University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Offloading in a mobile environment using Bluetooth Low Energy

Scientific Adviser:
S.l. dr. ing. Laura Gheorghe

Author:
Antonel-George Dobre

Bucharest, 2015

Abstract

The main concern for producers of new generation mobile devices, namely smartphones and accessories, is to keep them as small as possible, mobile for as long as possible and feature-rich. This proves to be a problem for most gadgets, as the implementation of new features and user-friendly interfaces usually have an impact on energy consumption. In this paper a method is proposed that helps prolong battery life of mobile gadgets, by employing a method known as code offloading. This thesis presents the fact that code offloading can also be beneficial when Bluetooth Low Energy is used as a communication channel instead of classic TCP/IP connectivity and provides a framework that helps smartphone application developers offload their most CPU-intensive tasks.

Contents

List of Figures

List of Tables

Notations and Abbreviations

API – Application Programmable Interface
BLE – Bluetooth Low Energy
BLEOffloadingFramework – Bluetooth Low Energy Offloading Framework
IDE – Integrated Development Environment
IPC – Inter-Process Communication
PC – Personal Computer
SDK – Software Development Kit
SDP – Service Discovery Protocol
UUID – Universally Unique Identifier

Chapter 1

Introduction

In past years, mobile devices have encountered a widespread use among technical and regular consumers world wide. This increase in popularity is based on the advent of the Internet and social media, together with easy to use, user-friendly interfaces and applications for hand held devices worldwide. A mobile device is not defined anymore as a strict communication device, but as a hand held computer that acts as an access point to content and information sharing.

Although this growth spark of recent years has lead to over the top technological advances, mobile devices have their limits which is mostly reflected in their size and battery life. The key for a successful device is to provide a user-friendly interface with a rich set of features, ranging from on-the-go connectivity to playing media. The main issue that arises in such a rich environment is the constraints on battery life and the fact that producers need to maintain a balance between usability and efficiency. Most smart phones today run on a typical battery of 1500 mAh [1], mainly because this is a limitation in size. Unlike smart phone technology that has developed drastically in the past few years, battery technology has been evolving for the past century and no such large breakthrough has been discovered in the last 15 years[2].

Developers for and of the smart phone platforms have since realized that they need to bypass the hardware constraints and create either power efficient chipsets and components or create efficient software that provides the selected features that are in demand. A method that can achieve a more efficient energy consumption on smart phone devices would be a mixed approach of power efficient hardware and the ability to communicate with other devices and software, called code offloading.

Computational offloading is a technique used to share the processing power of several devices between each other in order to achieve better performance. Code offloading is a technique that has gained a lot of interest recently due to the possibility of using the *System as a Service* architecture of cloud computing in order to offload resource intensive operations to cloud-based surrogates[3], especially in the case of low-power devices such as smartphones.

This technique usually occurs at the code level, where a mobile application may be partitioned such that some of the more process-intensive tasks or algorithms would be run on separate machines. The partitioning can be done either by the developer, statically, or it can be determined at runtime, dynamically, by a simple linear algorithm that measures the cost of data transfer and the potential offloading gain that can be achieved for certain tasks.

Because the cost might outweigh the benefit gained, offloading should be considered an optional process preferred in mobile operations that require high amounts of processing time and low amounts of data transferred between devices[4]. Previous works have also identified that computational offloading is required mainly by applications that implement graphical rendering, image and video processing techniques[5] [6].

Data binding or task delegation is a model that can be considered a form of offloading where developers create an application based on the loose coupled model. These types of application usually do not function without connectivity to other devices and may follow different schema [7], one of the most common methodology being Web API calls and thus enriching mobile applications with the power of Web technologies.

The main notion of this form of offloading is that instead of doing certain tasks on a mobile device, you create an asynchronous call to another machine that will do the work and then provide a result. It is different from code offloading because the working assumption is that the mobile device cannot do the offloaded task in either a timely fashion or does not have the required technology or security access.

A good example of data binding would be the path generation on an online map service, that could take a long time on a mobile embedded device versus a machine in the cloud that has access to caching algorithms and highly performant subroutines.

Following the notion of computational offloading with task delegation and the need for extended battery life, this paper proposes a system through which the process of code offloading and task delegation can be done efficiently over a low-energy imprint communication channel: Bluetooth Low Energy.

Bluetooth (IEEE 802.15.1) is a technology based on a wireless radio system designed for short-range and low-cost devices in order to replace cables for computer peripherals, such as mice, keyboards, printers, etc. Since its conception, this standard has seen a wide variety of use and has evolved from its main purpose of interconnecting peripherals to creating small, wireless, personal area networks (WPAN) that permit advanced data transfer such as: file sharing, transmitting TCP/IP packets, data streaming over simulated Serial Ports and other uses.

Together with Bluetooth 4.0 specification a new design was proposed for low-energy devices, which represents a trade-off between energy consumption, latency and throughput. This new specification has been dubbed Bluetooth Low Energy and since 2010 it has been implemented in most hand held devices along side Bluetooth, in a setup called dual-mode.

1.1 Project Description

One of the important factors when discussing computational offloading is the communication channel between devices. Most work in this field has been in done with the objective of using the powerful computational model of Cloud Computing[4] in order to establish an offloading system for mobile devices that transfers data and code through Internet connectivity, either through Wi-fi or 3GR wireless technologies. Usually this type of system implies that offloading is possible only when the right connectivity makes possible a certain gain of computational power, which is not always the case.

In order to mitigate the drawbacks of the aforementioned model, this paper proposes a new framework that uses the advantages of Bluetooth Low Energy in order to lower the cost of transferring the data between devices, thus maximizing the gain that a system can achieve when offloading.

As such, we present the BLEOffloadingFramework (Bluetooth Low Energy Offloading Framework), a framework designed for developers of mobile applications that uses low energy communication channels in order to transfer data and code between hand held devices, such as smartphones or tablets, with the specific objective of saving battery life of the desired embedded system.

One of the most energy consuming part in mobile embedded systems is the main Processing Unit. A correlation exists between the battery life of smartphones and the amount of time the

CPU is doing work [1], especially in modern day devices, where processors have a high potential for computational power, but are not especially power efficient. Smartphone operating systems such as Android have additional protections against unnecessary CPU usage such as the Power Wake Lock, in which it creates a link between the screen of the device and the processor and forces the system to go into a sleep mode when the screen is turned off, thus preserving battery [8].

As such, one of the motivation behind offloading and the project presented in this paper would be creating an efficient and easy way to offload computational tasks in order to preserve battery life, with the key objective of extending user experience.

For this purpose, the BLEOffloadingFramework uses the low-energy specification of Bluetooth in order to transfer data between devices. This technology has been proved to be more efficient in transferring small chunks of data between devices [9] and helping the connection setup of the standard BR/EDR Bluetooth.

The framework follows a typical offloading system architecture, but instead of relying on Internet connectivity of the device it uses a wireless personal area network established through Bluetooth Low Energy. In a general offloading scenario when the right conditions are met the compute part of an algorithm can be transferred to the cloud, sending back a result of said computation that can be presented to the user or used in other parts of the application.

The BLEOffloadingFramework takes this concept and applies it to a much smaller range network. The framework consists of an Offloading Server, that can be any Bluetooth enabled device, such as a Personal Computer, a laptop or even an embedded device, such as a Raspberry PI [1]. This server always runs a low-energy advertising technique (described in Section 2.1) in which it promotes its existence to other devices.

If a device picks up on such a server and wants to perform offloading it will create and send a request to the server with the desired method that it wants to offload. The server then decides if it can process that request or not. It will send back to the device a Accept/Reject type of answer, in which Accept states that it can begin the offloading and Reject means that it is too busy at the moment.

Once the request is accepted, the server will compute the selected method and offer a response back to the application.

Experimental results demonstrate that this typical scenario presents an increase in battery life, because the mobile device does relatively low processing on itself.

Because of its lightweight infrastructure and use of low-energy technology, the BLEOffloadingFramework presents a series of advantages over other offloading scenarios. In this paper, the method and implementation of this framework will be presented, together with the context and motivation behind this type of offloading and also the experimental setup used to demonstrate the principles of the framework.

1.2 Project Objectives

This thesis proposes a new system for computational offloading and task delegation by providing a seamless method for code and data transferring through Bluetooth Low Energy. The objectives of this project are as follow:

- To present a new method for computational offloading with the goal of helping developers create energy efficient applications on mobile platforms.

[1] Raspberry PI - a small ARM-based System on a Chip embedded device that can run a lightweight Linux distribution, used mainly for education and small embedded projects

- To use the low-energy technology of Bluetooth in order to demonstrate the principle of code offloading and task delegation for mobile devices.
- To provide developers of mobile applications with an easy method for implementing offloading over Bluetooth, through the use of an Application Programmable Interface (API).
- To make use of at least two offloading methods and provide a comparison between them.
- To determine a testing method for use in comparing offloading solutions based on the framework.
- To be scalable in terms of implementation and methodology.
- To be generic in terms of methodology and architecture and thus not be limited to a certain platform or device.
- To be transparent to both developers and users of this framework and to always ask for permission when offloading is possible so that there is no possibility of privacy obstruction.
- To be secure, thus protecting user data and privacy.

1.3 Thesis structure

In order to present the advantages of an offloading framework over the Bluetooth Low Energy communication channel, this paper first presents the background and related work that has caused the need for such a framework in Section 2.

The main considerations when constructing such an offloading framework, and also the components that have to be considered when designing the system are presented in Section 3. In this section the design for the offloading client and the offload server is presented and also the two main offloading solutions are proposed.

The implementation of the offloading framework, both regarding the Android client and the offload Linux server, is regarded in Section 4.

In order to test the functionality of the system and to demonstrate that there is a possible gain from using offloading over Bluetooth Low Energy, in Section 5 a test setup for the BLEOffloadFramework is presented, together with experimental results and a small guideline for the offloading solutions proposed earlier.

The conclusions of this thesis are drawn in Section 6, where we can see the advantages that offloading brings us, with a note on future implementation marks.

Chapter 2

Background and Related Work

This chapter presents the technical background of the technologies used in constructing the BLEOffloadingFramework. As previously stated, Bluetooth wireless technology is used as a communication channel between offloading devices, as such the key terms of this standard are presented in Section 2.1.

The main consideration of the offloading framework is extending the battery life of embedded devices, with a special focus on smartphone mobile devices. In section 2.2 the Android smartphone operating system is presented and the motivation for choosing this type of devices as the beneficiary for the framework.

In section 2.3 several offloading systems are presented and the main advantages of BLEOffloadingFramework over other systems is detailed.

2.1 Bluetooth Low Energy

Bluetooth(BT) has been a long standing standard for small area wireless communications. Most mobile devices, ranging from PDAs to mobile phones and other gadgets, use this technology in order to communicate effortlessly over short distances, making possible file transfers, contact sharing, wireless audio and video streaming and much more. With the progress of Internet and the Cloud, though, the need for small Personal Area Networks has been reduced, as its drawbacks became more and more obvious - battery life of mobile devices has been reduced and the added overhead of Bluetooth communications is not sustainable, has a low throughput and a small range.

Together with the specification of Bluetooth 4.0, Bluetooth SIG has also announced the standard for Bluetooth Low Energy(BLE)[10]. This standard focuses on a trade-off between energy consumption, latency, piconet size and throughput. The advent of this standard, versus other similar wireless solutions such as ZigBee, is due to the fact that it is applicable in a larger variety of use cases: healthcare devices, small electronics, low power devices, Internet of Things or security measures.

This standard also offers full backwards compatibility, as the added benefit of low-energy transmissions can be used in parallel with the normal Bluetooth 4.0 specification. This is applicable because BLE mainly relies on parameter configuration and short, but consistent, device discovery.

In classic BT applications, when two devices needed to communicate they had to be set in Discoverable mode, identify each other and create a secure connection in a process referred to

as pairing and then follow the specifications of certain Profiles. We can compare this wireless connection capability to the OSI stack, where instead of protocols, we have profiles that specify how to interact with different devices. For example, in order to connect to a Bluetooth enabled Mouse or keyboard and use its facilities the device needs to follow the guidelines of the Human Interface Device (HID) profile. An example of the Bluetooth Low Energy stack can be seen in Figure 2.1.

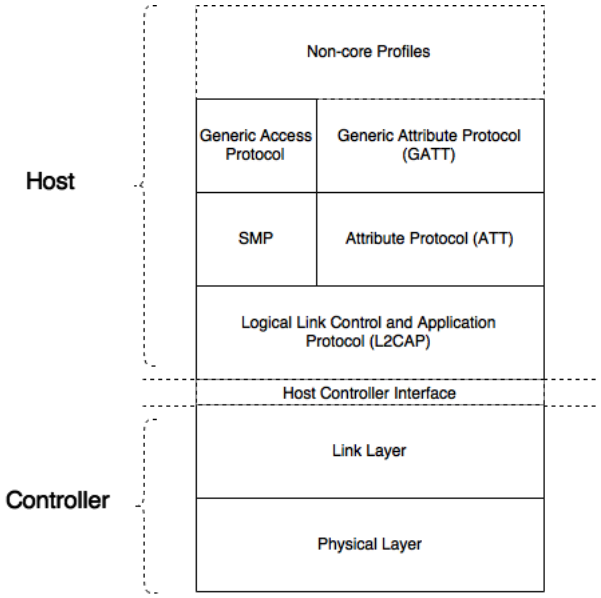

Figure 2.1: Bluetooth Low Energy Stack

Just as in classic Bluetooth[11], the BLE protocol stack is comprised of two main components: the Host and the Controller. The Controller component is usually a small piece of hardware, integrated on a System-on-a-Chip and contains the Physical Layer, which controls wireless modulation, and the Link Layer. The Host component is comprised of several protocols, each working on top of each other:

- Logical Link Control and Adaptation Protocol (L2CAP) - this layer is based on the Bluetooth BR/EDR L2CAP and its prime objective is to multiplex data of higher layer protocols
- Attribute Protocol (ATT)- this protocol defines the communication between two devices, in a client-server architecture. On the server side, a set of data structures called attributes is maintained. The client has the ability to access and modify attributes through requests to the server.
- Generic Attribute Profile (GATT) - this profile determines methods through which devices can perform discovery of services and reading/writing characteristics(a set of data comprised of values and properties).
- Security Manager Protocol (SMP) - This profile cannot be attributed to a specific place in the stack, as it represents the security standards that BLE has to adhere to. The main security modes that are present in BLE are called LE Security Mode 1 and LE Security Mode 2, which work on two separate layers: Link Layer and ATT layer.
- Generic Access Profile (GAP) - This profile specifies the roles and procedures that the device has to adhere to in order to provide the discovery of services and the management of connections.

It is worth mentioning that the GAP profile permits several operating modes, through which several techniques can be established. One such technique is called Advertising. In this technique, a device assumes the role of GAP Broadcaster, which sends small packets of data constantly. These packets contain a string of bytes that are used for identification of the device, the service used and also vendor-specific bytes.

Devices can act as Observers in BLE and such pick up on notifications and packets transmitted by a device in Broadcaster role. This technique is called scanning, and most devices can efficiently scan for advertisement packets by applying filters at the Link Layer and such only receive notifications if there is a specific Broadcaster device in range.

The BLEOffloadingFramework uses this technique in order to identify servers. If a mobile device desires to offload, it will start a scan over Bluetooth Low Energy and waits to see if it picks up any packets from a server. If such a packet exists, then the offloading framework on the client side becomes active and when an application wants to use this system a request will be generated.

When the mobile devices exceeds the server range, it will loose contact with the server (no more advertising packets detected) and such it will shutdown the framework until a new server comes in range.

Even though the main drawback would be the small range of BLE, which is around 10 meters for most devices and is dependent on the hardware, using this model of server detection and data transfer represents a tradeoff between the latency caused by a network connection to a distant cloud server in other offloading systems and the availability of such systems.

2.2 Android

Android is an operating system designed for smartphones, with a focus on usability, touch input and efficiency, both in power and computational abilities. It was first defined as a "software stack for mobile devices that includes an operating system, middleware and key applications" [12]. Today, Android exists on numerous devices, including smart watches, TVs and even cars [13].

This operating system is designed on top of a modified Linux Kernel[1] with a specific stack designed with user applications on the very top. This model permits the enabling of security protocols on the lower levels (closer to the hardware), while providing a feature rich environment for third party developers that deliver content to this specific ecosystem.

Applications for the Android framework are created using a set of tools and helping methods called a Software Development Kit(SDK). Google, the maintainer of Android, offers this SDK in order for developers to create and maintain content for their smart phones. The SDK can also be augmented with an Integrated Development Environment(IDE) that uses the tools offered in order to further simplify the work of developers.

In the top tier of the Android architecture, two main components are of use for developers especially: the Activity and the Service classes. We mention these as classes, because all applications are based on the Software Development Kit, which is available in the Java language.

The Activity class is the main component of an Android application and represents one or more windows that the user can interact with using input methods, be it touchscreen or keyboard and mouse. Activities run in the foreground of the system, have their own life cycle and usually occupies the whole screen, offering the user content and information. An application can contain several Activities and they are organized in a stack fashion: unless specifically

[1] Linux Kernel - http://www.kernel.org

closed, the Activities are pushed in a stack, so that when the user is done with the top level activity, he can immediately have access to the previous activities by pressing the back button.

The Service class represents a background process that can be created in order to offer support for applications. These classes do not have a user interface and are usually started when an Activity explicitly calls to activate the Service or when it performs an operation called Binding. Process Binding in Android represents a form of secure Inter-Process Communication(IPC) between applications and services provided by the Android framework. The Service class is especially useful for the BLEOffloadingFramework, as the model for the offloading client is based on a background application that manages the Bluetooth Low Energy communication with the server.

Because of the availability of this code, the plethora of devices it runs on an the simple interface that it provides, the BLEOffloadingFramework is constructed mainly for the Android Operating System.

2.3 Related Work

This article is based primary on the works of [14] in which an offloading mechanism based on the application life cycle is proposed.

In this model, an application has several states such as interaction with an user via the GUI, processing multiplayer commands, simulations, graphic pipe rendering or terrain generation and some of them are done cyclically by the application. The research is based on the fact that certain states of the above loop can be offloaded completely on other devices or on servers in the cloud. One such application that fits this pattern is OpenTTD.

An example regarding OpenTTD is the offloading of the Artificial Intelligence agents that act as players throughout a game. These agents give out commands and take decisions like a real player and are a core part of the application infrastructure, that use up a lot of processing power, depending on the complexity of the algorithms used. One way to improve on this technique is to search for other states that can be offloaded, besides the agent scripts, or to apply a fine-grain distributed technique.

As an example we can either use the same technique in the GenerateWorld state, which is an initialization state. We can send the settings used to generate over a network communication to a cloud service and generate the world there, the result being a large amount of already processed data that the application can use. While for small worlds this method might bring a very small improvement, or none at all, for large maps can benefit from this technique. By applying fine-grain distributed technique we can mark the methods or parts of code that can be offloaded and offload them. This means that we have to create an abstraction that encapsulates the code, which is usually process-intensive, and offload that segment to a server that knows how to interpret it and simulate results. This technique resembles the Java Remote Method Invocation or Remote Procedure Calls.

The framework presented in this paper proposes a more generic approach to offloading, in that the focus will be more on optimizing the data path between devices together with the capability of applying this framework on almost any type of application.

In [6] a more general offloading solution is proposed. In this method, using the high performance and mobility of cloud technology a virtual machine is created that simulates the exact environment of an application from a mobile device. With such a medium, code translating from mobile device to another machine is straightforward and less error prone. The only drawback that may occur is the fact that data usually has to be transferred across multiple points in order to be processed and that the mobile device has to always be online.

The BLEOffloadingFramework will try to handle these problems by reducing the amount of time spent on data communication, while maintaining the generic aspect of the code that can be offloaded.

Chapter 3

General Architecture

In order to obtain the maximum gain in a computational offloading system, its components have to be designed with efficiency. This chapter presents the environment and data flow of the BLEOffloadingFramework and presents the motivation behind the design.

3.1 Offloading Agents

When discussing code offloading or task delegation two roles can be defined for the devices involved:

- The offloading device (The Client)

  This device is usually a low performance device or a device that pertains a certain gain when it transfers part of its code to another device, or when it delegates a task to a more powerful device. In the context of mobile devices, the technology they comprise has seen a powerful increase in recent years [15]. As consumers tend to move farther away from stationary computing stations, such as Personal Computers (PCs) [16], so too has the focus of recent year research has moved to a more mobile friendly environment.

  Even with new technology being developed every day, mobile systems are met with increasing issues due to physical restraints (e.g. overheating due to closeness of components) or battery life. This represents the main motivation for augmenting the current computational needs of an embedded device, such as a smartphone, through the help of computational offloading.

  The framework presented in this paper considers mobile devices as the main beneficiary of code offloading and task delegation, with the main purpose of reducing energy consumption. Other models can be implemented and the system described here can be applied to other type of devices, due to the generic approach in its construction, but the focus will remain on smartphone technology.

- The computational device (The Server)

  As discussed in Section 2.3, recent studies and research papers have focused on using the new found Cloud Computing technology in order to handle this role. There are certain advantages provided by this infrastructure including availability of said service, a powerful processing model and the interchangeability of data [4], but there are also disadvantages, namely in the fact that the connection used for data transferring may cause a bottleneck when offloading and thus creating a negative user experience.

The BLEOffloadingFramework handles the disadvantages by considering an alternative communication link, which a shorter range, but with greater applicability in terms of gain and user experience. Thus, the computational device that the code is offloaded should be a device that is BLE enabled, but presents greater performance in terms of computing power. Types of devices that respect this condition range from Personal Computers with Bluetooth enabled to laptops or other similar machines.

For the purpose of this paper, the computational device used in experimental results is another embedded device with access to a constant energy source. The motivation behind this choice is to demonstrate that any device can run an offloading server and help preserve the battery life of mobile devices.

Considering these two agents present in the offloading scenario, appendix A refers to the main work flow between devices.

3.2 Offloading Solutions

One of the objectives of this project is to determine several solutions for offloading and task delegation and provide a comparison between these methods. For the BLEOffloadingFramework two methods of code offloading are provided for developers and are described in the following sections.

3.2.1 Task delegation through Remote method invocation

Remote method invocation [17] for the Java language, or remote procedure call for C/C++, is a model through which applications or programs can access and call methods or procedures that are stored on remote machines. The motivation behind these calls are numerous, from security concerns to availability of subroutines and data between machines.

The generic model through which these calls are placed involve setting up a secure connection between the client and the server, transferring the proper method identification and parameters and waiting for the result of that method, a work flow similar to 3.1.

In an offloading context, remote method invocation represents a way to delegate certain tasks to different machines, thus enabling applications to be more power efficient by not consuming a lot of CPU time. It does have a drawback through the fact that the method requires constant online connectivity in order for it to work and produces quite a significant overhead.

In order to combat the disadvantages of the classical model, the framework proposed in this project follows a slightly different approach.

When creating the application, the developer will note the methods or functions that he wishes to offload. These methods will be present both on the target application and on the offloading server. When offloading is possible in an application, the BLEOffloadingFramework client automatically detects the methods it can offload through this model and creates a request to the server in which it states the method identification and parameters. The server will either accept or reject the request based on its availability and the presence of said method in its database. An example work flow can be seen in Figure 3.2.

Using this solution permits applications to run separately, in an offline context and only offloads when the right conditions are met. In the BLE framework, one goal is to offer developers a set of commonly used algorithms that can be used both on the client and server and provide the possibility to switch between client or server processing when the need arises, thus enabling programmers to create energy-efficient applications.

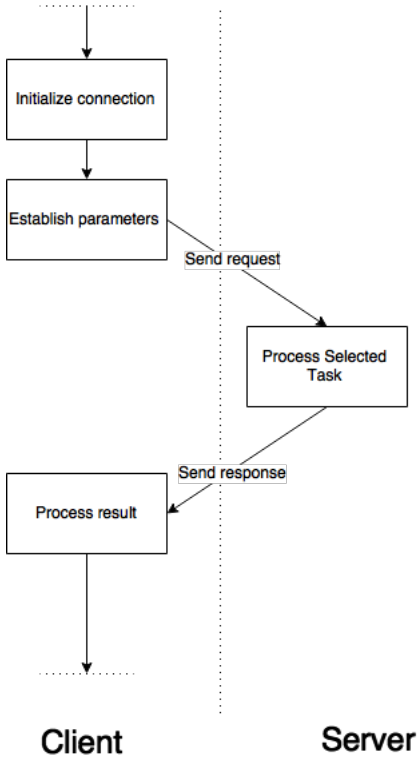

Figure 3.1: A generic work flow for Remote Method Invocation

3.2.2 Data binding of loosed coupled systems

Loosely coupled systems are by definition applications or programs in which all of their components have little or no knowledge of other separate components. The components in such applications communicate through message passing or similar interface based options, where it is not exclusive for one component to know the inner workings of the other. Examples of such applications include applications that rely on web content, where they make asynchronous REST (Representational State Transfer) API calls to web servers in order to obtain data, such as represented in Figure 3.3.

Applying this principle of separation of data between components of application, the BLE-OffloadingFramework permits the inclusion of developer defined requests in order to obtain a better performance for their applications. The offloading server can launch on demand a new process that acts as a separate component for the application. When a request for that specific application is received, the server passes it to the process, thus assuring bidirectional communication between components in an energy efficient way.

This is different from method invocation because the component program that runs on the server can handle much more complex operations such as caching data from the Internet, constant processing of information, clustering and much more.

In Figure 3.4 an example application that uses Loose Coupling offloading is presented. The application's purpose is to show pictures from a certain web server. As such, if offloading is possible it creates a special request to the server in order to see if a fetch picture component is present.

This component is a separate process that belongs to the server and periodically downloads and caches pictures from the applications web server. If the requested image is already cached

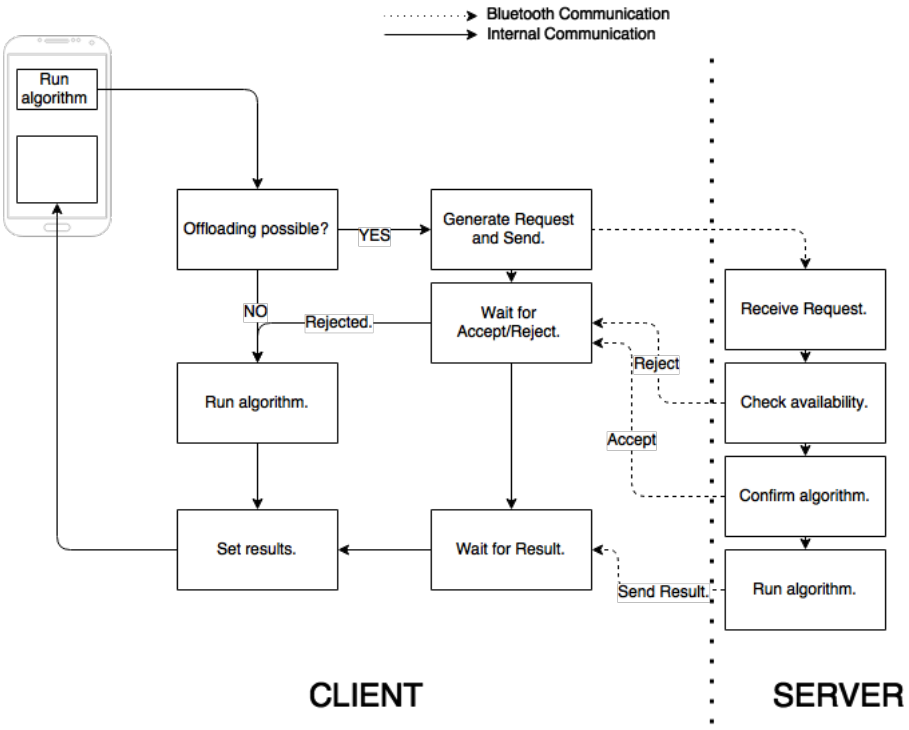

Figure 3.2: Work Flow for Task Delegation through Remote Method Invocation

in the system, then the process will return as a result that specific image. As such, the mobile application saves energy by bypassing the process of creating a Web API request to a remote server and waiting for the download action to complete.

This method is suited for application that requires constant Internet connection for its content, so that instead of always updating it will serve a cached version, which is much faster and in most cases, just as accurate.

The BLEOffloadingFramework client that requires data binding can send a general type request to the server that will be identified and passed to a component program. In this case the developer that wishes to implement this method will have full control over the type and amount of data transferred between devices, leaving connection management and flow control over to the framework, thus providing a simple way to transfer large data over Bluetooth Low Energy.

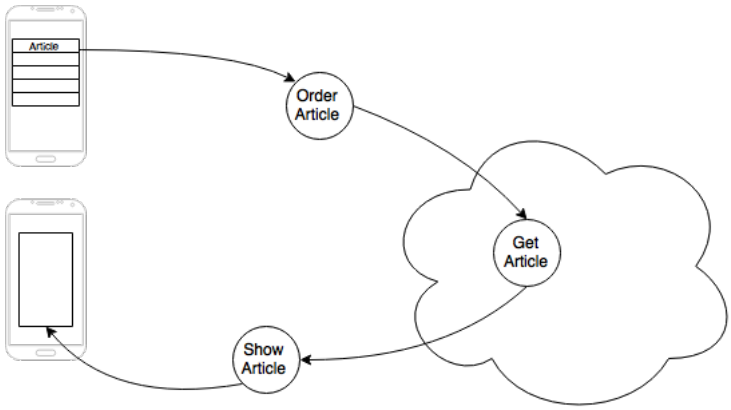

Figure 3.3: Example of a loose coupled application that makes a request to a remote web server

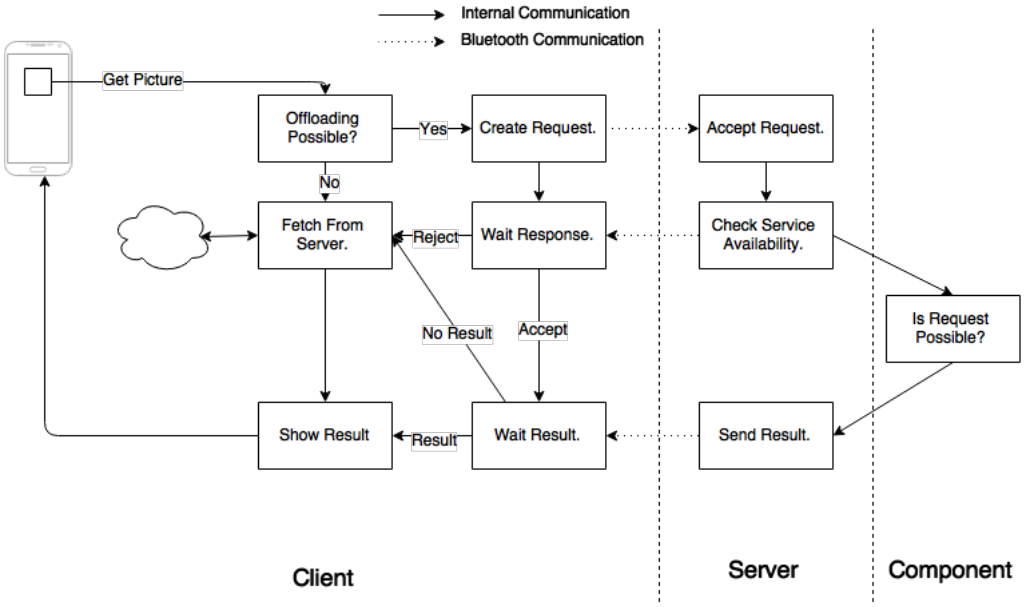

Figure 3.4: Example work flow for an application that uses Loose Coupling offloading method

Chapter 4

Framework Implementation

With the scope of helping developers create power efficient applications through the use of offloading and to demonstrate the inherent gain of code offloading and data binding through Bluetooth Low Energy in this chapter we will present the BLEOffloadingFramework. This framework was constructed for use with the Android Operating System and provides an Application Programmable Interface (API) for developers to use inside their applications in order to offload their more CPU-intensive tasks to a BLE-capable Linux machine.

4.1 The client - Android Framework

As previously state, Android is a mobile operating system tailored specifically for smartphones and other embedded devices, such as TVs or wearables, that offers a wide range of features for users and developers alike. In order to create an application for these devices, a developer may use the Software Development Kit (SDK) that is offered for every version of Android.

The BLEOffloadingFramework uses the Android SDK version 20 and upwards, as this API level has the features needed for effective Bluetooth Low Energy communication.

Because we are discussing a system that is to be used by developers, one prime objective is to be modular: the framework must not interfere with the application and must be coupled or decoupled from the application with ease. The BLEOffloadingFramework presents itself as a Java library that developers can integrate into their application, without changing too much the actual code. This library contains a class called OffloadService, which is an Android Service that provides:

- Searching and establishing connection automatically to Offload Server
- Handling of the lifecycle of the Bluetooth Low Energy socket
- Notifications to the user of the smartphone when offloading is possible
- An interface to developers through which they can perform offloading requests

The offload service is present in the library offered to developers and has to be declared in the application manifest in order to be used (example in Listing 4.1). As well, any application that desires to connect with the framework must declare that it needs permission for BLUETOOTH and BLUETOOTH_ADMIN. This represents all the changes needed in the configuration of the application.

```
1 <uses-permission android:name="android.permission.BLUETOOTH"/>
2 <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
3 <uses-feature android:name="android.hardware.bluetooth_le"
      android:required="false"/>
4 ...
5 <application>
6 ...
7         <service android:name="com.offloadingframework.
              offloadlibrary.OffloadService"/>
8 ...
9 </application>
```

Listing 4.1: Example Application Manifest

In order to start the service, the application needs to perform a request called "binding" to the Android system. The developer can choose to perform this action at application start up, when the needs arises or even permit the user to select when to perform offloading through settings or other similar methods.

Once enabled, the service will start a low energy scan filtering for a specific Universally Unique Identifier (UUID) . This type of identifier is a standard used in software construction in order to enable distributed systems to uniquely identify information without significant central coordination. In Bluetooth Low Energy and normal Bluetooth, UUIDs are used, among other things, to uniquely determine services offered to other devices, such as the Gatt Service, HID service, etc.

The 128-bit value UUID that is used by the BLEOffloading framework is shown in Listing 4.2.

```
1 private static UUID uuid = UUID.fromString("6686416b-aa45-798f
      -1747-680bd1c4a59b");
```

Listing 4.2: UUID used for discovery of Offloading Service

The offload service will perform low energy scans by default for 5 seconds at 10 seconds intervals. The developer can tailor these values by overriding the "offloading_ble_scan_interval" and "offloading_ble_scan_duration" properties of the library.

Once an offloading server with the selected UUID is identified through scans, the service will interrogate the GATT service for its characteristics, identifying as much information about the server, such as its location, its time and date and its availability. If the information provided proves agreeable (such as the availability of offloading services) then the offload service creates a RFCOMM socket to the server.

The RFCOMM socket remains active as long as both devices are in range and periodically sends and receives at least a PING/ACK message. This type of message passed between the mobile phone and the server determines if the server can handle requests. If at any time an ACK message is missed the server is considered lost or out of range and the process of scanning is started anew. An example of the work the service is doing during in its initialization phase is given in Figure 4.1.

Besides managing the connection the Offload Service provides a Message Thread that uses the already established Android Handler architecture in order to receive and send messages to the application. The handler works within the same thread as the service and provides methods designed for message passing and running methods at certain points in time. The ScanningRunnable and "PING/ACK Runnable" run using this handler and its *postDelayed()* method. The *postDealyed()* method allows the running of certain methods after an amount of time has passed. This is very helpful when you require the same task to be executed periodically.

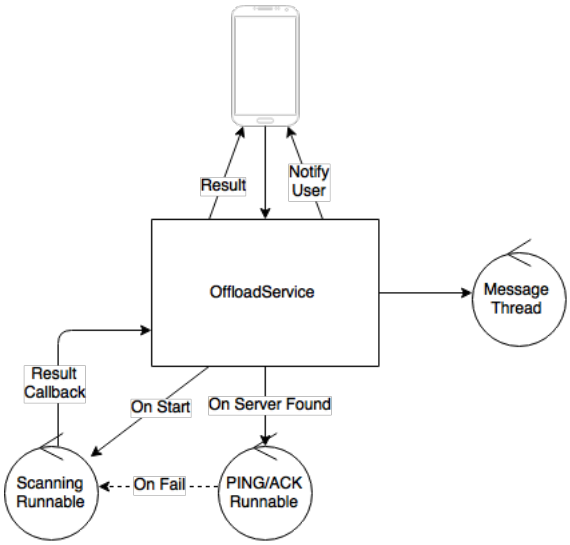

Figure 4.1: Actions started and managed by the OffloadService

Another feature of the Android Handler system is message passing. Every application and service can have a Handler that runs on its main thread that listens to messages and takes certain actions when a message is received. The offload service handler uses these messages in order to provide an interface to the user application.

In the context of BLEOffloadFramework, any request to the offload server is done through message passing to the OffloadService. For the task delegation offload method mentioned in Section 3.2.1, the application can send a message of a certain type, defined in the static class *Contants* and register a Runnable callback in order to wait for the result. This class exposes a number of tasks that can be offloaded, with the certainty that these algorithms exist and can be run on the offload server.

For the case of Data Binding solution, there is a special message type of message called **REQUEST_GENERAL_OFFLOAD** that will deliver to the offload service an OffloadRequest object that will be sent to the server. If the server accepts these types of requests, a callback with the result will be called when offloading is done. The callback will receive a not available answer when offloading is not possible or the server does not accept that request.

As previously stated, the communication between the offload service and the server is done through a RFCOMM socket managed by the client. In order to speed up the process, all the control messages sent through the socket are of the byte primitive type. All the user data is converted to byte form before sending, in a process called encoding. The server decodes the byte string that is sent through the channel and processes the request. All the results or responses sent back to the client are also encoded. This strategy permits the encoding and decoding process to also contain data archiving steps, thus minimizing the time it takes to transfer data across the RFCOMM channel.

An example offloading call on a test application is present in appendix B.

4.2 The server - Linux Embedded System

The offloading server has been designed with the possibility of running on multiple Linux based machines ranging from desktops to laptops or even other embedded devices such as Raspberry PI or MinnowBoard Max.

The server must be Bluetooth Low Energy compatible and uses the open source BlueZ[18] library as means for communication through Bluetooth. This library offers an API and a set of wrappers in order to manage and create Bluetooth connections, manage pairing requests and handle several Low Energy features such as advertising and scan filtering.

4.2.1 Server structure

The BLEOffloadingFramework server project is written in the C language and has the following structure described in Figure 4.2:

```
[19:37 tony>~/Projects/offloadingframework/server $]: tree
.
├── [tony     4.0K]  aux
│   ├── [tony     4.0K]  bin
│   │   └── [tony     2.7K]  mandelbrot_aux.o
│   ├── [tony     3.3K]  mandelbrot_aux.c
│   ├── [tony     3.6K]  mandelbrot.c
│   └── [tony       95]  mandelbrot.h
├── [tony     4.0K]  bin
│   ├── [tony     5.5K]  main.o
│   └── [tony     4.3K]  server.o
├── [tony     4.0K]  comm
│   ├── [tony     4.0K]  bin
│   │   └── [tony      17K]  bluetooth.o
│   ├── [tony      12K]  bluetooth.c
│   ├── [tony     7.0K]  bluetooth.c~
│   ├── [tony     1.7K]  bluetooth.h
│   ├── [tony      853]  bluetooth.h~
│   ├── [tony     2.2K]  offload-server.c
│   └── [tony      119]  offload-server.h
├── [tony       15]  configure
├── [tony     4.0K]  main.c
├── [tony     2.3K]  Makefile
├── [tony      29K]  offload-server
├── [tony     4.0K]  other
│   ├── [tony     4.0K]  bin
│   ├── [tony        0]  decode.c
│   ├── [tony        0]  decode.h
│   ├── [tony        0]  encode.c
│   └── [tony        0]  encode.h
├── [tony     2.2K]  server.c
├── [tony      126]  server.h
└── [tony     4.0K]  third_party
    ├── [tony     4.0K]  bin
    └── [tony     4.0K]  photofeed
        ├── [tony        0]  Makefile
        ├── [tony        0]  photofeed.c
        └── [tony        0]  photofeed.h

10 directories, 26 files
[19:37 tony>~/Projects/offloadingframework/server $]:
```

Figure 4.2: The structure of the BLEOffloadingFramework Server

- Root - *main.c server.c* These files represent the starting point of the server program. Here, the main variables are initialized, checking is performed in order to see if Bluetooth

can be enabled on the system and also contains the main loop that listens for connections and starts additional threads by request.

- Communication API - comm In this directory there are wrappers for the BlueZ library, specifically for the starting of advertising technique with certain parameters, initialization of both LE Gatt Service and SDP service registering. This directory also contains the initialization of the RFCOMM sockets and the mutex synchronized methods for writing and reading from those sockets.
- Auxiliary Programs - aux This directory contains the functions and programs required for the task delegation solution. Each algorithm is contained in its own file with a header file that describes it and the parameters it needs in order to be called.
- Third Party applications - third_party This folder contains a set of programs that should be called when the server initializes. These represent the programs needed for the data binding of loose coupled systems offloading solution. Each program has its own sub-folder and needs to be registered in the server.c file mentioned above.
- Helper functions - other In this directory there are present some additional files that are needed in order to speed up the process, such as UUID transformations and encoding/decoding helper functions.

This structure permits the organization of the server in a scalable way. Developers that want to add methods for Task delegation can do so by adding a C file and a C header file in the *aux* folder. Afterwards they must add the type of request the server will send to the **handle_request()** function referenced in *server.c* main file.

For the Data Binding through loose coupling method, a developer can add their binaries in the *third_party* folder and add the same **handle_request()** logic as above.

The project Makefile is created in such a way to include all the files from a specific folder and also add binaries from third parties in order to create a running executable. Additional libraries and functions that can be used as-is can be inserted inside a separate folder under *third_party*.

4.2.2 Server logic

The server needs to be started with root privileges, as it relies on communication with the Bluetooth kernel module and needs access to the HCI device present on the machine. A special user can be created that has access to the Bluetooth subsystem and such avoid the risk of this server causing security concerns.

When it first boots up, the offload server will try to set the parameters for advertising by calling through HCI the BT device present on the machine. If this is successful, it will then activate the advertising feature on the chipset with those specific parameters. After this phase, it will register a Low Energy GATT service that describes the services provided by the server through Bluetooth and also registers a new service with the Service Discovery Protocol (SDP) , a protocol used by classic Bluetooth in order to identify the capabilities of devices in a piconet.

After registering these services, the initialization of the RFCOMM socket is performed. This socket is set in the *listen* state and we use the Linux *select*[19] mechanism in order to perceive new events on that socket. The *select* mechanism "allows event monitoring on multiple file descriptors, waiting until one or more of the files descriptors become ready for some class of I/O operations"[1]. As such, in the main loop of the server application a *select* is performed on the RFCOMM socket with a maximum number of "child" file descriptors for each subsequent client socket connected. The code behind this operation is described in appendix C.

[1] select - Linux ManPages (the output of "man select" on any Linux system)

Noteworthy is the *handle_request()* near the end of the main loop, as this where the process of decoding and selecting the request happens. Once in this stage, the server application decodes the received message and creates an appropriate response to the client.

If it can handle the request then it will send a preemptive **ACCEPTED** message to the client, in order for it to prepare for the upcoming result. If in its featured database, which is actually a parsable XML file, the method request for offload does exist, then it will send a **REJECTED** message back to the client.

After sending the **ACCEPTED** message, the server will launch a new thread that will serve as the processing environment for the offloaded task. That thread will use the thread safe socket write and read defined in *comm/bluetooth.c* in order to transmit data back to the client.

In the case of loose coupled systems, the server launches during its *init* phase the programs that are coupled to it. These processes have to handle their own lifecycle and have to terminate when the parent process (the offload server) sends a SIGINT process signal. When a socket is opened with the request for that specific program, the socket is passed to the child process.

The child process must not close the socket, as it will cause a reconnect from the smartphone client, and the offload server must not use that specific socket once the request has been sent. This is done using a special mark on the RFCOMM socket that states that the socket is in use by another process. When the work is done on that socket, it will be marked as free by the child process and such the server can continue to receive other requests.

Finally, after each socket iteration, a clean up is performed on the open file descriptors. This clean up is based on the fact that a socket has received updates in the last two minutes. If a socket has not been used in more that two minutes, it will be closed, thus freeing that file descriptor for future use.

To note is the fact that the server does not take into account the type of device it receives updates from. It does not collect identification data of any kind and it acts mostly as a relay for other devices.

Chapter 5

Experimental Results

After defining the architecture of the BLEOffloadingFramework and how it is implemented, in this chapter we will define a testing methodology that can be used to see the benefit of using offloading over Bluetooth in applications.

First off we have to define the terms used in calculating the gain through offloading. Let $\Delta\tau$ be the gain of power resulted from using offloading instead of normal work on the current system. $\Delta\tau$ is expressed in energy consumption over a specific time period and is defined as the energy consumption over a certain period of time required to perform a task (τ_{local}) on a given machine minus the energy required on the same machine to send that task over to another machine and receive the result ($\tau_{offload}$). As such, we can describe the gain as in Formula 5.1:

$$\delta\tau = \tau_{local} - \tau_{offloaded} \tag{5.1}$$

Both τ_{local} and $\tau_{offloaded}$ are experimental variables and should be expressed in the energy it took to perform a certain action over a period of time (this is the measurement of power). In order to calculate the gain in a more general fashion that does not depend on the device that it runs on, we will consider the amount of energy over time as being expressed in battery usage, a relative term used in smatphones and other similar devices to express the amount of battery power left in their devices. This variable is expressed in the percentage of energy left in the battery versus the energy contained in the battery at full capacity. In this case, τ_{local} is expressed in Formula 5.2 with the following relative terms:

$$\tau_{local} = \frac{Percentage\ of\ battery}{Period\ of\ time\ while\ task\ has\ run} \tag{5.2}$$

Because we expresses this in percentages, the calculations are not exact, but by repeating the task and ignoring the little overhead caused by the initialization of variables or stack calling, we can extend the period of time to a magnitude of hours. This permits us to show the larger impact upon the user with ease.

In the same style, we have to define $\tau_{offload}$ in Formula 5.3.

$$\tau_{offload} = \tau_{BLEConnectionHandling} + \tau_{WaitPeriod} + \tau_{BLETransfer} \tag{5.3}$$

Where each term of the equation can be expressed as stated in the Formula 5.4, Formula 5.5 and Formula 5.6.

$$\tau_{BLEConnectionHandling} = \frac{Percentage\ of\ battery}{The\ amount\ of\ time\ the\ connection\ is\ alive\ and\ is\ managed} \tag{5.4}$$

$$\tau_{WaitPeriod} = \frac{Percentage\ of\ battery}{The\ amount\ of\ wait\ time\ for\ offloading\ to\ be\ done} \tag{5.5}$$

$$\tau_{BLETransfer} = \frac{Percentage\ of\ battery}{The\ amount\ of\ time\ it\ takes\ for\ the\ data\ to\ be\ transferred} \tag{5.6}$$

With these notions in mind we will first explain the experimental setup with an example offloading scenario based on task delegation and afterwards determine the impact of the above variables in a comparison between methods.

5.1 Framework Test Setup and Results

In order to determine the experimental gain of the framework a test setup has been devised for the BLEOffloadFramework.

This setup uses as a test case the example application provided in Appendix B. The application shows to the user an image generated with the Mandelbrot set [20] with the set number of pixels for width and height, scaled to fit on the device screen.

The same algorithm is provided through task delegation on the offloading server and is accessible by the application through the **TASK_MANDELBROT** message type provided to the offloading service.

In this setup we will permit an external program to perform user behavior type clicks on the application through a testing process called Instrumentation [21]. "Instrumentation testing allows you to verify a particular feature or behavior with an automated JUnit TestCase. You can launch activities and providers within an application, send key events, and make assertions about various UI elements." [1] This permits us to do the same task (generate the Mandelbrot set) over a period of two hours. During this time we will measure the battery charge by registering a BroadcastReceiver for battery level and logging every two minutes.

During the period, the device will be put in a special mode, called "Airplane Mode" in which all data communication is disabled, in order to minimize the impact of the system on the test results. Bluetooth can be enabled in this mode separately and is enabled during the offload test, thus accounting for the energy consumption of the Bluetooth subsystem of Android.

We will discuss two main cases:

1. Generating Mandelbrot with offloading disabled

   This will cause a separate thread to be launched inside the application process that will generate a Mandelbrot of 1600x1024 pixels with an iteration mark of 100.

2. Generating Mandelbrot with offloading enabled

   This will cause the OffloadService to begin its activity of scanning for an offload server, connecting to it and send a request for a Mandelbrot set with the specified parameters.

The results of running this setup is presented in 5.3.

To note is the fact that although this setup was tailored to a specific test application, the setup steps remain the same for any application developed using the BLEOffloadFramework. These steps are as follows:

[1] http://www.netmite.com/android/mydroid/development/pdk/docs/instrumentation_testing.html

1. Implement application with both offloading and local implementations.
2. Create a common user scenario that would be used inside the application.
3. Implement the user case into a JUnit test case using the Instrumentation Test Framework of Android.
4. Implement inside the test case or in a separate dummy service, a BroadcastReceiver for battery charge updates.
5. Run the test for a predefined amount of time that should show impact. On lower end devices, 2 hours is sufficient, while on higher devices more than 8 hours is advised.
6. Compare results for both offloading enabled and disabled.

The result of the aforementioned test setup can be expressed in a graphic of data points, expressed in the percentage of battery at a given time frame. As such Figure 5.1 expresses the percentage of battery over 60 samples in 2 hours when offloading is disabled. Figure 5.2 represents the same sample size during 2 hours when offloading is disabled. Both figures have been obtained on the same device, an Intel based tablet with a 2 Ghz processor with 1Gb of RAM.

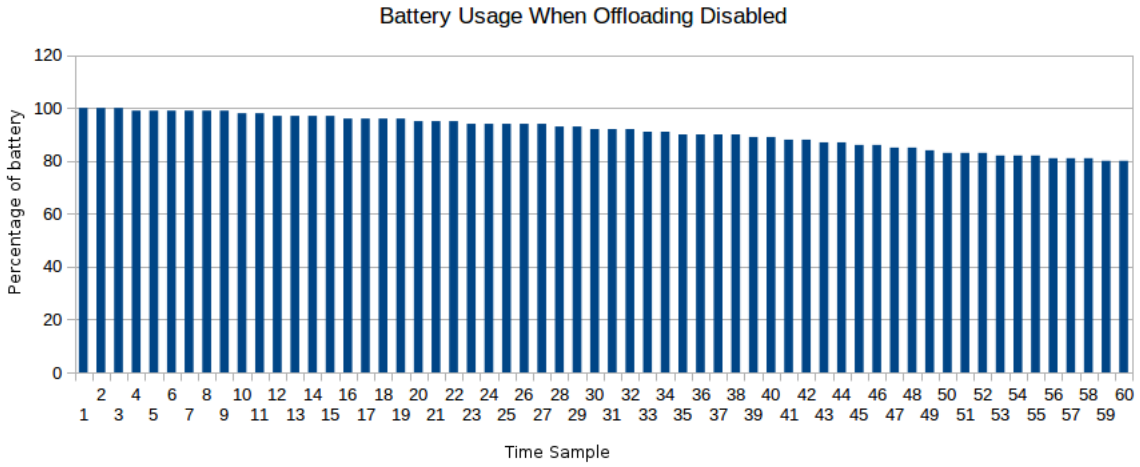

Figure 5.1: A graphic depicting battery usage when offloading is disabled.

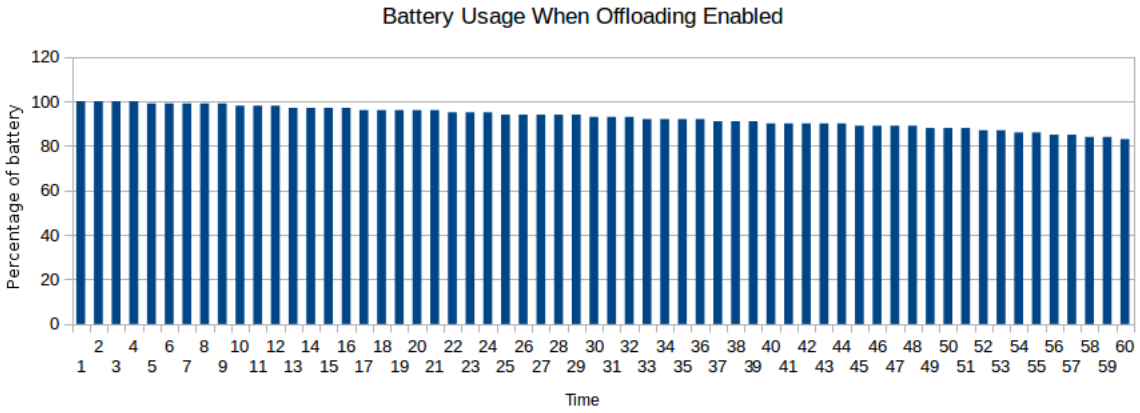

Figure 5.2: A graphic depicting battery usage when offloading is enabled.

Figure 5.3 represents the comparison between Figure 5.1 and Figure 5.2 as the dependency of battery charge over the period of time.

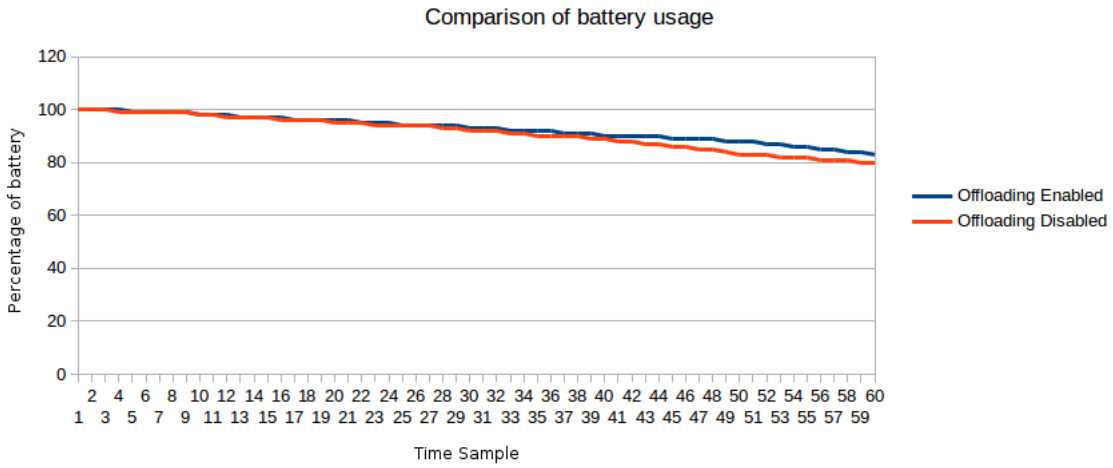

Figure 5.3: Comparison between battery usage in the test setup.

We can define the gain $\Delta\tau$ as the difference between the battery percentage when offloading is enabled and when offloading is disabled, during the whole period of time. In Figure 5.3 we can establish that even if at the beginning no difference can be recognized, while we progress in time the variation of battery charge becomes more and more feasible. This goes to prove that the offloading technique is plausible over Bluetooth Low Energy, pertaining a gain over a longer period of time.

The low difference in battery usage can be attributed to the fact that during offloading, the Bluetooth system is enabled and this provokes more CPU usage and communication between different subsystems.

Also noteworthy is the fact that during the 2 hours period, the Mandelbrot set was generated 1430 times in the case of local computation, versus 1200 times when offloading said computation to a remote server. This is because the applications standard approach was to wait for the image to be completed on the offload server and then shown on the screen through the use of the Bitmap and ImageView API of Android. This downtime has been noted as $\tau_{WaitPeriod}$ in our previous declarations and can be mitigated when offloading through the method of data binding.

In the loose coupled model for offloading the server either has data available, in the case of caching, or it needs very few computation overall. As such the time required for waiting for data to become available is very small, which leads to an increased $\tau_{WaitPeriod}$, thus maximizing the gain overall. To note is the fact what when doing task delegation, the local gain is higher for that specific task, even accounting for the wait period. The purpose of task delegation is to save as much battery life, even if it takes a bit longer for the task to complete.

Because of the advantage the loose coupled method brings, it also has to be considered when developers are using the BLEOffloadingFramework.

When considering using one method or the other there are some guidelines to consider:

- The object that has to be offloaded

  When designing the application with offloading in mind, it is important to note the offloading target. If the target is a task that needs to be performed rarely, but it requires a lot of CPU time in order to be accomplished, it is best to use the task delegation solution

in order to maximize the local offloading gain. If the target requires constant access to data, the overall gain can be higher if the loose coupled solution is used.

- Development speed

  BLEOffloadingFramework offers a multitude of tasks and algorithms already implemented in the server. As such, it is much faster to implement offloading through task delegation than creating a new process to be run by the offload server, as is the case for loose coupled solutions.

- Availability of data

  In the end it goes down to data: does the application needs unprocessed data to be fetched from the Internet and then presented to the user? does that data needs to be processed first? can it be cached for future use by other offloading clients? If the design is data centric, then the loose coupled method is probably best, while as if processing is involved, then task delegation might be the solution.

Chapter 6

Conclusion and Future Development

Mobile phone battery life has been the subject for debate among many enthusiasts and technicians alike, but not many have found a sustainable solution for long lasting smartphone capabilities. In this paper such a solution was proposed, through the use of the BLEOffloadingFramework.

Whereas most offloading frameworks currently being developed use the highly powerful cloud platforms in order to transfer computational power, this presents a limitation through the fact that it may take a long time for code or data to move from the mobile device to the cloud engine and back again.

In order to mitigate this limitation, the BLEOffloadingFramework uses Bluetooth Low Energy a communication channel between devices, thus mitigating the limitations of Internet connectivity and bringing offloading in a more local, well defined space. Besides the gain brought through offloading, this framework also presents a number of advantages to developers and users alike, such as:

- It represents a new method of code offloading through a different communication channel.
- It uses Bluetooth Low Energy, so that the overall energy consumption for Bluetooth Smart devices is heavily decreased.
- Using this framework has shown a gain in battery life even for the simplest of applications.
- It provides two different offloading methods and use cases for both of them.
- It is simple to use and implement as a means of communicating data efficiently between devices.
- It provides a test reference setup in order to present the actual gain the framework brings.
- It is scalable for different machines and platforms.
- It is transparent, providing notifications for the user whenever it will try a connection to an offload server.

As such, the BLEOffloadingFramework presents to be a new method of doing offloading and even though the net gain is not as big as other available platforms or offloading systems, it represents a fresh way to use the power of Bluetooth Low Energy in order to augment already established concepts.

In the future, the BLEOffloadingFramework needs to implement even more features:

- Portability - the framework should be available on other platforms as well, one main step would be to port the offloading server to a mobile device itself, in order to promote the concept of mobile distributed offloading, where devices can share code between them in order to achieve a more expensive task.
- Security - at the moment the framework relies on the encoding and decoding steps in order to provide security of data between devices, but this should change to a more secure channel and investigate the impact of using a more complex socket in the framework.
- Efficiency - the framework requires a more efficient message passing system and should probably benefit more if this was changed completely to a GATT service characteristic reading/writing.
- Content - in order to provide developers with more and more options, content has to be added to the framework in terms of new tasks that can be delegate or third party applications that can be launched.

Bibliography

[1] Denzil Ferreira, Anind K Dey, and Vassilis Kostakos. Understanding human-smartphone concerns: a study of battery life. In *Pervasive computing*, pages 19–33. Springer, 2011.

[2] Megan Geuss. Why your smartphone battery sucks, 2015.

[3] Muhammad Shiraz, Abdullah Gani, Rashid Hafeez Khokhar, and Rajkumar Buyya. A review on distributed application processing frameworks in smart mobile devices for mobile cloud computing. *Communications Surveys & Tutorials, IEEE*, 15(3):1294–1313, 2013.

[4] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, (4):51–56, 2010.

[5] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[6] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[7] Andreas Klein, Christian Mannweiler, Joerg Schneider, and Hans D Schotten. Access schemes for mobile cloud computing. In *Mobile Data Management (MDM), 2010 Eleventh International Conference on*, pages 387–392. IEEE, 2010.

[8] Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. Android power management: Current and future trends. In *Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on*, pages 48–53. IEEE, 2012.

[9] Elke Mackensen, Matthias Lai, and Thomas M Wendt. Performance analysis of an bluetooth low energy sensor system. In *Wireless Systems (IDAACS-SWS), 2012 IEEE 1st International Symposium on*, pages 62–66. IEEE, 2012.

[10] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[11] Jaap C Haartsen. The bluetooth radio system. *Personal Communications, IEEE*, 7(1):28–36, 2000.

[12] Android Developers. What is android, 2011.

[13] Wikipedia. Android (operating system), 2015.

[14] Alexandru-Corneliu Olteanu, Nicolae Tapus, and Alexandru Iosup. Extending the capabilities of mobile devices for online social applications through cloud offloading. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 160–163. IEEE, 2013.

[15] Saeid Abolfazli, Zohreh Sanaei, and Abdullah Gani. Mobile cloud computing: A review on smartphone augmentation approaches. *arXiv preprint arXiv:1205.0451*, 2012.

[16] PC Gartner Lowers. Forecast as consumers diversify computing needs across devices. *Acceesible from: www. gartner. com/it/page. jsp*.

[17] Troy Bryan Downing. *Java RMI: remote method invocation*. IDG Books Worldwide, Inc., 1998.

[18] Marcel Holtmann, Max Krasnyansky, et al. Bluez, official linux bluetooth protocol stack, 2007.

[19] Michal Ostrowski and M Math. *A mechanism for scalable event notification and delivery in Linux*. PhD thesis, University of Waterloo, 2000.

[20] Benoit Mandelbrot. *Fractals and chaos: the Mandelbrot set and beyond*, volume 3. Springer Science & Business Media, 2013.

[21] Martin Kropp and Pamela Morales. Automated gui testing on the android platform. *on Testing Software and Systems: Short Papers*, page 67, 2010.

Appendix A

BLEOffloadingFramework General Use Case

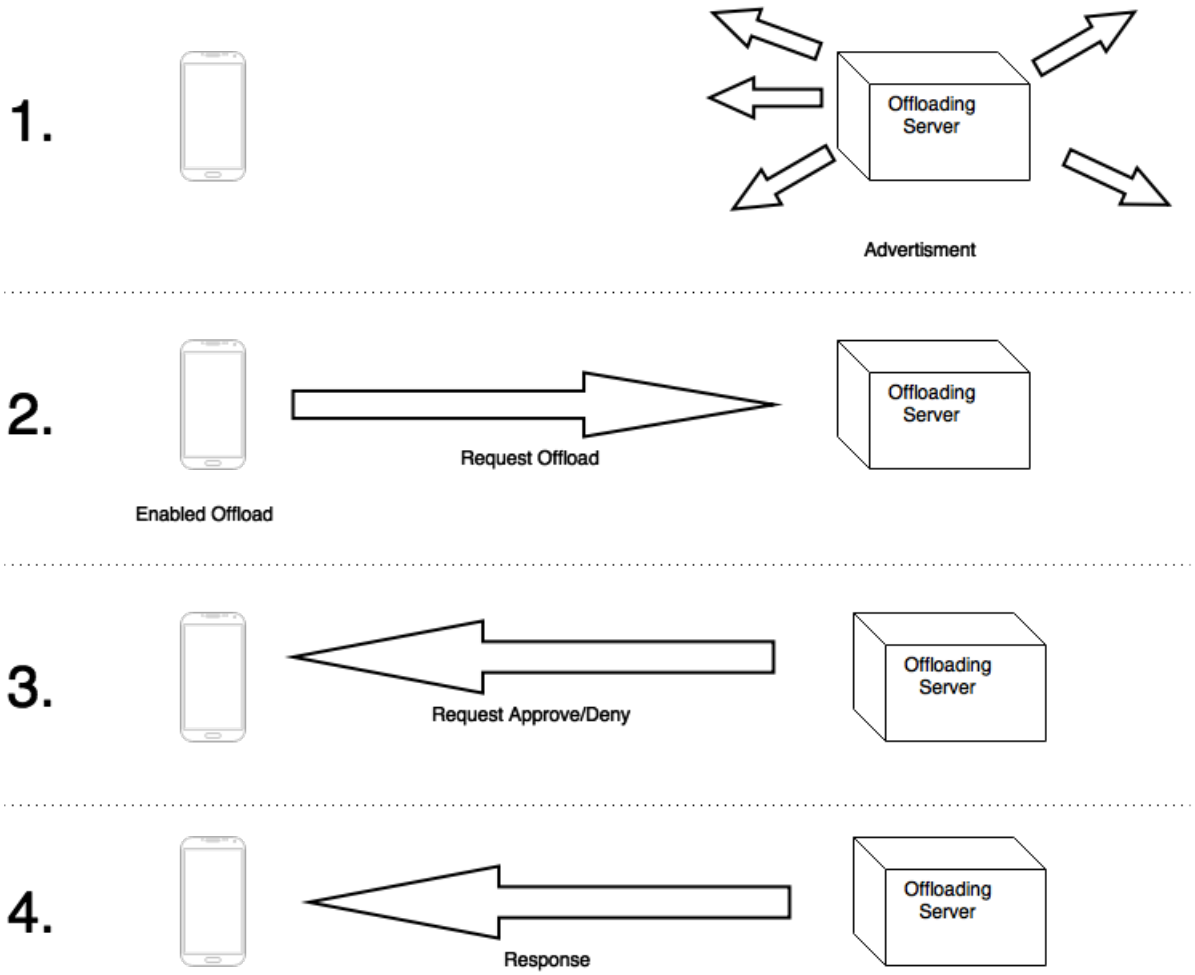

Figure A.1: Offloading Framework

1. Advertising phase: An offloading server uses the technique known as BLE Advertising in

order to promote it's whereabouts to nearby devices.

2. Enabling phase: A mobile device determines there is a server nearby through BLE Scan Filtering on a specific set of UUID. If an application desires to offload a certain part of its code to the server, it will generate a request to that server, which contains the offloading task, either pieces of code or specific task delegation.
3. Request phase: Once the server and mobile device know of each other and the mobile device generates a request, the server can respond with either Accept or Reject, stating that offloading is either possible or, respectively, not possible. If offloading is possible, the mobile device will wait for the results of the selected task.
4. Response phase: If the offloading was possible and successful, then the server will send back a response to the device containing the result of said computation

Appendix B

Example Client implementation and work flow of the BLEOffloadingFramework

In order to present the benefits of the BLEOffloadingFramework, a sample application has been constructed. This application is composed of a single Activity, presented in Figure B.1.

This application, by default generates an image based on the Mandelbrot Set and shows this image on the screen. When the ToggleButton is set for offloading, the application requests the Bluetooth subsystem to start and also starts the OffloadService with an Intent call such as in Listing B.1.

```
1 Intent intent = new Intent(this, OffloadService.class);
2 startService(intent);
```

Listing B.1: Starting the OffloadService through an Intent

When the service receives the start intent, it will run an onStartCommand(), presented in Listing B.2, that:

1. Checks to see if Bluetooth is enabled and obtains a link to the BluetoothAdapter service.
2. Starts a main service thread that remains active while the service is active.
3. Instantiates a handler on the above thread.
4. Obtains a reference to the Bluetooth Low Energy Scanner API.
5. Starts the ScanRunnable thread.

```
1 @Override
2     public int onStartCommand(Intent intent, int flags, int startId)
          {
3
4         mContext = getApplicationContext();
5         final BluetoothManager bluetoothManager = (BluetoothManager)
              mContext.getSystemService(Context.BLUETOOTH_SERVICE);
6         mBluetoothAdapter = bluetoothManager.getAdapter();
7
8         Log.d(LOG_NAME, "Started Service.");
9
```

Figure B.1: The sample application main Activity.

```
10          HandlerThread thread = new HandlerThread("
                ServiceStartArguments", Process.
                THREAD_PRIORITY_BACKGROUND);
11          thread.start();
12
13          mServiceLooper = thread.getLooper();
14          mServiceHandler = new OffloadServiceHandler(mServiceLooper);
15
16
17          Message msg = mServiceHandler.obtainMessage();
18          msg.arg1 = Constants.START_SCANNING;
19
20          if(Build.VERSION.SDK_INT >= 21) {
21              mLEScanner = mBluetoothAdapter.getBluetoothLeScanner();
22              settings = new ScanSettings.Builder()
23                      .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
24                      .build();
25              filters = new ArrayList<ScanFilter>();
```

```
26          }
27
28          mServiceHandler.sendMessage(msg);
29
30          return Service.START_NOT_STICKY;
31      }
```

Listing B.2: onStartCommand() of OffloadService

The ScanRunnable object is called once 5 seconds and it's main function is to start and stop the scan for advertising packets from different sources. Once a server is found, a callback is called in order to register devices that are suitable offload servers. The code for this is presented in Listing B.3.

```
1  private Runnable scanForServerRunnable = new Runnable() {
2          @Override
3          public void run() {
4              scanForServer();
5              mServiceHandler.postDelayed(this, SCAN_THREAD_DELAY);
6          }
7      };
8
9  public void scanForServer(){
10
11         if(mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled
               ()){
12             return;
13         }
14
15         //stop scanning if there already is a socket
16         if(offloadSocket != null && offloadSocket.isConnected())
17             return;
18
19         Log.d(LOG_NAME, "Started␣Scanning.");
20
21         mServiceHandler.postDelayed(new Runnable() {
22             @Override
23             public void run() {
24                 if (Build.VERSION.SDK_INT < 21) {
25                     mBluetoothAdapter.stopLeScan(mLeScanCallback);
26                 } else {
27                     mLEScanner.stopScan(mScanCallback);
28                 }
29             }
30         }, 10000);
31
32
33         if(Build.VERSION.SDK_INT < 21){
34             mBluetoothAdapter.startLeScan(mLeScanCallback);
35         }else{
36             mLEScanner.startScan(mScanCallback);
37         }
38
39     }
40
```

```
41 private ScanCallback mScanCallback = new ScanCallback() {
42             @Override
43             public void onScanResult(int callbackType, ScanResult
                   result) {
44                 Log.d(LOG_NAME, "Found␣advertisement:␣" + result.
                       toString());
45                 if(foundServer) return;
46
47                 ScanRecord record = result.getScanRecord();
48                 SparseArray<byte[]> advPacket = record.
                       getManufacturerSpecificData();
49
50                 StringBuilder buffer = new StringBuilder();
51
52                 for (int i = 0; i < advPacket.size(); i++) {
53                     for (byte b : advPacket.valueAt(i)) {
54                         buffer.append(String.format("%02X␣", b));
55                     }
56                 }
57
58                 if(result.getDevice().getUuids() != null)
59                     for(int i = 0; i < result.getDevice().getUuids()
                           .length; i++){
60                         Log.e(LOG_NAME, "UUID:␣" + result.getDevice
                               ().getUuids()[i]);
61                     }
62
63                 Log.d(LOG_NAME, "PACKET:␣" + buffer.toString());
64                 if (buffer.toString().contains("48␣45␣4C␣4C␣4F␣57␣4F
                       ␣52␣4C")) {
65
66                     BluetoothDevice device = result.getDevice();
67                     handleSocket(device);
68
69                 }
70
71             }
72
73             @Override
74             public void onBatchScanResults(List<ScanResult> results)
                   {
75
76                 for (ScanResult result : results) {
77
78                     Log.d(LOG_NAME, "Found␣advertisement␣in␣batch:␣"
                           + result.toString());
79
80                     if(foundServer) return;
81
82                     ScanRecord record = result.getScanRecord();
83                     SparseArray<byte[]> advPacket = record.
                           getManufacturerSpecificData();
84
```

```
85                      StringBuilder buffer = new StringBuilder();
86
87                      for (int i = 0; i < advPacket.size(); i++) {
88                          for (byte b : advPacket.valueAt(i)) {
89                              buffer.append(String.format("%02X␣", b))
                                    ;
90                          }
91                      }
92
93                      Log.d(LOG_NAME, "PACKET:␣" + buffer.toString());
94                      if (buffer.toString().contains("48␣45␣4C␣4C␣4F␣
                            57␣4F␣52␣4C")) {
95                          handleSocket(result.getDevice());
96                          return;
97                      }
98
99                  }
100
101             }
102
103             @Override
104             public void onScanFailed(int errorCode) {
105                 Log.d(LOG_NAME, "Scan␣failed.");
106                 foundServer = false;
107             }
108         };
```

Listing B.3: The ScanForServerRunnable and its callbacks

If a suitable server is found, then a call to the *handleSocket()* method is performed, present in Listing B.4. In this method the main Input and Output objects are defined and a call to the PING/ACK runnbale is performed.

```
1  public void handleSocket(BluetoothDevice device){
2
3          if(offloadSocket != null && offloadSocket.isConnected()){
4              return;
5          }
6
7          try {
8              offloadSocket = device.
                   createInsecureRfcommSocketToServiceRecord(uuid);
9              offloadSocket.connect();
10             socketPrintWriter = new PrintWriter(offloadSocket.
                   getOutputStream(), true);
11             socketBufferedReader = new BufferedReader(new
                   InputStreamReader(offloadSocket.getInputStream()));
12
13         } catch (IOException e) {
14             //e.printStackTrace();
15             Log.e(LOG_NAME, "Could␣not␣connect␣to␣server.");
16             foundServer = false;
17             return;
18
19         }
```

```
20
21                  /*Present a notification to the user.*/
22          showNotification();
23          //set the variable used to check if a connection is
                available
24          foundServer = true;
25
26          mServiceHandler.removeCallbacks(sanityCheckRunnable);
27          mServiceHandler.removeCallbacks(scanForServerRunnable);
28
29      }
```

Listing B.4: The **handleSocket()** method

Appendix C

Offload Server Main Loop

```
1  void start_main_loop_rfcomm(){
2      int result = -1;
3      int i = 0;
4      fd_set afds;
5      fd_set rfds;
6      struct timeval tv;
7      int clientRfSock;
8      struct sockaddr_rc rem_addr = {0};
9      socklen_t rfcommConnInfoLen;
10     socklen_t sockAddrLen;
11     bdaddr_t clientBdAddr;
12
13     printf("Starting main loop and listening for rfcomm sockets..\n"
           );
14     while(1){
15         FD_ZERO(&afds);
16         FD_SET(rfcomm_socket, &afds);
17         tv.tv_sec = 1;
18         tv.tv_usec = 0;
19
20         result = select(FD_SETSIZE, &afds, NULL, NULL, &tv);
21         if(result < 0){
22           printf("select doesn't work.\n");
23           break;
24         }
25
26         for(i = 0; i < FD_SETSIZE; ++i)
27           if(FD_ISSET(i, &afds)){
28              if ( i == rfcomm_socket){
29                 /*Connection Request on original socket */
30                 sockAddrLen = sizeof(rem_addr);
31                 clientRfSock = accept(rfcomm_socket, (struct
                       sockaddr *) &rem_addr, &sockAddrLen);
32                 if(clientRfSock < 0){
33                   printf("Error accepting a socket."); break;}
34                   FD_SET(clientRfSock, &afds);
35                 }
```

```
36                  else{
37                    /* If socket is not available do not read from it
                         or handle requests */
38                    if(socket_available(i) == 0) continue;
39
40                    /*Data arriving on already connected socket*/
41                    char buffer[MAX_BUFFER_SIZE];
42
43                    mutex_lock(client_mutex(i));
44                    int read_bytes = read(i, buffer, MAX_BUFFER_SIZE);
45                    mutex_unlock(client_mutex(i))
46
47                    printf("%s\n", buffer);
48                    handle_request(i, buffer, read_bytes);
49                  }
50              }
51
52              clean_up_fds();
53          }
54  }
```

Listing C.1: Main Loop of BLEOffloadingFramework server

Index