ”stress tests”. Again, it is no coincidence that those very sensors correspond to the Arduino UNO boards.



(a) One sensor



(b) Seven sensors

Figure 10.5: Ping tests while performing periodic discoveries (with one and seven sensors).

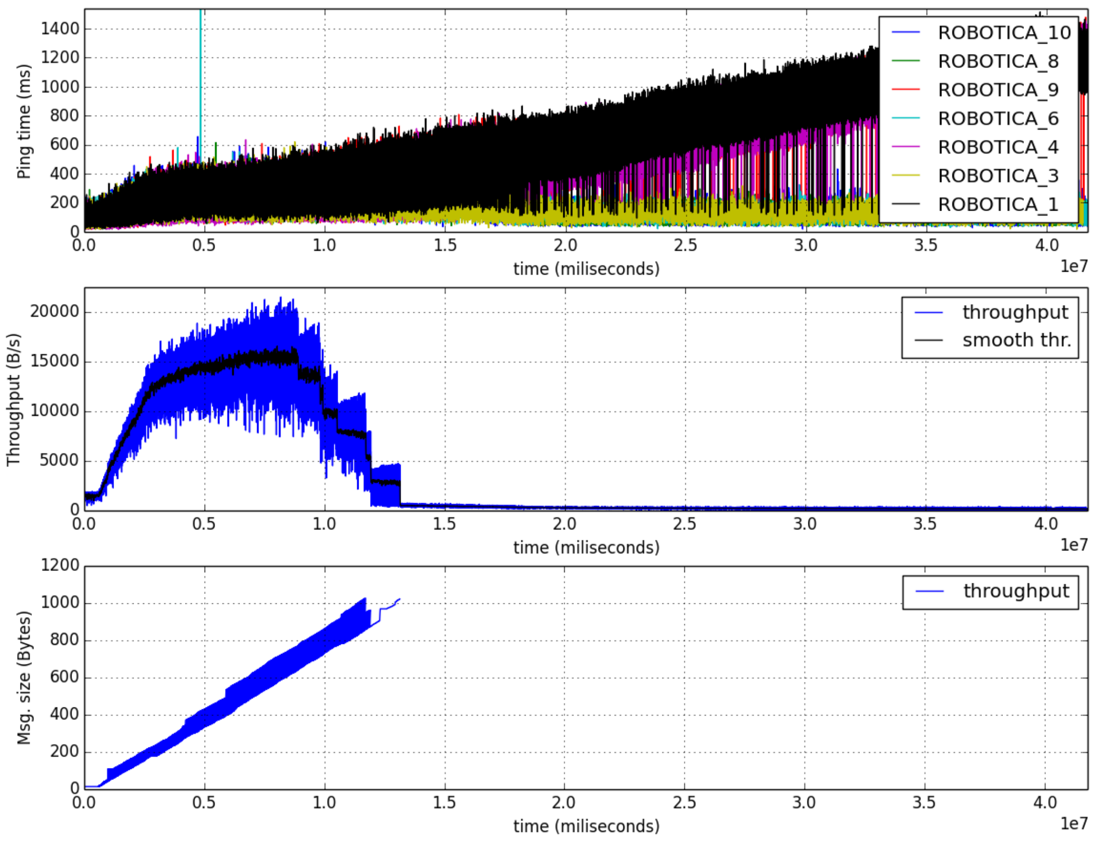


Figure 10.6: Stress test.

10.4 Additional tests

During the testing process, some tests did not have the outcome that was expected, yet a few of them happened to be very insightful. Following, some of the most relevant are explained.

Maximum number of paired devices in one discovery

During the performance tests, I noticed about a peculiar behavior of the discovery and pairing process. While trying to perform a test on a static scenario with 8 Arduino devices, it seemed that an initial discovery did not always find the same number of devices (and therefore the number of paired devices for the test changed as well). However, after a subsequent execution of the application (in a matter of seconds or minutes), more devices than the previous time were found (but still not all).

After some trial and errors, I concluded that with just one discovery, only three or four devices (from the 8 that were around) were being found. After terminating the application (and the connection with the current devices), a new discovery was able to find five to seven devices. By the third subsequent discovery all nearby sensors seemed to be found (and 7 of them connected). All three smartphones showed the same behavior. Following discoveries kept finding all the sensors nearby, unless they were rebooted.

This finding shows that a new plan in which a few discoveries (three or four at max) are done at the beginning would guarantee that all nearby devices are found, since doing a single one has proved to be insufficient.

Sudden CPU usage increase

At an early time of the performance testing, I stumbled upon a previously unseen behavior in the CPU's usage while looking at a test's results. The graph of the logged data clearly showed a point in which in less than two hours the battery level plummeted when it should not have. After analyzing the data, it was evident that an increase of the CPU usage had caused the battery to exhaust in less than two hours.

This test consisted of an initial discovery plan and a static scenario with 7 Arduinos nearby (all connected with the Android application). The only communications that happened during the test were the PING messages that went from the phone to the Bluetooth modules and back (for each module).

During the first six hours of the test, the smartphone showed a mean battery drain of 0.06% per minute, and a mean CPU usage of 66%. Then, all of a sudden, the battery drain rate increased to a 0.61% per minute, and the mean CPU usage to 98%. Figure 10.7 shows the results of the test. After analyzing the log files I discovered that for some reason, four

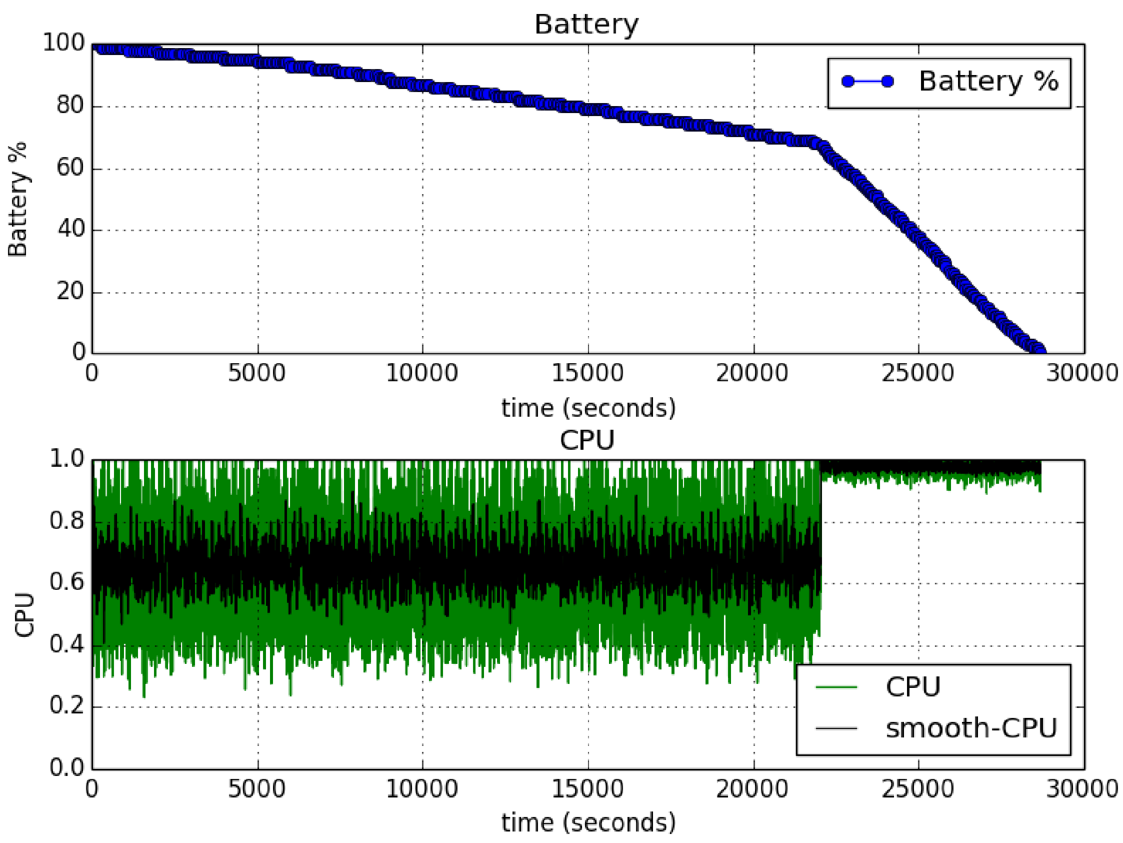


Figure 10.7: Sudden CPU usage and battery drain increase.

Arduino boards were suddenly disconnected from the Android application. The thing is that when this disconnection happened, the Android application entered in some sort of looping state that choked the phone's resources.

The story behind this particular test is meaningless by itself, but it serves as a perfect example for the case in point that a bad application design gets the battery killed faster than performing simultaneous Bluetooth communications with several devices.

CPU usage and battery drain anomaly

A very odd result was obtained during one of the discovery tests—with the Galaxy Nexus, specifically—. While performing a continuous discovery, the battery drain is normal until a point when the CPU usage decreases and the battery starts to drop faster than before (figure 10.8 shows the test's result in question). If anything, when the CPU drops the battery drain should slow down, or at least stay still, but not plummet! If zoomed in the moment where the CPU usage decreases (see figure 10.9), it can clearly be seen that in the second part the CPU usage changes with each new discovery.

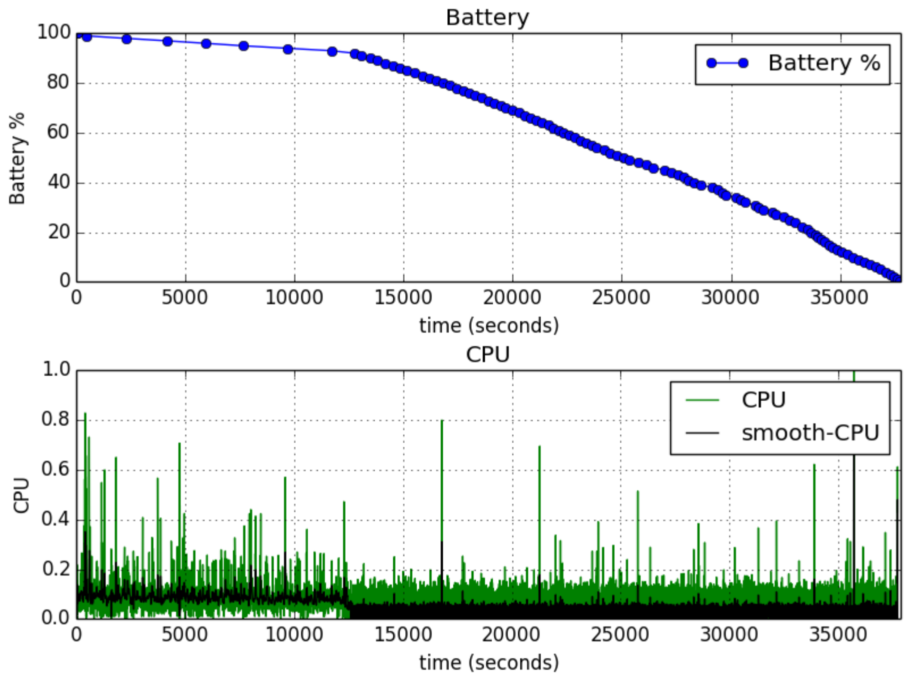


Figure 10.8: CPU usage and battery drain anomaly.

In figure 10.8 it is clear that the mean CPU usage decreases, but it seems as if the readings are more "dense" (as if there were a higher number of CPU usages, but of less weight). A little more insight is given by figure 10.9, where the exact point in which the CPU usage changes is shown. The vertical lines denote the end and beginning of a new discovery. Note how the CPU usage changes on a par with the discoveries during the last part of the graph. From this picture, it seems as if during the first part the discovery did not alter the CPU usage at all, while at some point in time, it does. Still, no reasonable explanation has been found for this anomaly.

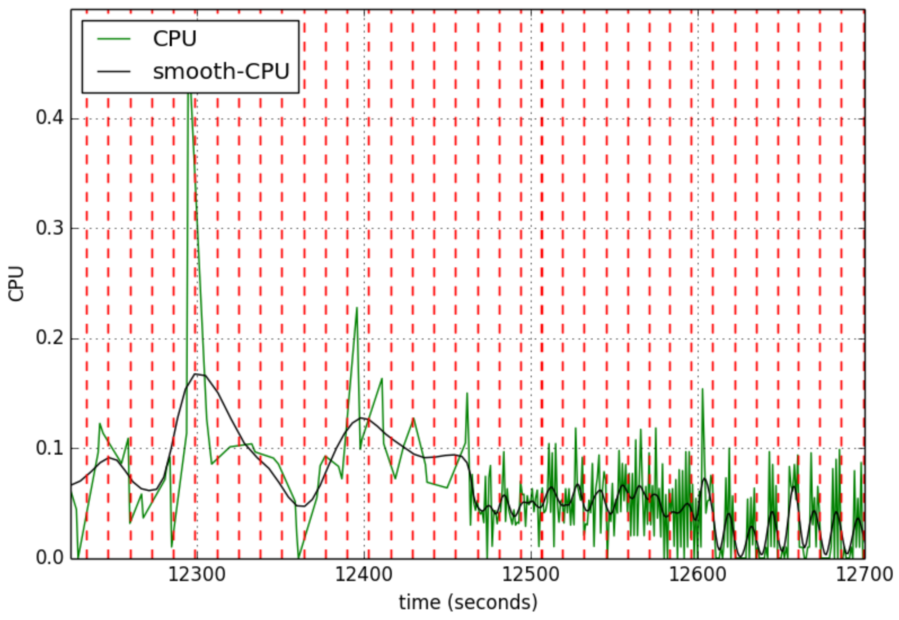


Figure 10.9: CPU usage and battery drain anomaly zoomed in. The red lines point out the beginning and end of the discoveries.

10.5 Discussion

Impact of Bluetooth discovery on the CPU

Figure 10.10 clearly shows that the CPU usage varies according to the discovery of Bluetooth. This capture is from a test done with the Galaxy Nexus, and it seems as if the CPU soars near the end of each discovery.

The time span of the discoveries in this particular capture is around 11 seconds. However, as observed in other tests, the discovery process can take between 10 and 15 seconds. Table 10.4 shows the mean time span of the discoveries for the different smartphones used in the tests that involved continuous discoveries.

Impact of discovery on Bluetooth communications

Figure 10.11 shows the normal ping and throughput values for a single device. During this capture, only the communications to calculate the ping times were being done, so these are the ”purest” readings that can be done to see the normal behavior of Bluetooth in Android.

However, this correlation changes noticeably when the smartphone is performing a

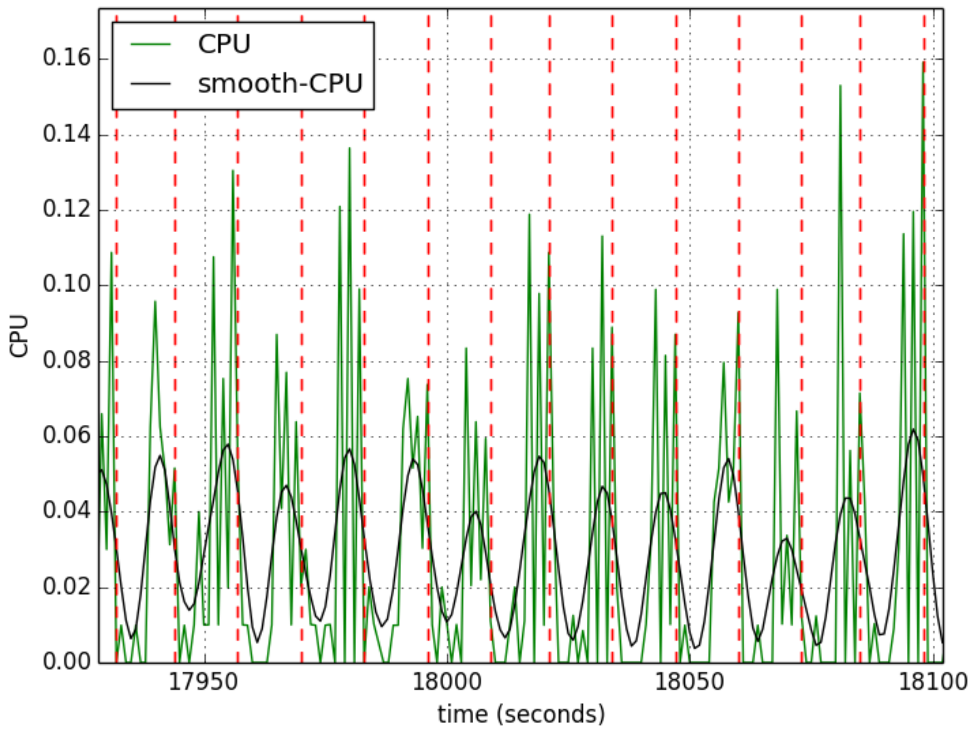


Figure 10.10: CPU usage during continuous discovery (the vertical red lines point the times when a discovery has finished and the next has started).

Bluetooth discovery. As figure 10.12 shows, the impact of a single discovery notably alters the performance of the ongoing communications. The vertical red lines enclose the span when a Bluetooth discovery is being done. It can be seen that the ping is affected, as the mean time is almost duplicated during the discovery.

Performance, nº of devices and size of the messages

According to the results of the "ping tests" shown in figure 10.4, the response time for a message sent from the smartphone to a sensor and back (a ping) fairly increases with more devices (about ~15 ms for each new device). When a discovery takes place, the ping times increase differently according to the Arduino board, as shown in figure 10.5b.

When the size of the messages gets bigger, however, the ping time grows considerably (as shown in figure 10.6).

Device	Discovery time (milliseconds)
Nexus S	10,280
Galaxy Nexus	12,860
Nexus 5	13,750

Table 10.4: Mean discovery span times.

Errors in the communications

During the performance testing process, a bizarre behavior was identified around the messages' error rate. Indeed, under ”normal” circumstances (no discoveries and just ping messages) the errors seem to happen only with the Arduino UNO boards, while the MEGAs yield none. Nevertheless, in an scenario where ”stressing” messages are incoming, both the Arduino UNO and MEGA boards yield errors.

It has been assumed that this is due to a hardware issue regarding the incompatibility between the boards and the DF-Bluetooth modules. However, the origin of this issue could be the *SoftwareSerial* library that simulates the serial communications.

Table 10.5 shows the results obtained in a single test with an initial discovery and a static scenario with six Bluetooth sensors nearby (two of them being Arduino UNOs, and MEGAs the rest). It is clearly shown that the MEGA boards—strange though they may seem—have not yielded a single corrupted message, while almost a 4% of the messages sent by the Arduino UNOs appear to be corrupted.

Table 10.6 shows a similar content, with the difference that this time the Bluetooth sensors sent ”stress” data at ~100 bytes/s each. Although in this scenario all boards have produced corrupted messages, the UNOs yield ten times more errors than the MEGAs.

The reason for this may be related to the aforementioned case in which a discovery seemed to disrupt the MEGA boards' ping much more than the UNO's (see section 10.3). The reason for that is thought to be the faster speed of the virtual or simulated serial connection of the Arduino UNO boards. If indeed the virtual connection is faster than the non-simulated one, it could be natural for it to yield more errors.

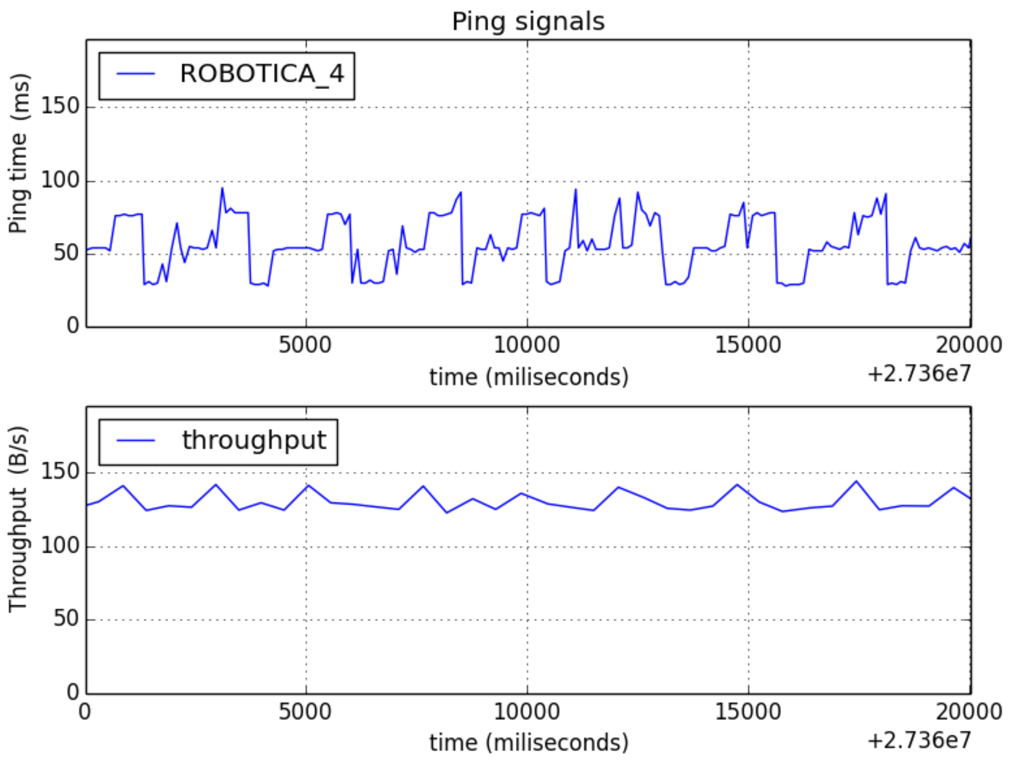


Figure 10.11: Relation between ping and throughput under normal conditions (without discovery nor incoming ”stress” messages).

$Board_{(module)}$	**Clean pings**	**Corrupted messages**	**Error rate**
$MEGA_{r1}$	562,900	0	0%
$MEGA_{r4}$	562,900	0	0%
$MEGA_{r5}$	562,900	0	0%
UNO_{r6}	270,700	10,842	3.85%
UNO_{r8}	400,200	15,588	3.75%
$MEGA_{r10}$	562,900	0	0.0%
Total	2,922,500	26,430	0.90%

Table 10.5: Corrupted ”ping” messages rate.

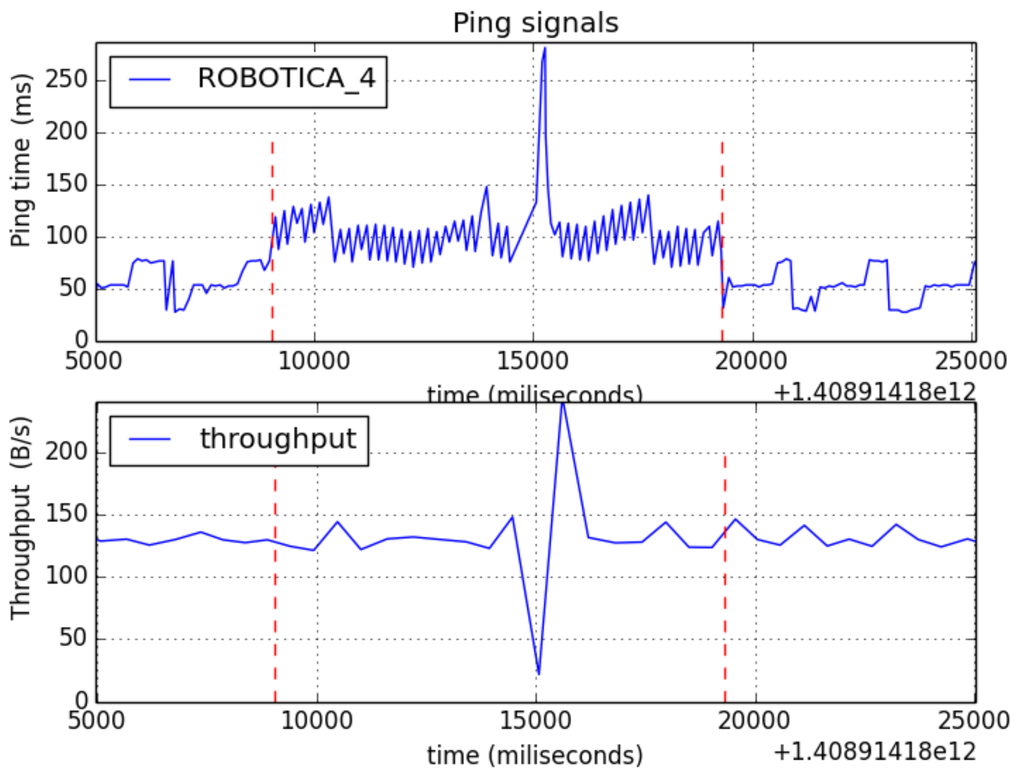


Figure 10.12: Relation between ping and throughput while discovering.

Board$_{(module)}$	**Clean pings**	**Clean data**	**Corrupted messages**	**Error rate**
$MEGA_{r1}$	343,900	343,200	2,120	0.31%
$MEGA_{r4}$	343,600	342,900	1,050	0.15%
$MEGA_{r5}$	344,400	343,900	984	0.14%
UNO_{r6}	255,200	291,500	13,081	2.39%
UNO_{r8}	300,400	340,800	14,545	2.27%
$MEGA_{r10}$	397,000	395,700	1,114	0.14%
Total	1,984,500	2,058,000	32,894	0.81%

Table 10.6: Corrupted messages rate.

11. CHAPTER

Control and monitoring

11.1 Evolution of the project

One of the most powerful features of GitHub is the statistics reports that are automatically generated. Although this information is publicly available[1], it is worth mentioning here as well. Note that these graphs only correspond to the on-line repository, and that more work may have been done in a local machine.

Weekly contribution report

Figure 11.1 shows the weekly committed contributions for the months of July and August. There are two major periods in which more contributions have been made. The low number of contributions in the time between is a reflection that no commits have been uploaded, probably because of the lack of a ”publishable” code.

”Code frequency”

The following figure 11.2 shows the weekly additions and deletions to the repository during each week. The large deletion in the first half of August is originated because of the deletion of some log-files that should not have been committed in the first place.

[1]The statistics can be found here.

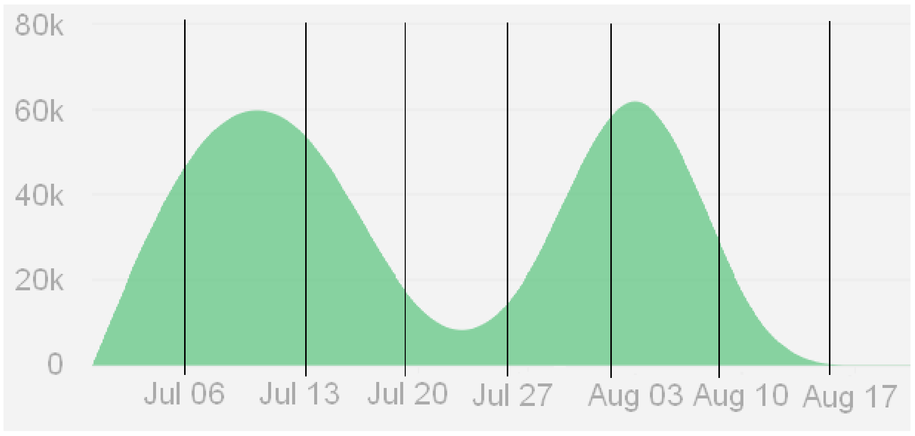


Figure 11.1: Weekly contributions.

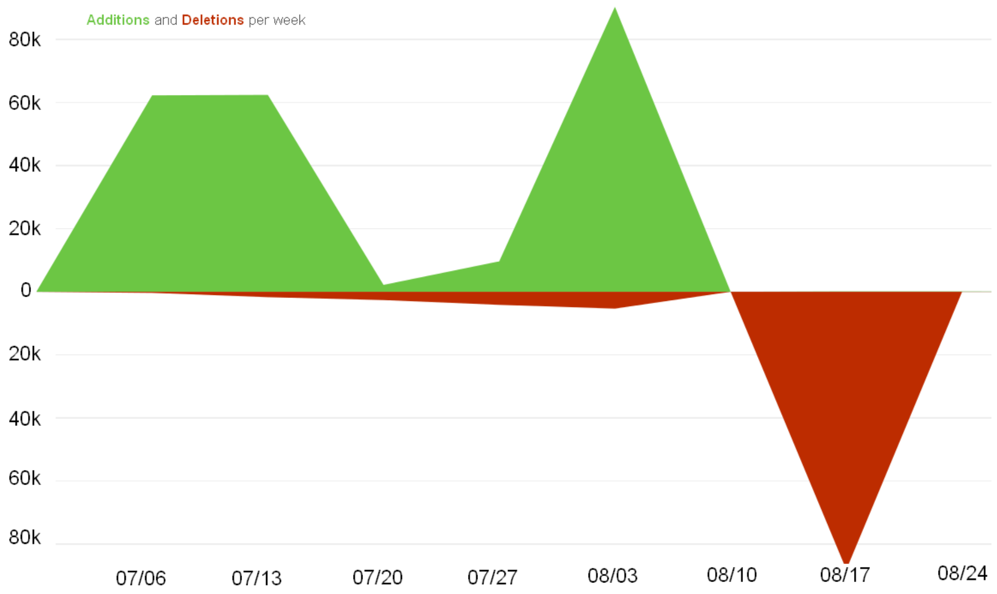


Figure 11.2: Additions and deletions to the online repository per week.

”Punch Card”

Figure 11.3 shows the frequency of updates to the repository based on the day of week and time of the day. The size of the black circle indicates commit frequency. A total of 30 commits or updates have been done since the start of the project. As it can be seen, most of them have been uploaded at afternoon hours, with no particular preference as of the day of the week.

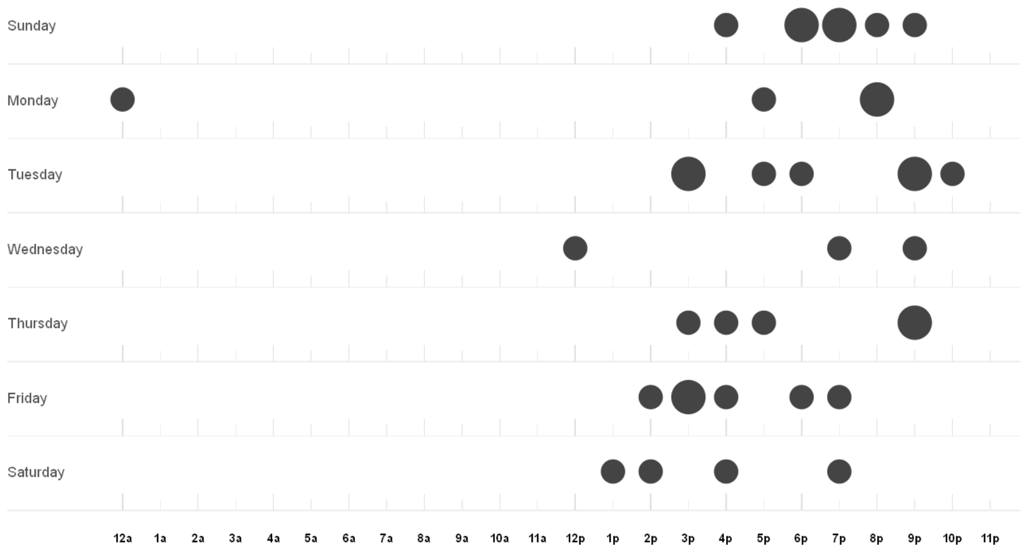


Figure 11.3: ”Punch card” (frequency of updates to the repository based on the day of week and time of the day).

11.2 Deviations from initial plan

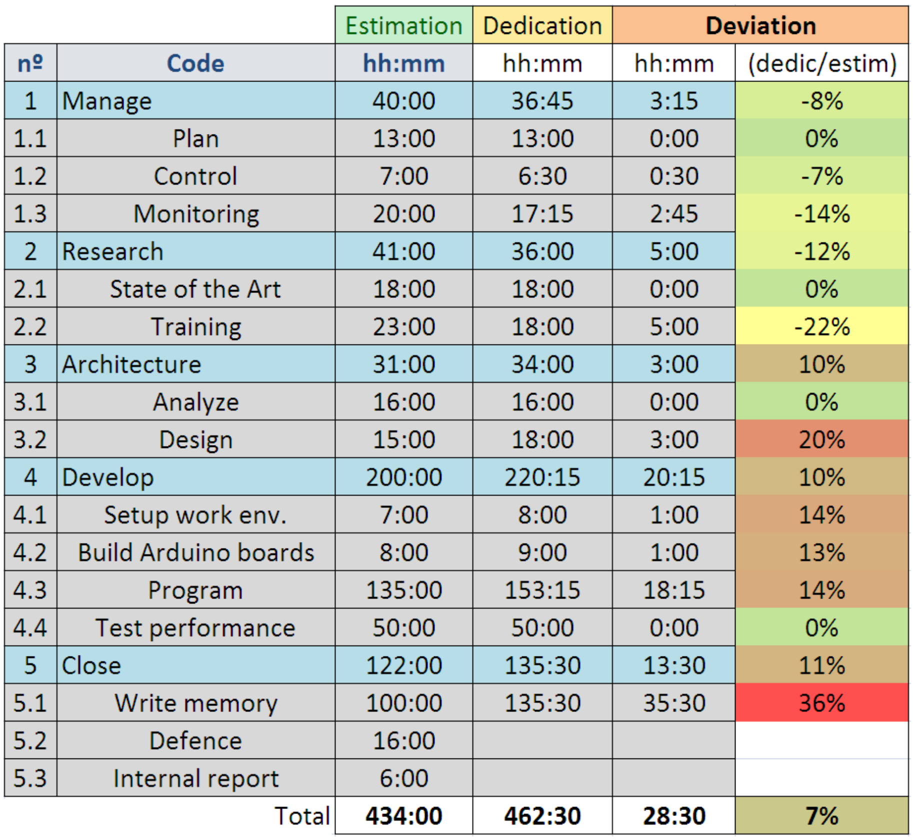
In this section a review over the hours worked on each task compared to the initial estimations is done.

On the spot, there has not been a fatal delay nor risk that threatened the fulfillment level of the project. However, there have been deviations from the estimated number of hours and the final real dedication. Although they have not been severe, it is worth noting where and why these deviations have happened.

The initially expected deadline for the defense has been changed to a an earlier time. However, there’s enough time for preparing it, so no further readjustments need to be done.

Initially, a daily iteration method was devised (see section 2.5 and figure 2.2). At the beginning of the project I followed this method to avoid getting bored with just performing the same task daily, but then I decided to give it up and concentrate fully on the task that required more urgency.

Figure 11.4 shows the planned and dedicated hours per task, with the calculated deviations for each. Note that the last column is the deviation percentage for each task individually, so the largest percentage needs not have more hours. Additionally, note that the last two tasks have not dedicated hours, as at the time of this writing they have not been started.



		Estimation	Dedication	Deviation	
nº	Code	hh:mm	hh:mm	hh:mm	(dedic/estim)
1	Manage	40:00	36:45	3:15	-8%
1.1	Plan	13:00	13:00	0:00	0%
1.2	Control	7:00	6:30	0:30	-7%
1.3	Monitoring	20:00	17:15	2:45	-14%
2	Research	41:00	36:00	5:00	-12%
2.1	State of the Art	18:00	18:00	0:00	0%
2.2	Training	23:00	18:00	5:00	-22%
3	Architecture	31:00	34:00	3:00	10%
3.1	Analyze	16:00	16:00	0:00	0%
3.2	Design	15:00	18:00	3:00	20%
4	Develop	200:00	220:15	20:15	10%
4.1	Setup work env.	7:00	8:00	1:00	14%
4.2	Build Arduino boards	8:00	9:00	1:00	13%
4.3	Program	135:00	153:15	18:15	14%
4.4	Test performance	50:00	50:00	0:00	0%
5	Close	122:00	135:30	13:30	11%
5.1	Write memory	100:00	135:30	35:30	36%
5.2	Defence	16:00			
5.3	Internal report	6:00			
	Total	**434:00**	**462:30**	**28:30**	**7%**

Figure 11.4: Planned and dedicated hours with deviations. The deviation is calculated row-wise, not over the total.

It is clear that the main deviation has happened in the writing process of this memory itself, with a surplus of 36 hours. The reason for this is nothing else than a lack of experience with writings of such caliber, which made my estimations less accurate. The writing process itself has not been the most difficult part, but rather the reviewing and final polishing. The fact that it has all been done with LaTeXhas added an additional struggle as well. Indeed, even if the learning curve for LaTeXhas not been as slow as I expected, finding out how to correct some final details has taken a lot of effort. This is provably what has made me spend those unexpected ”extra” hours.

The programming task has taken more than the expected time as well. Firstly because the Android application took more than the expected effort to finish, and secondly because the process of creating the graphs with the Python plotting scripts has been rather slow.

The next largest deviation has happened in the training process, this time with a shortage of five hours. It is not a big deal of time, and the reason for this shortage is justified by the fact that most of the technologies used had already been used, so less hours than expected have been required.

Figure 11.5 portrays the difference between the estimation and final dedication for each high-level tasks.

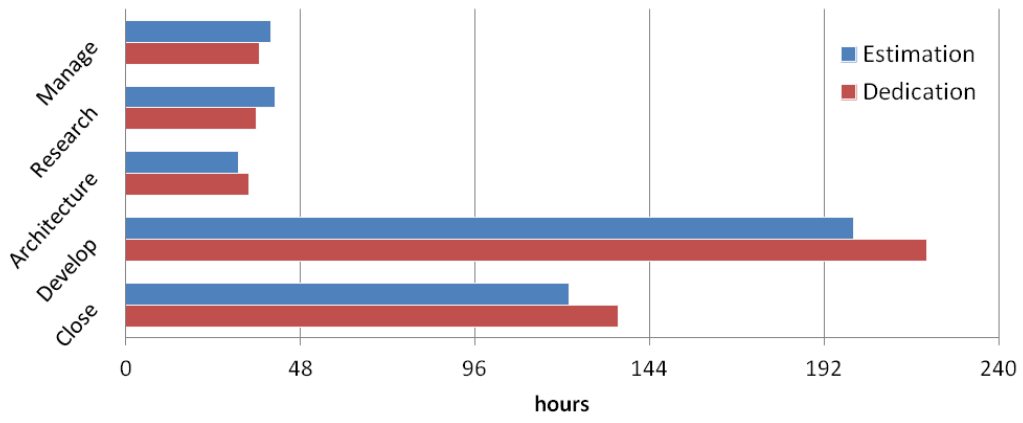


Figure 11.5: Estimated and deviated hours for high-level tasks.

Figure 11.6 shows a comparative between the high-level tasks according to the hours dedicated to each every month. During the first weeks of the project the management tasks have taken most of the dedication.

The next couple of months (April and May) have almost entirely been dedicated to researching. Most of the analysis and design of the system has been done in June, while most of the development in July and early August. Finally, the writing of this memory has been done in August. The dedication of hours has not been constant over time, as it can be seen in figure 11.7, where the accumulated hours during the months that this project has spanned is shown. During the first three months not much was done, as I had not the time for dedicating to the project, so most of the work has been done from late June until early September.

11.3 Assessment of the project

Following, an objective assessment of the project is done. Then, a personal evaluation of the performance of the project and the work done is made.

11.3.1 Key Performance Indicators

With the Key Performance Indicators defined in the first chapter (section 2.7.1) at hand, it's time to asses the level of fulfillment of the goals that were proposed at the beginning.

Quantifiable indicators:

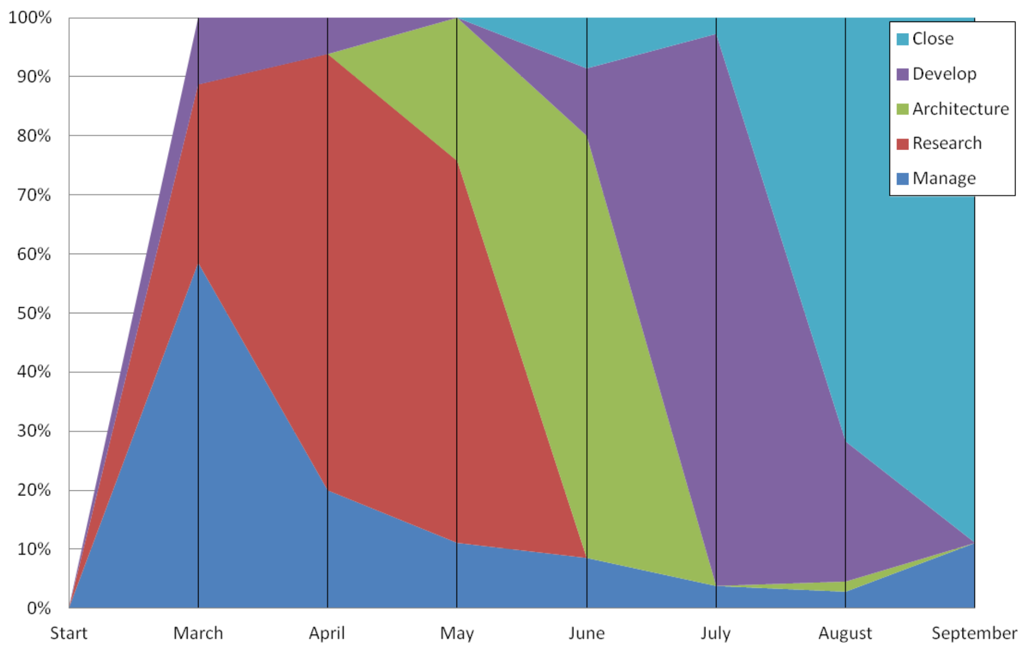


Figure 11.6: Dedication percentage to each high-level task during the project's life.

- Schedule keep-up: the schedule has been respected without any changes.
- Hours worked: although a few more hours than expected have been necessary, the total deviation has been of almost a 6%, so it can be considered as successful.
- Lines of code: although no formal calculation for the number of code-lines has been done, the *GitHub* repository's statistics show that the contributions to the project have been concentrated in July.
- Efficiency of the framework: the developed Android application does not use more resources than needed.

Qualitative indicators:

- Clearness of the code: the Android application is fully documented, but the Arduino and Python logs are not.
- Satisfaction with the results obtained: the results are clear, and valuable conclusions have been deduced.
- Usefulness of the research: the project has made some valuable findings, although more could be made if more tests were done.
- Experience gained: I have gained a lot of experience on Bluetooth, Android, Arduino and Python. I have learned how to manage a big project on my own as well.

Leading indicators:

- Director's approval: the director has given his approval for the presentation and

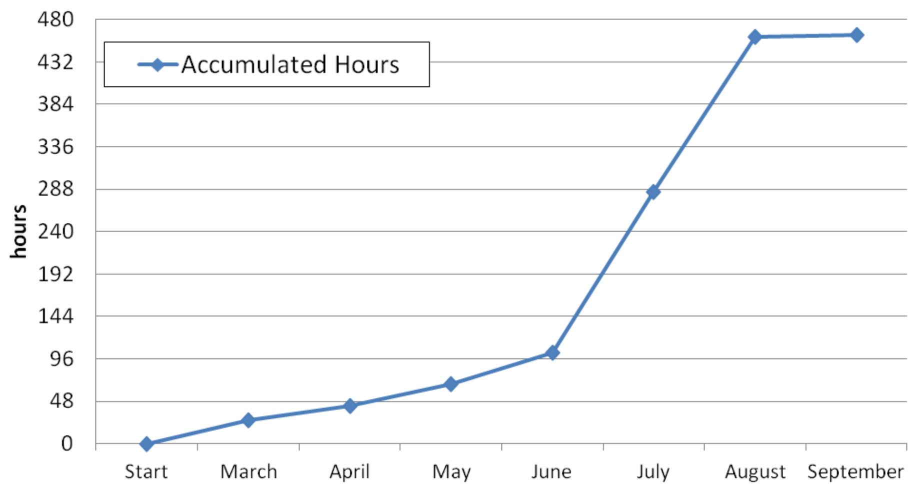


Figure 11.7: Accumulation of hours for each high-level task.

defense of this project.

- Peer feedback: from the outset, it seems that the project seems interesting to the few people that have been around.

In general, the project seems to be successful, and mainly the shortage of the different tests done could be improved.

11.3.2 Personal evaluation

- The overall feeling with that the work done is very satisfactory. I have been able to carry a long project without too many problems, and the outcome could be very beneficial for the research field around wireless mobile communications. There's plenty of room for improvements and extensions, but the initial conclusions obtained seem promising.
- The state of the art of the project gives a good picture of the adjoining technologies and researches to this one. However, many references are quite outdated. It is not easy to find recent researches around Bluetooth Classic, mostly because of the "boom" that Bluetooth *Smart* has had lately. Still, it would have been great to find peers that work or research on more similar things to this project.
- The graphs showed in the benchmarking process are not as clear as desired, as the time scales vary from one to another.
- It has been a pity that not all the tests that could have been made have been done, since there surely is potential for more findings. For example, the progressive scenarios in which the devices appear and disappear throughout a test have not been

carried out.

- There have not been major or unexpected problems, and the contingency plan has not been needed.

12. CHAPTER

Conclusions

In this chapter a review on the results obtained by the project is done, mostly focusing on the testbed testing and benchmarking. In addition to this, the future work that could be done around this project and the main learned lessons are also gathered. Finally, a personal valuation of the project as a whole and the developed work is done.

12.1 Obtained results

Impact of battery level on communications

The tests done show no correlation between the performance of the communications (or as [Edith and Gunningberg, 2013] puts it, *Quality of Information*) and the current battery level, since the PING and throughput levels seem to be constant throughout tests with static scenarios. The reason behind this is that the batteries of the Android phones output a constant power or current regardless of the battery level.

Impact of the discovery on communications

As expected, the results confirm that the discovery process is a heavyweight procedure (as previously mentioned in section 4.2.1). While a discovery is in process, the ping times for the sensors increase noticeably, while the mean throughput decreases a bit as well.

It also seems that a discovery consumes much more resources when there's at least one discoverable sensor nearby, as the comparison between figures 10.2 and 10.3 clearly shows.

Discovery length

As previously mentioned in section 4.2.1, in [Perrucci et al., 2011] it is stated that the length of a discovery increases when there's more Bluetooth devices nearby. The tests done show that this is not true for at least the three Android phones used in this project, since the discovery process seems to always take between 10 and 15 seconds, regardless of the number of nearby devices.

Minimum initial discoveries

According to what has been said in section 10.4, a single initial discovery may not be enough to find all Bluetooth devices in the area. Programs willing to connect with as much devices as possible should perform three or four subsequent discoveries so as to ensure that all in the nearby area have been found.

Maximum number of connected devices

Not to the surprise of anybody, the Android application has not been able to connect more than seven devices at the same time. This is due to Bluetooth's own limits when dividing the bandwidth, and was a known issue before starting to develop the project.

Still, it has been quite a surprise that Android's inherent Bluetooth manager has not called any exception when trying to connect an eighth device. Even more worrying is the fact that it keeps trying to open a connection, and after a while, a not very meaningful error message is raised.

This implies that any application willing to connect with as much Bluetooth devices as possible should keep track of the number of connections so that no needless connection-requests are made.

Optimal message size

According to the ”stress tests” (see section 10.3), there seems to be a point in which the bigger the message size, the greater the ping and error rate becomes, up to a point in which the totality of messages is lost or corrupt.

Seeing the results, it could be concluded that the optimal message size is near 300 bytes. This is where the ping times stay below 400 ms and the throughput is near its maximum, while the error rate seems to be low. However, in order to reach the maximum throughput, this optimal message size could be stretched up to 700 bytes, as the ping stays below 500 ms, although the error rate starts to increase considerably.

Arduino UNO vs MEGA

Although it was not the purpose of the present project to evaluate the performance of the Arduino UNO and Arduino MEGA boards, it has been unavoidable to find some differences between them. Although the UNOs yield much more errors than the MEGAs with messages of up to ~300 bytes, they seem to be faster as well (or at least their response time or ping remains almost unchanged under any circumstance) .

12.2 Future work

Scientific contribution

The first and most interesting thing that could be done while carrying on with this project is to write a formal research paper and get to publish it in a journal or conference. This way, researchers and developers all around could benefit from the discoveries made by this project. However, the benchmarks set in this project are quite technology-dependent, so some further work and a wider variety of tests is necessary.

Support for multiple protocols

As previously mentioned in other chapters, this project has not considered the Bluetooth *Smart* protocol (version 4.0 of the Bluetooth stack). Enhancing this project's framework to support this protocol seems not very complicated, although newer Bluetooth modules that support the new specifications would be needed. Doing so could bring some light to compare the ”old” and newer versions of Bluetooth, and could help to decide which of them would be most suitable for different purposes, focusing on the field of Personal Area Networks.

In addition to Bluetooth *Smart*, the possibility of reorienting the whole framework in order to support as many wireless protocols as possible has been considered as well. WI-FI, ZigBee, Ultra Wide Band etc. are some of the most popular alternatives to Bluetooth, and many smartphones have built-in capabilities to perform communications under these protocols. Such a framework could be of valuable use for the benchmarking processes of many projects related to wireless communications.

Remote storage of data

Another line of work that could augment the usability of the developed framework would be to provide new mechanisms to store the logged data. The process of connecting the device with the computer to copy the logged files is not very comfortable, so automating this task would come in handy. Instead of a normal PC, the same could be done with a remote FTP server or additional storage alternatives.

In addition to this, if the logged data becomes available at an external place (apart from the smartphone performing the tests) in real time, a live representation of the testing process could be made. However, real-time streaming from the phone to a remote location would provably consume more resources than needed, so the tests' results may become tainted.

Battery and power measurement

The method used to monitor the battery drain rate is not very trustworthy, as it has little to no accuracy in short periods of time. An external mechanism (other than Android's built in *Intent* broadcaster) to measure the batteries' level and power could help to achieve finer grain readings. There are several ongoing researches focused on measuring the power consumption of modern smartphones in which inspiration can be found, so this should not be a great deal of effort. Still, getting ahold of the tools needed may be a bit more complicated.

Likewise, collaborations could be made with ongoing researches such as [Carroll and Heiser, 2010], where the authors perform an almost complete profile for smartphones on the topic of power consumption. By chance, they lack a formal procedure of measuring Bluetooth's consumption, so both parties could benefit from a collaboration.

Corrupt message measurement

The method used to measure the error rate can only identify a corrupted message if there is an inconsistency between the message's payload and the CRC (Cyclic Redundancy Check, or checksum). Therefore, all the messages that contain an error in any of the other fields cannot be identified.

Perhaps, a mechanism to calculate the error rate in a byte-to-byte scale—instead of in messages—could be helpful if the accuracy of the measurements wanted to be improved.

In addition to this, there may be an interest to count the corrupt messages as valid throughput.

Strict testing environment

The tests done in this project have not measured nor taken into account the interferences caused by external factors. An environment that restricts the effects of outer radio signals may yield purer results. It could be interesting, perhaps, to perform some tests inside a *Faraday cage* or in a *screen room* with electromagnetic shielding[1].

Anyway, the currently performed tests are good because they have been done ”in the wild”, in an environment much similar to where the commercial applications and technologies are actually used.

Improved Python scripts

The different scripts that have been developed could be unified and improved into a desktop graphic application. Instead of rewriting the configuration for each new scenario, an abstracted mechanism that requires no re-programming might be interesting as well. Finally, the efficiency of the plotting scripts needs to be revised, as explained in section 9.3.

[1]Electromagnetic shielding is the practice of reducing the electromagnetic field in a space by blocking the field with barriers made of conductive or magnetic materials.

Appendices

A. APPENDIX

Quality check-list

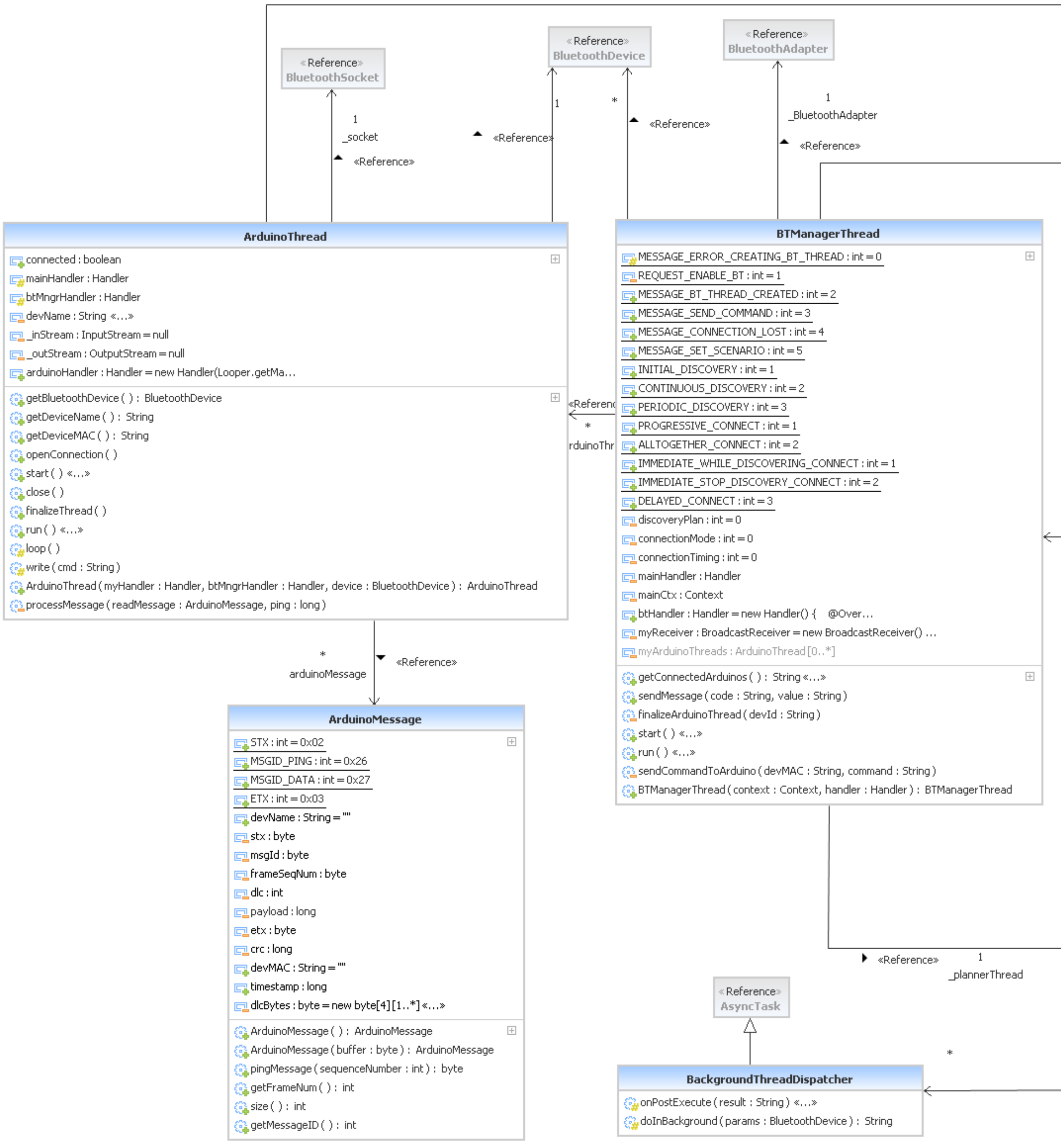
Overall Project Quality Checklist	y/n
Is the statement of scope accurate, consistent and up to date?	
Does the execution of the project go along with the planning?	
Are estimates still accurate or acceptable?	
Is the project schedule fulfillable?	
Is the developed work consistent with the scope?	
Is the system performance and behavior acceptable within the scope?	
Do all required resources including hardware, software, tools, people, and others remain fully available?	
Do all project risks remain controlled?	
Are there new constraints that will affect the product or the project? If so, are they documented?	
Are the stakeholders properly informed on the progress of the project?	
Has a backup for the project's data been recently done?	

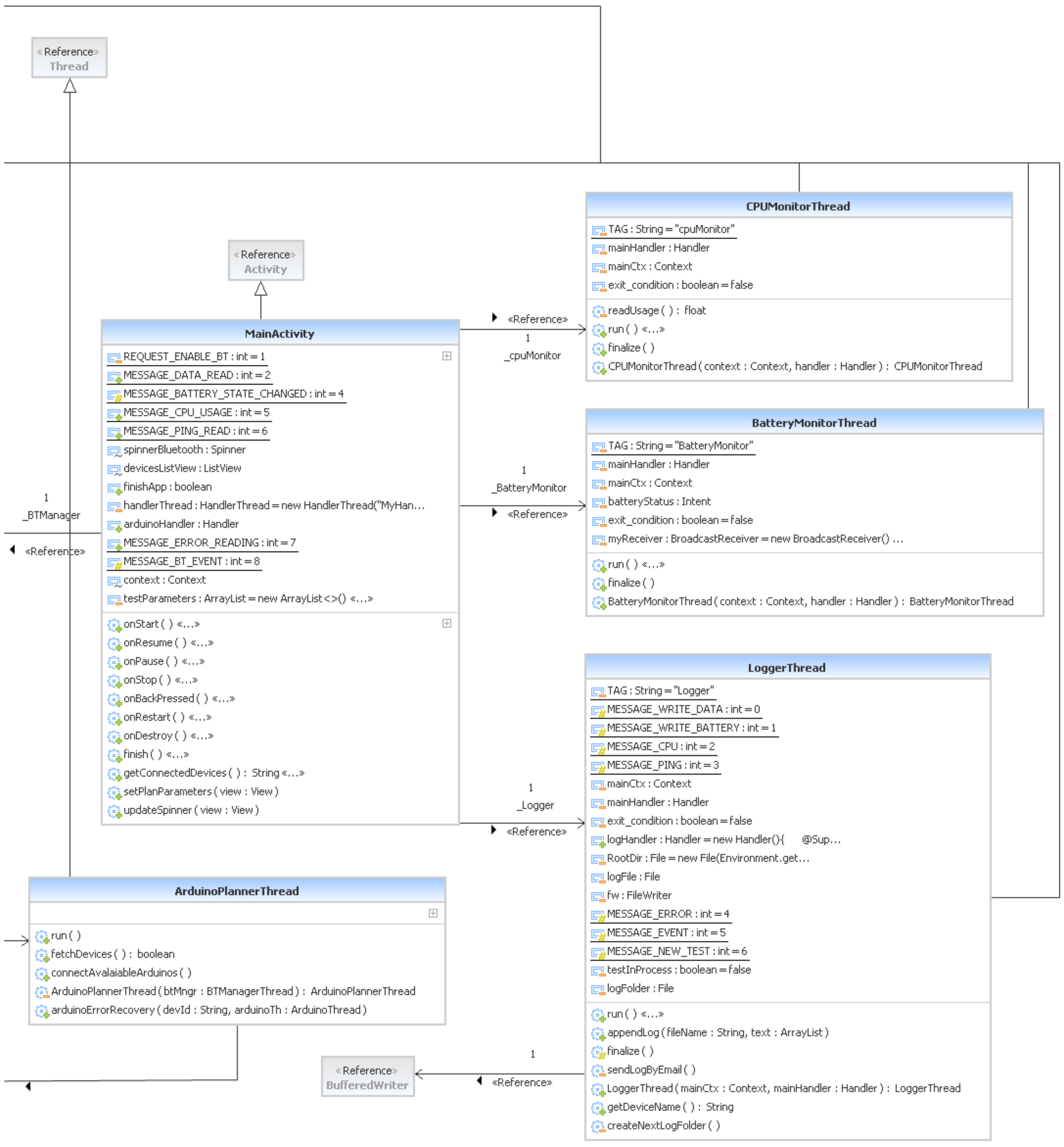
Product Quality Checklist	y/n
The system contains no unfixed bugs or issues	
The functionalities implemented are within the scope of the project	
There is not any malfunctioning hardware device	

B. APPENDIX

UML Class Diagram

Figure B.1: UML class diagram of the Arduino application.


«Reference» BluetoothSocket
«Reference» BluetoothDevice
«Reference» BluetoothAdapter
1
_socket
«Reference»
«Reference»
1
*
«Reference»
1
_BluetoothAdapter
«Reference»
ArduinoThread
connected : boolean
mainHandler : Handler
btMngrHandler : Handler
devName : String «...»
_inStream : InputStream = null
_outStream : OutputStream = null
arduinoHandler : Handler = new Handler(Looper.getMa...
getBluetoothDevice ( ) : BluetoothDevice
getDeviceName ( ) : String
getDeviceMAC ( ) : String
openConnection ( )
start ( ) «...»
close ( )
finalizeThread ( )
run ( ) «...»
loop ( )
write (cmd : String)
ArduinoThread (myHandler : Handler, btMngrHandler : Handler, device : BluetoothDevice) : ArduinoThread
processMessage (readMessage : ArduinoMessage, ping : long)
«Referenc
*
rduinoThr
BTManagerThread
MESSAGE_ERROR_CREATING_BT_THREAD : int = 0
REQUEST_ENABLE_BT : int = 1
MESSAGE_BT_THREAD_CREATED : int = 2
MESSAGE_SEND_COMMAND : int = 3
MESSAGE_CONNECTION_LOST : int = 4
MESSAGE_SET_SCENARIO : int = 5
INITIAL_DISCOVERY : int = 1
CONTINUOUS_DISCOVERY : int = 2
PERIODIC_DISCOVERY : int = 3
PROGRESSIVE_CONNECT : int = 1
ALLTOGETHER_CONNECT : int = 2
IMMEDIATE_WHILE_DISCOVERING_CONNECT : int = 1
IMMEDIATE_STOP_DISCOVERY_CONNECT : int = 2
DELAYED_CONNECT : int = 3
discoveryPlan : int = 0
connectionMode : int = 0
connectionTiming : int = 0
mainHandler : Handler
mainCtx : Context
btHandler : Handler = new Handler() { @Over...
myReceiver : BroadcastReceiver = new BroadcastReceiver() ...
myArduinoThreads : ArduinoThread [0..*]
getConnectedArduinos ( ) : String «...»
sendMessage (code : String, value : String)
finalizeArduinoThread (devId : String)
start ( ) «...»
run ( ) «...»
sendCommandToArduino (devMAC : String, command : String)
BTManagerThread (context : Context, handler : Handler) : BTManagerThread
*
arduinoMessage
«Reference»
ArduinoMessage
STX : int = 0x02
MSGID_PING : int = 0x26
MSGID_DATA : int = 0x27
ETX : int = 0x03
devName : String = ""
stx : byte
msgId : byte
frameSeqNum : byte
dlc : int
payload : long
etx : byte
crc : long
devMAC : String = ""
timestamp : long
dlcBytes : byte = new byte[4][1..*] «...»
ArduinoMessage ( ) : ArduinoMessage
ArduinoMessage (buffer : byte) : ArduinoMessage
pingMessage (sequenceNumber : int) : byte
getFrameNum ( ) : int
size ( ) : int
getMessageID ( ) : int
«Reference»
1
_plannerThread
«Reference» AsyncTask
*
BackgroundThreadDispatcher
onPostExecute (result : String) «...»
doInBackground (params : BluetoothDevice) : String


«Reference» Thread
«Reference» Activity
MainActivity
REQUEST_ENABLE_BT : int = 1
MESSAGE_DATA_READ : int = 2
MESSAGE_BATTERY_STATE_CHANGED : int = 4
MESSAGE_CPU_USAGE : int = 5
MESSAGE_PING_READ : int = 6
spinnerBluetooth : Spinner
devicesListView : ListView
finishApp : boolean
handlerThread : HandlerThread = new HandlerThread("MyHan…
arduinoHandler : Handler
MESSAGE_ERROR_READING : int = 7
MESSAGE_BT_EVENT : int = 8
context : Context
testParameters : ArrayList = new ArrayList<>() «…»
onStart ( ) «…»
onResume ( ) «…»
onPause ( ) «…»
onStop ( ) «…»
onBackPressed ( ) «…»
onRestart ( ) «…»
onDestroy ( ) «…»
finish ( ) «…»
getConnectedDevices ( ) : String «…»
setPlanParameters (view : View)
updateSpinner (view : View)
«Reference» 1 _cpuMonitor
CPUMonitorThread
TAG : String = "cpuMonitor"
mainHandler : Handler
mainCtx : Context
exit_condition : boolean = false
readUsage ( ) : float
run ( ) «…»
finalize ( )
CPUMonitorThread (context : Context, handler : Handler) : CPUMonitorThread
1 _BatteryMonitor «Reference»
BatteryMonitorThread
TAG : String = "BatteryMonitor"
mainHandler : Handler
mainCtx : Context
batteryStatus : Intent
exit_condition : boolean = false
myReceiver : BroadcastReceiver = new BroadcastReceiver() …
run ( ) «…»
finalize ( )
BatteryMonitorThread (context : Context, handler : Handler) : BatteryMonitorThread
1 _BTManager «Reference»
1 _Logger «Reference»
LoggerThread
TAG : String = "Logger"
MESSAGE_WRITE_DATA : int = 0
MESSAGE_WRITE_BATTERY : int = 1
MESSAGE_CPU : int = 2
MESSAGE_PING : int = 3
mainCtx : Context
mainHandler : Handler
exit_condition : boolean = false
logHandler : Handler = new Handler(){ @Sup…
RootDir : File = new File(Environment.get…
logFile : File
fw : FileWriter
MESSAGE_ERROR : int = 4
MESSAGE_EVENT : int = 5
MESSAGE_NEW_TEST : int = 6
testInProcess : boolean = false
logFolder : File
run ( ) «…»
appendLog (fileName : String, text : ArrayList)
finalize ( )
sendLogByEmail ( )
LoggerThread (mainCtx : Context, mainHandler : Handler) : LoggerThread
getDeviceName ( ) : String
createNextLogFolder ( )
ArduinoPlannerThread
run ( )
fetchDevices ( ) : boolean
connectAvalaiableArduinos ( )
ArduinoPlannerThread (btMngr : BTManagerThread) : ArduinoPlannerThread
arduinoErrorRecovery (devId : String, arduinoTh : ArduinoThread)
«Reference» BufferedWriter
1 «Reference»

Bibliography

[bt2, 2001] (2001). *Bluetooth Specification Version 1.1, Part K:5, Serial Port Profile*. STMicroelectronics. https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=8700.

[st2, 2003] (2003). *Push-pull four channel driver with diodes (L293D, L293DD)*. STMicroelectronics. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00000059.pdf.

[hxm, 2010] (2010). *HxM Bluetooth API Guide*. Zephyr Technology, Ph: 1-443-569-3603. Ref. ID: 9700.0031. http://zephyranywhere.com/media/pdf/HXM1_API_P-Bluetooth-HXM-API-Guide_20100722_V01.pdf.

[lin, 2014] (2014). *Linux Programmer's Manual*. Linux. PROC(5) - "process information pseudo-filesystem". http://man7.org/linux/man-pages/man5/proc.5.html.

[Akingbehin, 2005] Akingbehin, Kiumi, A. (2005). Alternatives for short range low power wireless communications. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2005 and First ACIS International Workshop on Self-Assembling Wireless Networks. SNPD/SAWN 2005. Sixth International Conference on*, pages 320–321. IEEE.

[Andrdoid-Developers, 2014a] Andrdoid-Developers (2014a). Andrdoid developers api guides. https://developer.android.com/guide.

[Andrdoid-Developers, 2014b] Andrdoid-Developers (2014b). Andrdoid developers reference. https://developer.android.com/reference/packages.html.

[Bluetooth-SIG, a] Bluetooth-SIG. About the bluetooth sig. http://www.bluetooth.com/Bluetooth/SIG/.

[Bluetooth-SIG, b] Bluetooth-SIG. High speed bluetooth comes a step closer: enhanced data rate approved. https://www.bluetooth.org/About/bluetooth_sig.htm.

[Bluetooth-SIG, c] Bluetooth-SIG. Specification documents: Service discovery. https://www.bluetooth.org/en-us/specification/assigned-numbers/service-discovery.

[Bluetooth-SIG, 2007] Bluetooth-SIG (2007). Bluetooth specification version 2.1 + edr. https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=241363.

[BlueZ,] BlueZ. Official linux bluetooth protocol stack. http://www.bluez.org/.

[Bui, 2012] Bui, T. H. (2012). Performance assessment of android applications. https://software.intel.com/en-us/android/articles/performance-assessment-of-android-applications.

[Bunszel, 2001] Bunszel, C. (2001). Magnetic induction: A low-power wireless alternative. http://defenseelectronicsmag.com/archive/magnetic-induction-low-power-wireless-alternative.

[Cano et al., 2007] Cano, J.-C., Cano, J.-M., Calafate, C., Gonzalez, E., and Manzoni, P. (2007). Evaluation of the trade-off between power consumption and performance in bluetooth based systems. In *Sensor Technologies and Applications, 2007. SensorComm 2007. International Conference on*, pages 313–318. IEEE.

[Cano et al., 2006] Cano, J.-C., Cano, J.-M., González, E., Calafate, C., and Manzoni, P. (2006). Power characterization of a bluetooth-based wireless node for ubiquitous computing. In *Wireless and Mobile Communications, 2006. ICWMC'06. International Conference on*, pages 13–13. IEEE.

[Carroll and Heiser, 2010] Carroll, A. and Heiser, G. (2010). An analysis of power consumption in a smartphone. In *USENIX annual technical conference*, pages 271–285.

[Chakraborty et al., 2010] Chakraborty, G., Naik, K., Chakraborty, D., Shiratori, N., and Wei, D. (2010). Analysis of the bluetooth device discovery protocol. *Wireless Networks*, 16(2):421–436.

[Edith and Gunningberg, 2013] Edith, C.-H. N. and Gunningberg, P. (2013). Pervasive and mobile computing.

[Fainberg and Goodman, 2001] Fainberg, M. and Goodman, D. (2001). Analysis of the interference between ieee 802.11 b and bluetooth systems. In *Vehicular Technology Conference, 2001. VTC 2001 Fall. IEEE VTS 54th*, volume 2, pages 967–971. IEEE.

[Fong et al., 2003] Fong, B., Rapajic, P. B., Hong, G. Y., and Fong, A. C. M. (2003). Factors causing uncertainties in outdoor wireless wearable communications. *Pervasive Computing, IEEE*, 2(2):16–19.

[Frenzel, 2002] Frenzel, L. E. (2002). Wireless pan alternatives to bluetooth. `http://electronicdesign.com/lighting/wireless-pan-alternatives-bluetooth`.

[Gamecho et al., 2013] Gamecho, B., Gardeazabal, L., and Abascal, J. (2013). Combination and abstraction of sensors for mobile context-awareness. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, UbiComp '13 Adjunct, pages 1417–1422, New York, NY, USA. ACM.

[Golmie and Mouveaux, 2001] Golmie, N. and Mouveaux, F. (2001). Interference in the 2.4 ghz ism band: Impact on the bluetooth access control performance. In *Communications, 2001. ICC 2001. IEEE International Conference on*, volume 8, pages 2540–2545. IEEE.

[Google-IO, 2013] Google-IO (2013). Best practices for bluetooth development. `https://developers.google.com/events/io/sessions/326240948`.

[Guerreiro, 2013] Guerreiro, J. (2013). A biosignal embedded system for physiological computing.

[Haartsen, 1998] Haartsen, J. (1998). Bluetooth-the universal radio interface for ad hoc, wireless connectivity. *Ericsson review*, 3(1):110–117.

[Helal, 2002] Helal, S. (2002). Standards for service discovery and delivery. *Pervasive Computing, IEEE*, 1(3):95–100.

[Huang and Rudolph, 2007] Huang, A. and Rudolph, L. (2007). *Bluetooth Essentials for Programmers*. ITPro collection. Cambridge University Press.

[Intel, 2012] Intel (2012). Usb 3.0* radio frequency interference impact on 2.4 ghz wireless devices. `http://www.usb.org/developers/whitepapers/327216.pdf`.

[John Hunter, 2013] John Hunter, Darren Dale, E. F. M. D. m. d. t. (2013). Matplotlib reference guide. `http://matplotlib.org/`.

[Kewney, 2004] Kewney, G. (2004). High speed bluetooth comes a step closer: enhanced data rate approved. `http://www.newswireless.net/index.cfm/article/629`.

[Labiod et al., 2007] Labiod, H., Afifi, H., and De Santis, C. (2007). *Wi-Fi™, Bluetooth™, Zigbee™ and WiMax™*. SpringerLink Engineering. Springer.

[Lin, 2003] Lin, C. C. (2003). What's a tri-state buffer? `http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/CompOrg/tristate.html`.

[Lyytinen and Yoo, 2002] Lyytinen, K. and Yoo, Y. (2002). Ubiquitous computing. *Communications of the ACM*, 45(12):63–96.

[Margolis, 2011] Margolis, M. (2011). *Arduino Cookbook*. O'Reilly Media.

[Martinez, 2009] Martinez, J. L. (2009). Cpu checking. `http://www.pplusdomain.net/cgi-bin/blosxom.cgi/2009/04/02`.

[McDermott-Wells, 2004] McDermott-Wells, P. (2004). What is bluetooth? *Potentials, IEEE*, 23(5):33–35.

[mssaxm, 2013] mssaxm (2013). Don't slurp: How to read files in python. `http://axialcorps.com/2013/09/27/dont-slurp-how-to-read-files-in-python/`.

[Nitsch, 2003] Nitsch, S. (2003). /proc/stat explained. `http://www.linuxhowtos.org/System/procstat.htm`.

[Numpy-Developers, 2013] Numpy-Developers (2013). Numpy reference guide. `http://www.numpy.org/`.

[Perrucci et al., 2011] Perrucci, G. P., Fitzek, F. H., and Widmer, J. (2011). Survey on energy consumption entities on the smartphone platform. In *Vehicular Technology Conference (VTC Spring), 2011 IEEE 73rd*, pages 1–6. IEEE.

[Peterson et al., 2006] Peterson, B. S., Baldwin, R. O., and Raines, R. A. (2006). Bluetooth discovery time with multiple inquirers. In *System Sciences, 2006. HICSS'06. Proceedings of the 39th Annual Hawaii International Conference on*, volume 9, pages 232a–232a. IEEE.

[Robles and Kim, 2010] Robles, R. J. and Kim, T.-h. (2010). Review: context aware tools for smart home development. *International Journal of Smart Home*, 4(1).

[Shepherd, 2001] Shepherd, R. (2001). Bluetooth wireless technology in the home. *Electronics & Communication Engineering Journal*, 13(5):195–203.

[Starner, 2003] Starner, T. E. (2003). Powerful change part 1: batteries and possible alternatives for the mobile market. *Pervasive Computing, IEEE*, 2(4):86–88.

[Sutter, 2005] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210.

[Technology, 2013] Technology, A. C. (2013). Wireless pan alternatives to bluetooth. `www.insidechips.com/public/5837print.cfm`.

[Ververidis and Polyzos, 2008] Ververidis, C. N. and Polyzos, G. C. (2008). Service discovery for mobile ad hoc networks: a survey of issues and techniques. *Communications Surveys & Tutorials, IEEE*, 10(3):30–45.

[Weng et al., 2014] Weng, Z., Orlik, P., and Kim, K. (2014). Classification of wireless interference on 2.4 ghz spectrum.

[Wikipedia, 2014] Wikipedia (2014). Wikipedia, the free encyclopedia.

[Yang and Giannakis, 2004] Yang, L. and Giannakis, G. B. (2004). Ultra-wideband communications: an idea whose time has come. *Signal Processing Magazine, IEEE*, 21(6):26–54.

[Zhu et al., 2005] Zhu, F., Mutka, M. W., and Ni, L. M. (2005). Service discovery in pervasive computing environments. *IEEE Pervasive computing*, 4(4):81–90.