

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

Maecenas elementum venenatis dui, sit amet
vehicula ipsum molestie vitae. Sed porttitor
urna vel ipsum tincidunt venenatis. Aenean
adipiscing porttitor nibh a ultricies. Curabitur
vehicula semper lacus a rutrum.

Quisque ac feugiat libero. Fusce dui tortor,
luctus a convallis sed, lacinia sed ligula.
Integer arcu metus, lacinia vitae posuere ut,
tempor ut ante.

Abstract

Here goes the abstract about MySuperProject. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean aliquam lectus vel orci malesuada accumsan. Sed lacinia egestas tortor, eget tristique dolor congue sit amet. Curabitur ut nisl a nisi consequat mollis sit amet quis nisl. Vestibulum hendrerit velit at odio sodales pretium. Nam quis tortor sed ante varius sodales. Etiam lacus arcu, placerat sed laoreet a, facilisis sed nunc. Nam gravida fringilla ligula, eu congue lorem feugiat eu.

Contents

Acknowledgements	i
Abstract	ii
1 Introduction	1
1.1 Project Description	2
1.2 Project Objectives	4
2 Background and Related Work	5
2.1 Bluetooth Low Energy	5
2.2 Android	7
2.3 Related Work	7
3 General Architecture	9
3.1 Offloading Agents	9
3.2 Offloading Solutions	10
3.2.1 Task delegation through Remote method invocation	10
3.2.2 Data binding of loosed coupled systems	11
4 Project Implementation	14
4.1 The client - Android Framework	14
4.2 The server - Linux Embedded System	17
5 Experimental Results	21
5.0.1 Test setup	21
6 Conclusion and Future Development	22
A BLEOffloadingFramework General Use Case	23
B BLEOffloadingFramework General Use Case	25
C GATT Service of the BLEOffloadingFramework	26
D SDP Service Registration for the BLEOffloadingFramework	28
E BLEOffloadingFramework Offload Server Logic	30

List of Figures

1.1	A general view on offloading	3
3.1	A generic work flow for Remote Method Invocation	10
3.2	Work Flow for Task Delegation through Remote Method Invocation	11
3.3	Example of a loose coupled application that makes a request to a remote web server	12
3.4	Example work flow for an application that uses Loose Coupling offloading method	12
4.1	Actions started and managed by the OffloadService	16
4.2	The structure of the BLEOffloadingFramework Server	17
A.1	Offloading Framework	24

List of Tables

Notations and Abbreviations

API – Application Programmable Interface

BLE – Bluetooth Low Energy

BLEOffloadingFramework – Bluetooth Low Energy Offloading Framework

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 [7], 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 such no large breakthrough has been discovered in the last 15 years[9].

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[18], 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[14]. Previous works have also identified that computational offloading is required mainly by applications that implement graphical rendering, image and video processing techniques[3] [2].

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 [13], 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 this 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 advance 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 done with the objective of using the powerful computational model of Cloud Computing[14] 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 [7], 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 [4].

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 [15] 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 (such as the one described in 1.1). 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.

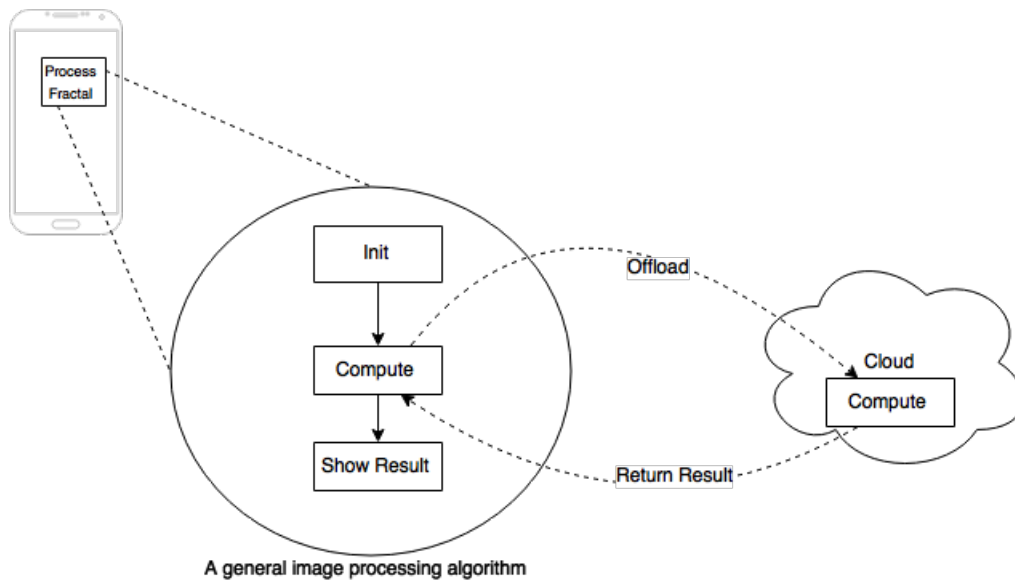


Figure 1.1: A general view on offloading

The BLEOffloadingFramework takes this concept and applies it to a much smaller range network. An example of how the framework works is depicted in appendix B.

As you can see from the appendix, 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

¹Raspberry 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

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.
- 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.

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 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[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.

TODO:

insert BLE stack picture here.

Just as in classic Bluetooth[11], the BLE protocol stack consists of two main parts: The Host and the Controller. The Controller contains the Physical Layer and the Link Layer and is integrated on a small System on A Chip with radio capabilities. The Host part of BLE usually runs on an application processor (such as the processor in a mobile phone) and contains upper layers of functionality such as:

- Logical Link Control and Adaptation Protocol (L2CAP) - this layer is an optimized and simplified protocol based on the Bluetooth L2CAP, with the main goal of multiplexing data of other higher layer protocols.
- Attribute Protocol (ATT)- this protocol defines the communication between two devices playing the role of client and server. The server maintains a set of attributes, where an attribute is a data structure with information managed by GATT. The client can access the server's attributes by sending requests, while the server responds with response messages. These type of messages follow a stop-and-wait scheme.
- Generic Attribute Profile (GATT) - this profile defines a framework that uses the ATT for the discovery of services and the exchange of characteristics between devices. A characteristic can be defined by a set of items with values and properties.
- Security Manager Protocol (SMP) - BLE offers various security services for protecting information exchanged between devices, but most of the supported security services can be categorized in LE Security Mode 1 and LE Security Mode 2, which provide security at the Link Layer and ATT layer, respectively.
- Generic Access Profile (GAP) - This profile specifies device roles, modes and procedures for the discovery of devices and services, the management of connection establishment and security.

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" [5]. Today, Android exists on numerous devices, including smart watches, TVs and even cars [19].

This operating system is designed on top of a modified Linux Kernel¹ 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.

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

2.3 Related Work

This article is based primarily on the works of [16] 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.

¹Linux Kernel - <http://www.kernel.org>

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 [2] 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 [1]. As consumers tend to move farther away from stationary computing stations, such as Personal Computers (PCs) [8], 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 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 [14], 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 the need of a constant energy source. The motivation behind this choice is to demonstrate that any computer like 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 B 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 [6] 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.

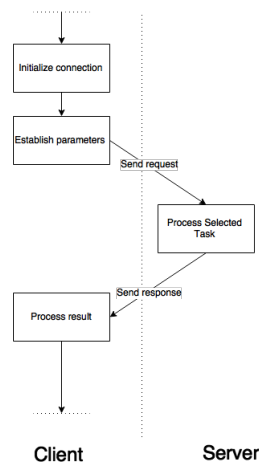


Figure 3.1: A generic work flow for Remote Method Invocation

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 always 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 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.

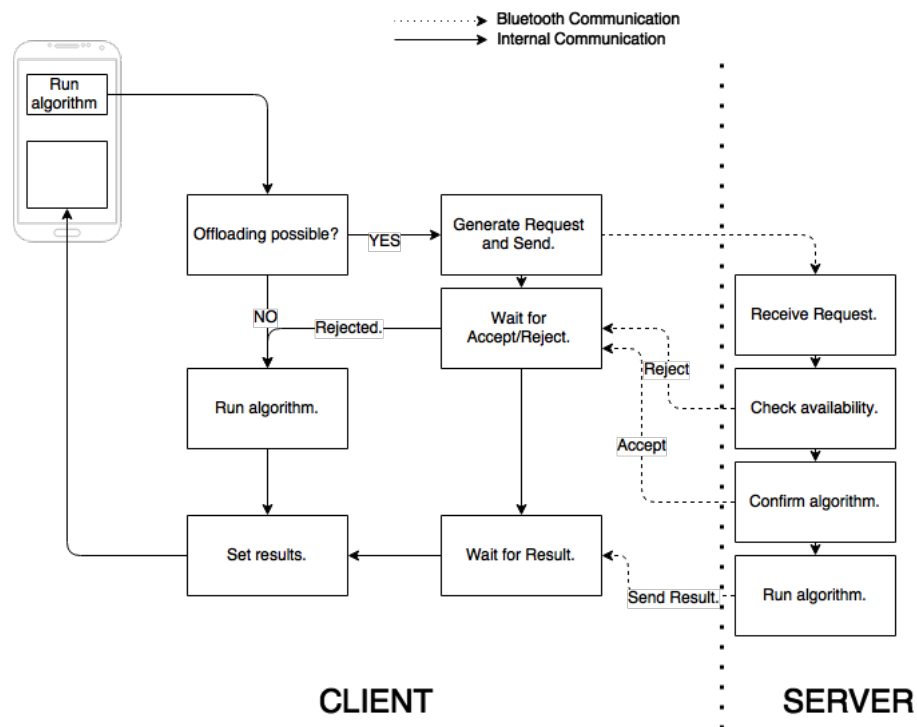


Figure 3.2: Work Flow for Task Delegation through Remote Method Invocation

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.

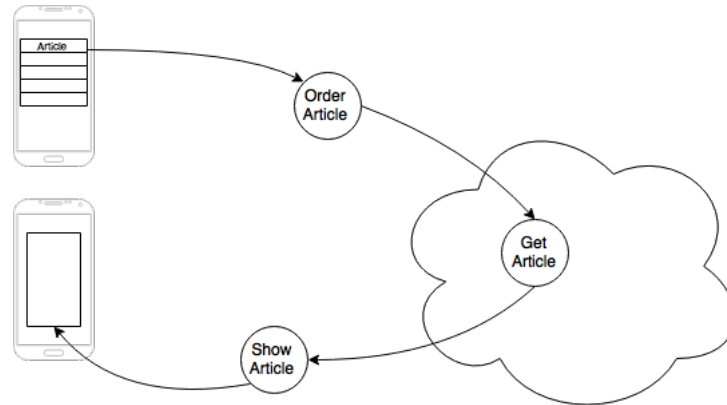


Figure 3.3: Example of a loose coupled application that makes a request to a remote web server

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.

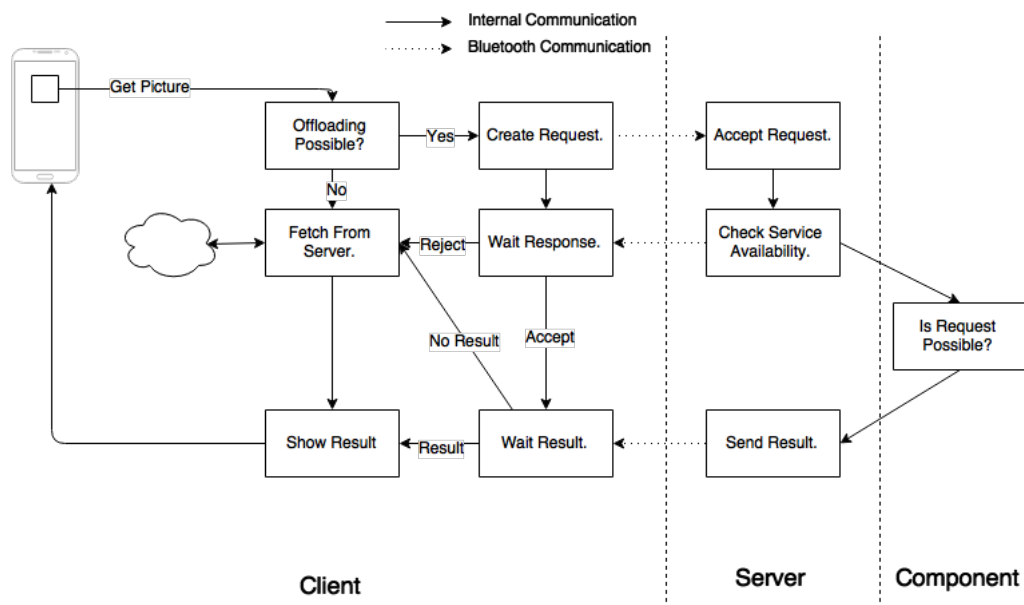


Figure 3.4: Example work flow for an application that uses Loose Coupling offloading method

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 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 it's content, so that instead of always updating it will serve a cached version, which is much faster and in most cases, just as accurate.

Chapter 4

Project 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 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 co-ordination. 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 [D.1](#).

```

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 such some 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

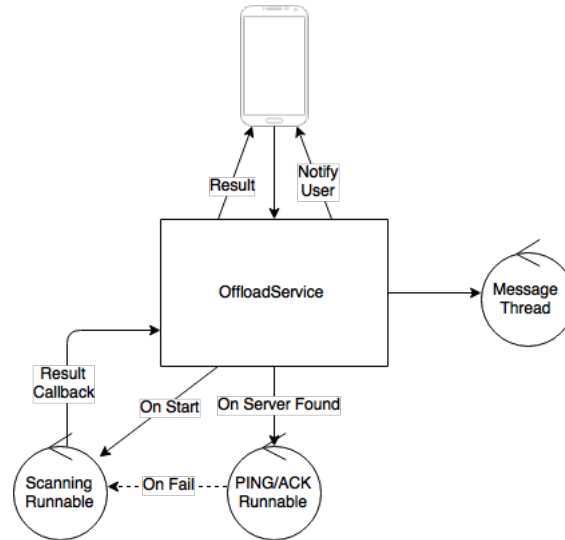


Figure 4.1: Actions started and managed by the OffloadService

ScanningRunnable and "PING/ACK Runnable" run using this handler and its *postDelayed()* method. The *postDelayed()* 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.

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 ??, the application can send a message of a certain type, defined in the static class *Constants* 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 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 ??.

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[12] 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.

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

```

tony@purgatory:~/Projects/offloadingframework/server$ tree
.
├── aux
│   ├── a.out
│   ├── mandelbrot_aux.c
│   ├── mandelbrot.c
│   ├── mandelbrot.h
│   └── new1.ppm
├── comm
│   ├── bluetooth.c
│   ├── bluetooth.h
│   ├── offload-server.c
│   └── offload-server.h
├── configure
├── main.c
├── Makefile
├── other
│   ├── decode.c
│   ├── decode.h
│   ├── encode.c
│   └── encode.h
├── server.c
├── server.h
└── third_party
    └── photofeed
        ├── Makefile
        ├── photofeed.c
        └── photofeed.h

5 directories, 21 files
tony@purgatory:~/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.

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. Examples of both these services are present in appendix C and appendix D.

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*[17] 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 Listing 4.3.

```

1 void start_main_loop_rfcomm() {
2     int result = -1;
3     int i = 0;
4     fd_set afd;
5     fd_set rfd;
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          );
15    while(1) {
16        FD_ZERO(&afd);
17        FD_SET(rfcomm_socket, &afd);
18        tv.tv_sec = 1;
19        tv.tv_usec = 0;
20
21        result = select(FD_SETSIZE, &afd, NULL, NULL, &tv);
22        if(result < 0) {
23            printf("select_doesn't_work.\n");
24            break;
25        }
26    }

```

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

```

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

```

Listing 4.3: Main Loop of BLEOffloadingFramework server

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 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.

A full iteration of the server logic is presented in [Appendix E](#).

Chapter 5

Experimental Results

TODO:

NOTE ON FIRST DRAFT: This is a stub and will be replaced.

In order to validate the offloading system, a series of performance and stress tests are the determining factor. In this chapter, the testing methodologies are described for the BLEOfloadFramework.

5.0.1 Test setup

In order to offer conclusive data, the test setup contains different types of mobile devices that can benefit from the Offloading Framework, in this case, two smart phone devices from different producers with different specifications. Both devices are using the Android Operating System, version 4.4, in order to benefit from the Bluetooth Low Energy technology.

Both devices are charged to maximum capacity, as indicated by the Android notification system and run the same applications. Example applications include simple programs that are computational intensive, such as image processing applications or route calculating algorithms, which are common algorithms among mobile devices.

The idea is to expose the device to a series of tests, conducted using UIAutomator, a testing tool for Android that emulates user behavior. The test sequence repeats a pattern of user touch inputs until the devices receives a low battery notification, after which the time it took for the device to deplete it's battery is measured as the time between the start of the UIAutomator test case and the low power event.

Several test cases are distinguishable: The case where offloading is disabled and when offloading is enabled using different methods. The results of these test cases will roughly predict the power consumption of the devices in real life scenarios and the initial data can be extrapolated in order to predict the overall gain of using the offloading system.

These tests should reveal that using the framework described in section ?? will have an impact on energy consumption, in the sense that it takes a longer time to reach the low-battery notification. Exact empirical results will be available once the API for Android applications is completed.

Chapter 6

Conclusion and Future Development

TODO:

NOTE ON FIRST DRAFT: This is a stub and will be expanded upon.

Consuming less energy has been a prime focus in mobile communities since the development of high end smart phone devices has exceeded the technological advancement of batteries.

Offloading computational intensive code from one device to another is a method through which power saving can be achieved and the BLEOffloadingFramework is one of many solutions available to handle this problem. This system offers a complete solution for Android application developers to bring new value to their programs and to receive more out of the mobile environment. The solutions presented here are scalable, designed to be easy to use and offer a new dimension when it comes to programming for the Internet Of Things.

Improvements to the system can also bring a new form of distributed performance boost, because, even though the case for a stable server and mobile client was discussed, the framework can be made to work with other mobile devices as well, permitting users to share their power between them in a seamlessly and easy way.

Appendix A

BLEOffloadingFramework General Use Case

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

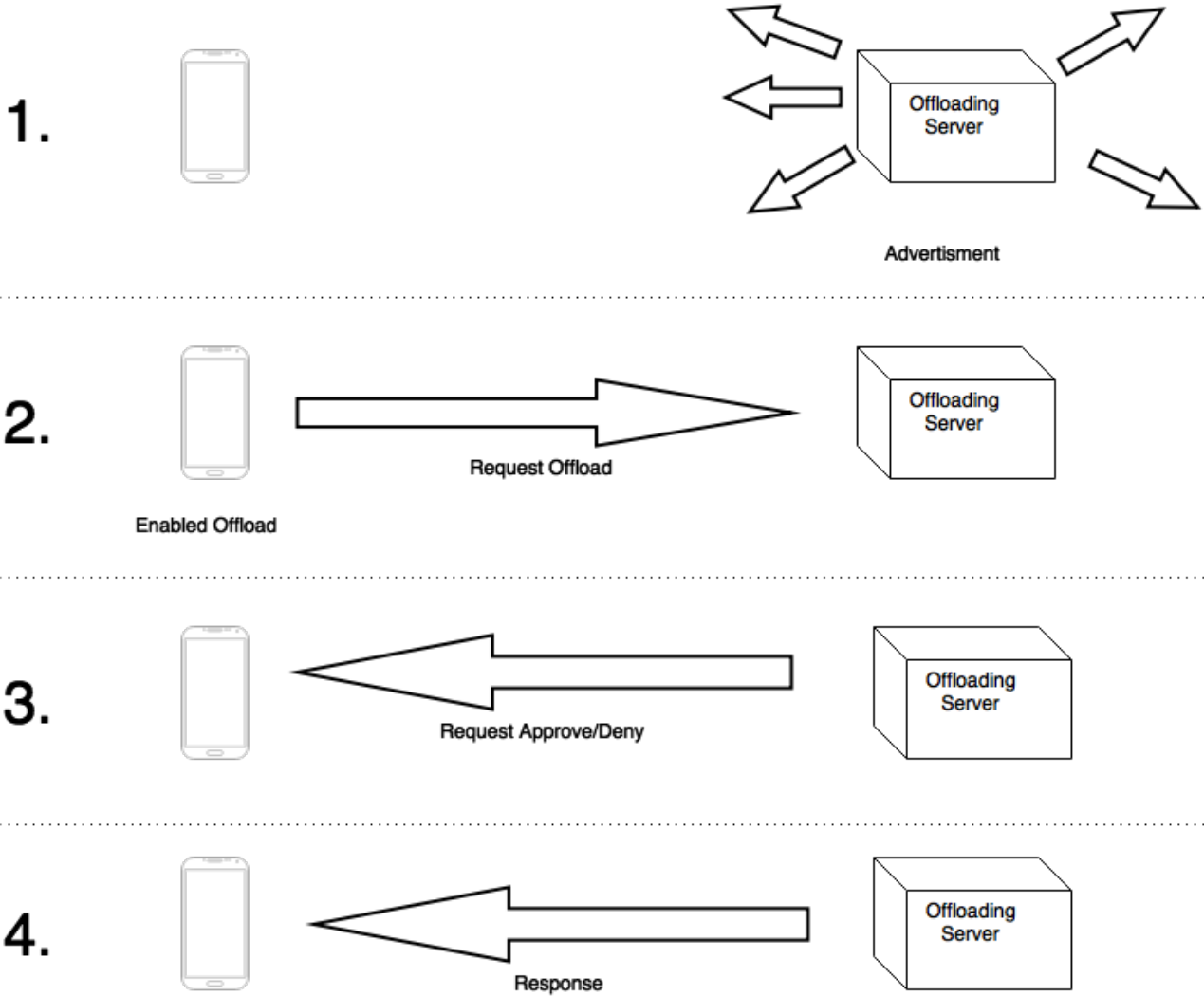


Figure A.1: Offloading Framework

Appendix B

BLEOffloadingFramework General Use Case

Appendix C

GATT Service of the BLEOffloadingFramework

```
1 struct gatt_offload_adapter {
2     struct btd_adapter      *adapter;
3     GSList                  *sdp_handles;
4 };
5 static GSList *adapters = NULL;
6
7 static void gatt_offload_adapter_free(struct gatt_offload_adapter *
8     gadapter)
9 {
10     while (gadapter->sdp_handles != NULL) {
11         uint32_t handle = GPOINTER_TO_UINT(gadapter->sdp-
12             handles->data);
13         attrib_free_sdp(gadapter->adapter, handle);
14         gadapter->sdp_handles = g_slist_remove(gadapter->
15             sdp_handles,
16             gadapter->sdp_handles->data);
17     }
18     if(gadapter->adapter != NULL)
19         btd_adapter_unref(gadapter->adapter);
20     g_free(gadapter);
21 }
22
23 static int adapter_cmp(gconstpointer a, gconstpointer b)
24 {
25     const struct gatt_offload_adapter *gatt_adapter = a;
26     const struct btd_adapter *adapter = b;
27     if (gatt_adapter->adapter == adapter) return 0;
28     return -1;
29 }
30
31 static gboolean register_offload_service(struct btd_adapter *adapter
32     )
33 {
34     bt_uuid_t uuid;
35     bt_uuid32_create(&uuid, OFFLOAD_SVC_UUID);
```

APPENDIX C. GATT SERVICE OF THE BLEOFFLOADINGFRAMEWORK27

```
32         return gatt_service_add(adapter, GATT_PRIM_SVC_UUID, &uuid);
33     }
34
35     static int gatt_offload_adapter_probe(struct btd_adapter *adapter)
36     {
37         struct gatt_offload_adapter *gadapter;
38         gadapter = g_new0(struct gatt_offload_adapter, 1);
39         gadapter->adapter = btd_adapter_ref(adapter);
40         if(!register_offload_service(adapter)){
41             printf("Offload_service_could_not_be_registered.\n")
42                 ;
43             return -EIO;
44         }
45         adapters = g_slist_append(adapters, gadapter);
46         return 0;
47     }
48
49     static void gatt_offload_adapter_remove(struct btd_adapter *adapter)
50     {
51         struct gatt_offload_adapter *gadapter;
52         GSList *l;
53         l = g_slist_find_custom(adapters, adapter, adapter_cmp);
54         if( l == NULL)
55             return;
56         gadapter = l->data;
57         adapters = g_slist_remove(adapters, gadapter);
58         gatt_offload_adapter_free(gadapter);
59     }
60
61     static struct btd_adapter_driver gatt_offload_adapter_driver = {
62         .name = "OFFLOAD_SERVICE_ADAPTER_DRIVER",
63         .probe = gatt_offload_adapter_probe,
64         .remove = gatt_example_adapter_remove
65     };
66
67     static int gatt_offload_init(void)
68     {
69         return btd_register_adapter_driver(&
70             gatt_offload_adapter_driver);
71     }
72
73     static int gatt_offload_exit(void)
74     {
75         return btd_unregister_adapter_driver(&
76             gatt_example_adapter_driver);
77     }
78
79     BLUETOOTH_PLUGIN_DEFINE(gatt_offload, VERSION,
80         BLUETOOTH_PLUGIN_PRIORITY_LOW,
81         gatt_offload_init, gatt_offload_exit
82     )
```

Listing C.1: GATT Service Implementation

Appendix D

SDP Service Registration for the BLEOffloadingFramework

```
1 sdp_session_t *register_service()
2 {
3     uint32_t service_uuid_int[] = {0, 0, 0, 0xABCD};
4     uint8_t rfcomm_channel = RFCOMM_CHANNEL;
5     const char *service_name = "Offload_Service";
6     const char *service_dsc = "And_experimental_offloading_
    service.";
7     const char *service_prov = "Offloading";
8     bt_uuid_t serviceUUID;
9     uuid_t root_uuid, l2cap_uuid, rfcomm_uuid, svc_uuid;
10    sdp_list_t *l2cap_list = 0,
11            *rfcomm_list = 0,
12            *root_list = 0,
13            *proto_list = 0,
14            *access_proto_list = 0;
15    sdp_data_t *channel = 0, *psm = 0;
16    sdp_record_t *record = sdp_record_alloc();
17    bt_string_to_uuid128(&serviceUUID, SERVICE_UUID);
18
19    // set the general service ID
20    sdp_uuid128_create( &svc_uuid, &serviceUUID.value.u128 );
21    sdp_set_service_id( record, svc_uuid );
22    // make the service record publicly browsable
23    sdp_uuid16_create(&root_uuid, PUBLIC_BROWSE_GROUP);
24    root_list = sdp_list_append(0, &root_uuid);
25    sdp_set_browse_groups( record, root_list );
26
27    // set l2cap information
28    sdp_uuid16_create(&l2cap_uuid, L2CAP_UUID);
29    l2cap_list = sdp_list_append( 0, &l2cap_uuid );
30    proto_list = sdp_list_append( 0, l2cap_list );
31
32    // set rfcomm information
33    sdp_uuid16_create(&rfcomm_uuid, RFCOMM_UUID);
34    channel = sdp_data_alloc(SDP_UINT8, &rfcomm_channel);
```

```
35     rfcomm_list = sdp_list_append( 0, &rfcomm_uuid );
36     sdp_list_append( rfcomm_list, channel );
37     sdp_list_append( proto_list, rfcomm_list );
38
39     // attach protocol information to service record
40     access_proto_list = sdp_list_append( 0, proto_list );
41     sdp_set_access_protos( record, access_proto_list );
42
43     // set the name, provider, and description
44     sdp_set_info_attr( record, service_name, service_prov,
45                       service_dsc );
46
47     int err = 0;
48     sdp_session_t *session = 0;
49
50     // connect to the local SDP server, register the service
51     record, and
52     // disconnect
53     session = sdp_connect( BDADDR_ANY, BDADDR_LOCAL,
54                           SDP_RETRY_IF_BUSY );
55     err = sdp_record_register( session, record, 0 );
56
57     // cleanup
58     sdp_data_free( channel );
59     sdp_list_free( l2cap_list, 0 );
60     sdp_list_free( rfcomm_list, 0 );
61     sdp_list_free( root_list, 0 );
62     sdp_list_free( access_proto_list, 0 );
63
64     return session;
65 }
```

Listing D.1: SDP Service Implementation

Appendix E

BLEOffloadingFramework Offload Server Logic

Bibliography

- [1] Saeid Abolfazli, Zohreh Sanaei, and Abdullah Gani. Mobile cloud computing: A review on smartphone augmentation approaches. *arXiv preprint arXiv:1205.0451*, 2012.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] Android Developers. What is android, 2011.
- [6] Troy Bryan Downing. *Java RMI: remote method invocation*. IDG Books Worldwide, Inc., 1998.
- [7] 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.
- [8] PC Gartner Lowers. Forecast as consumers diversify computing needs across devices. *Accessible from: www.gartner.com/it/page.jsp*.
- [9] Megan Geuss. Why your smartphone battery sucks, 2015.
- [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] Marcel Holtmann, Max Krasnyansky, et al. Bluez, official linux bluetooth protocol stack, 2007.
- [13] 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.
- [14] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, (4):51–56, 2010.
- [15] Elke Mackensen, Matthias Lai, and Thomas M Wendt. Performance analysis of a bluetooth low energy sensor system. In *Wireless Systems (IDAACS-SWS), 2012 IEEE 1st International Symposium on*, pages 62–66. IEEE, 2012.

- [16] 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.
- [17] Michal Ostrowski and M Math. *A mechanism for scalable event notification and delivery in Linux*. PhD thesis, University of Waterloo, 2000.
- [18] 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.
- [19] Wikipedia. Android (operating system), 2015.

Index

BLEOffloadingFramework, [2](#)

Bluetooth Low Energy, [2](#)

offloading, [1](#)