Energy Efficient Scheduling Manager for Cloud-based Mobile Applications

Chutiwan Rakjit

A thesis submitted to

Auckland University of Technology

in partial fulfillment of the requirements for the degree

of

Master of Computer and Information Sciences (MCIS)

2014

School of Computer and Mathematical Sciences

Table of Contents

List of Figures

List of Tables

Attestation of Authorship

“I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.”

Signature of Candidate: ……

Acknowledgements

I would like to express my deep and sincere gratitude to my supervisors, Dr William Liu and Dr Jairo A. Gutiérrez of the School of Computer and Mathematical Sciences, Auckland University of Technology. Their wide knowledge and logical way of thinking have been of great value for me. Their understanding, encouragement and personal guidance have provided a good basis for the present thesis.

I owe my loving thanks to my parents. Without their encouragement and understanding it would have been impossible for me to finish this work.

A sincere thank is given to Network and Security Research Group (NSRG) for their valuable suggestions and feedback to my research and conference presentations. The financial support of the contestable fund provided by School of Computer and Mathematical Sciences Research Committee is gratefully acknowledged. This generous support enables me for the trip to attend the conference to present my accepted papers. The trip to Hamilton, New Zealand where I have attended the New Zealand Computer Science Research Student Conference 2013, so as to present my paper entitled: Greener Cloud-based Scheduling Algorithm for Improving the Energy Efficiency of Mobile Devices.

Publications, presentations and talks

Publications

Published

C. Rakjit, W. Liu, J. A.Gutiérrez, “A Greener Cloud-based Scheduling Algorithm for Improving the Energy Efficiency of Mobile Devices,” New Zealand Computer Science Research Student Conference 2013 (NZCSRSC 2013), April 2013, New Zealand.

C. Rakjit, W. Liu, and J. A. Gutierrez, “EESManager: Making greener cloud apps,” in Energy Efficient and Green Networking (SSEEGN), 2013 22nd ITC Specialist Seminar on, 2013, pp. 31-36.

Presentations

1. New Zealand Computer Science Research Student Conference 2013 in Hamilton, New Zealand. Topic: “A Greener Cloud-based Scheduling Algorithm for Improving the Energy Efficiency of Mobile Devices”. April 2013.

Talks

1. Network and Security Research Group (NSRG) weekly meeting, research progress report on 3 months literature reviews. October 2012.
2. NSRG weekly meeting, seminar with Professor Daoliang Li on EESManager. July 2013.

Abstract

Mobile cloud computing is the state-of-the-art mobile distributed computing paradigm which comprises of three heterogeneous domains: mobile computing, cloud computing, and wireless networks aiming to enhance the computational capabilities of resource-constrained mobile devices towards a rich user experience. For example, heavy computations can be offloaded to the cloud to reduce energy consumption for the mobile device. However, we discovered that, in some mobile cloud application cases, it is more energy inefficient to use cloud computing than the traditional computing conducted in the local device.

In our study, we chose a navigation application, Osmand, running on an Android mobile platform to do the empirical measurement because Osmand has both cloud-based and non-cloud-based versions. So, we were able to compare non-cloud-based Osmand and cloud-based Osmand in term of energy efficiency. In the empirical measurements, we found that non-cloud-based Osmand and cloud-based Osmand consume a similar amount of energy regarding to LCD and GPS activities. For non-cloud based Osmand, the majority of CPU energy consumption was used on calculating route results. In addition, we found that the complexity of maps affects the CPU energy consumption. On the other hand, the cloud-based Osmand consumes more CPU energy than non-cloud-based Osmand because the majority of CPU energy consumption was used for creating events and displaying on the mobile screen. Moreover, cloud-based Osmand needs a network connection to send a request to the cloud to calculate a route result. Thus, 3G communications is considered as an extra factor that causes energy consumption in cloud-based Osmand. We found that 3G communications makes cloud-based Osmand energy inefficient due to tail energy in 3G communication costing extra energy consumption.

Therefore, a prototype of an Energy Efficient Scheduling Manager (EESManager) was proposed and developed based on the awareness of tail energy and the complexity of maps. The results from the evaluation show that EESManager can improve the energy efficiency of cloud-based Osmand in two cases: 1) if the mobile device receives route results back from the cloud within the tail time, and 2) in the case when the mobile device

does not receive the route results from the cloud within the tail time and the map scenario is simple.

Chapter 1 Introduction

In recent years, smart phone technologies have become powerful and smart phones are now like small computers for mobile users. Due to more powerful computation, better processing and sharper mobile screens in smart phones, there are various mobile applications in the market which use the advantages of smart phone technologies to make applications more interesting. However, there are many mobile applications such as games, navigation, voice recognition, and image retrieval which can cause batteries to drain quickly because of the heavy computation required by these multi-media applications. Therefore, battery drain has become a big issue in smart phones. A study [1] showed that mobile users in 15 countries worry the most about short battery life of their smart phones. Another study [2] also had a similar result with 38% of 215 iPhone users having short battery life as their biggest concern.

Industry and academics have widely recognised cloud computing as the next generation of computing infrastructure. Cloud computing can provide scalable and powerful computing power, processing memory and storage. Meanwhile, wireless broadband networks deploy rapidly, and this allows many mobile users to experience Internet services from their smart phone. Therefore, integrating cloud computing into the mobile environment is a promising and innovative idea to bring new varieties of services and facilities to mobile users. Mobile users can access the cloud via the existing Internet-based cloud and wireless technologies. Tasks and raw data from smart phones can be sent to the cloud for processing and implementation. Offloading heavy computation to the cloud can reduce energy consumption in smart phones, therefore resolving the battery life issue [3].

However, mobile cloud computing faces battery life challenges as well as shown in a measurement study [4] which found that three types of cloud-based applications actually consume more energy than non-cloud-based applications or device-based applications due to energy inefficient network communications in the cloud-based applications. Hence, a conclusion could be drawn that cloud-based applications are more energy inefficient than

non-cloud-based applications. However, another study [5] concluded that each mobile application is designed differently. Thus, each application has different energy consumption characteristics. It is really a case-by-case condition.

Therefore, this thesis emphasises on the energy efficiency of the chosen mobile application. Empirical energy measurements and energy models were proposed. Then our Energy Efficient Scheduling Manager (EESManager) was developed based on energy efficiency recommendations and the proposed changes were evaluated using a simulation environment.

1.1 Background

Mobile cloud computing is the combination of the mobile environment and cloud computing which draws benefits from both technologies. Mobile cloud computing moves the data processing and storage from smart phone devices to powerful computing machines in the cloud to overcome smart phone limitations. Mobile cloud computing provides mobile users with communication and entertainment that can be accessed anywhere and anytime with existing wireless technologies. Mobile cloud computing uses the same main concept as cloud computing; however, mobile devices have replaced PCs as the preferred means of accessing the internet. [6-8]

Despite mobile cloud computing being a new promising idea, it still faces several problems for mobile devices such as short battery life, storage, security and so on. One of the most critical concerns for mobile devices is energy efficiency. Smart phone capabilities are becoming increasingly powerful and fast. Unfortunately, battery technology has developed slowly and it is not able to match the energy consumption requirements of hardware components [9-11]. A battery has limits in terms of the amount of energy that it is capable to store and the battery technology is only increasing capacity 5% annually [12]. Therefore, solving the energy inefficiency in mobile devices has become a challenge of the mobile industry.

1.2 Motivation

The goal of this thesis is to improve the energy efficiency of a selected mobile application and a smart phone device. So, three research questions are posed to carry out this thesis.

1. What are the characteristics of a cloud-based application on the battery draining of smart phone devices compared to a non-cloud-based application by using existing technologies and tools?
2. What are the features, characteristics or scenarios of a cloud-based prototype built using energy efficient recommendations?
3. What are the results of comparing the energy use of the device-based application vs. the cloud-based application using a simulation environment?

To answer those research questions, we firstly identified characteristics of the selected mobile application Osmand that cause energy to be used by using existing technologies and tools which will be discussed in Chapter 3. Then, a prototype was developed based on energy efficiency recommendations discussed in Chapter 4. The results of comparing energy consumption using a simulation environment were presented in Chapter 5.

1.3 Thesis structure

The thesis's structure is depicted in Figure 1.1 below. The remainder of this thesis contains an introduction of relevant background knowledge, followed by the empirical energy measurement of Osmand and an analysis of energy models based on the empirical energy measurements and studies surveyed [13-20]. The simulation studies are then presented to evaluate our novel prototype. The chapters of the thesis are organised as follows:

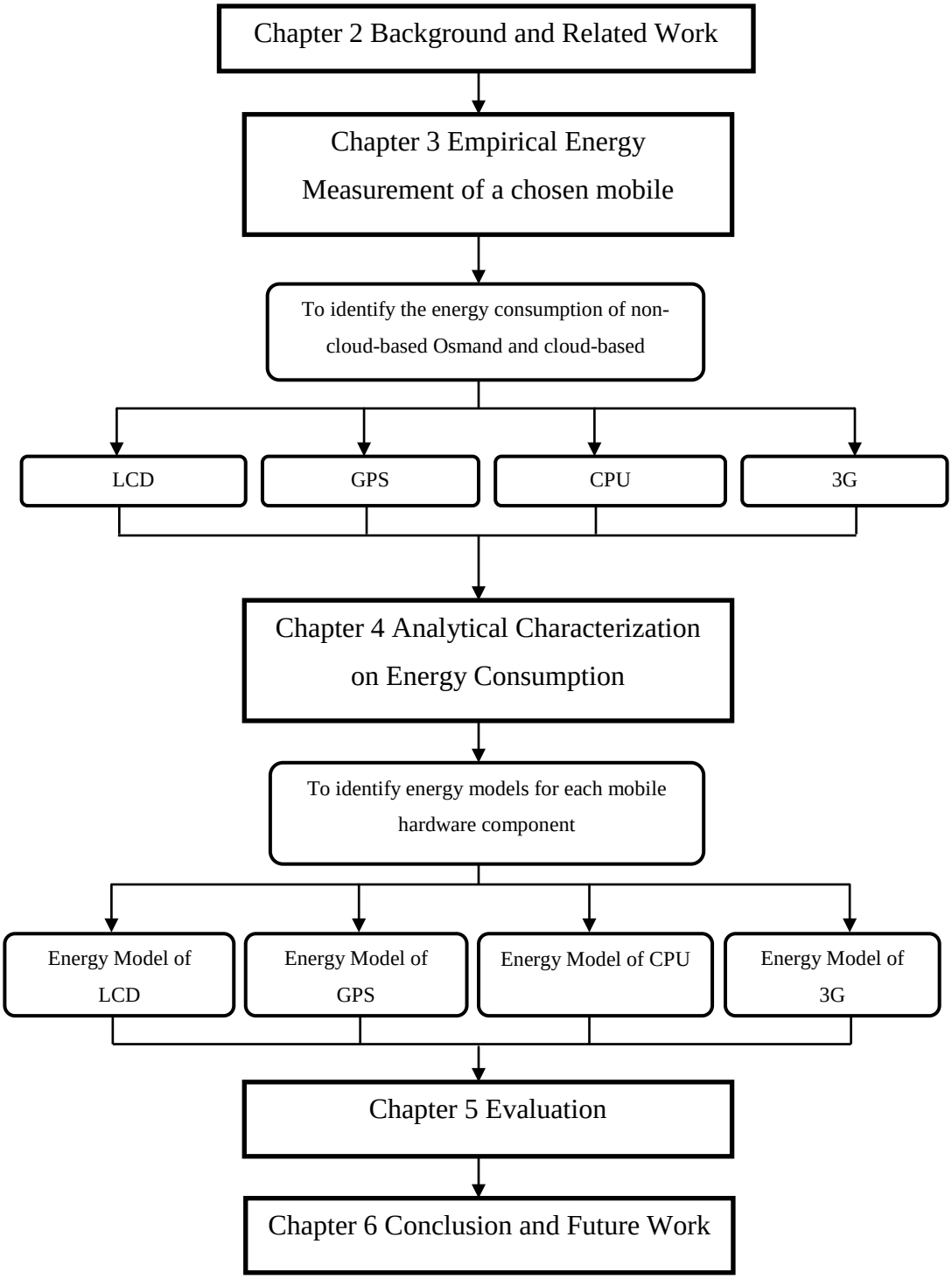


Figure 1.1 - Thesis structure

Chapter 2 gives a detailed background introduction of mobile cloud computing. The concept model of mobile cloud computing and its issues are introduced based on the existing literature. Furthermore, an overview of the Android platform that is used and

studied in this thesis is presented. Moreover, previous work related to mobile cloud computing is presented.

Chapter 3 presents the methodology of the thesis and the empirical energy measurements of the Osmand application in an actual smart phone device. Four hardware components (LCD, GPS, CPU and 3G) of the smart phone device involved in Osmand have been presented in terms of energy consumption. Each empirical energy measurement shows the comparison of non-cloud-based and cloud-based versions of the selected mobile application.

Chapter 4 describes the energy consumption of each hardware component in an energy model that is based on the empirical measurements and knowledge from several studies [13-20]. Also, the energy efficiency recommendations based on previous knowledge and the EESManager are presented.

Chapter 5 presents the simulation studies of EESManager in different scenarios. The comparison of non-cloud-based Osmand, original cloud-based Osmand and cloud-based Osmand with EESManager are used in the simulation studies to evaluate EESManager. A discussion on the performance of EESManager based on the simulation results is presented.

Finally, the findings and results are concluded in Chapter 6. In addition, we suggest some possible research directions to advance our prototypes in future work.

Chapter 2 Background and Related work

In this chapter, we give an overview of mobile cloud computing including its concept and architecture. The issues of mobile cloud computing are then introduced and bring out the research problems. Furthermore, an overview of Android platform which was used in the empirical measurement in Chapter 3 and previous related work are presented and discussed in details.

The remainder of this chapter is organised into three categories: (1) background on mobile cloud computing, (2) background on Android, and (3) previous research. In section 2.1, we give an overview of mobile cloud computing, its concept model and its architecture. In addition, this section addresses the issues and concern under mobile cloud environment. Section 2.2 describes Android and its architecture. Moreover, mobile applications and mobile cloud applications are introduced. Section 2.3 reviews the previous work which relates to mobile cloud computing. In section 2.3, it will be divided in to 2 sub-heading; 1) energy efficient approaches and 2) Measurement study of mobile cloud computing.

2.1 Mobile cloud computing

In this section, the concept model for mobile cloud computing is presented to analyse mobile cloud computing technology and then several architecture models are provided to organise mobile cloud computing systems. Afterwards, the partition of applications and offloading is explained. Finally, issues regarding mobile cloud computing will be discussed.

2.1.1 Concept model of mobile cloud computing

We can divide cloud computing services into three types including Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). However, mobile cloud computing is not divided into these three known layers of cloud computing services. The main key of mobile cloud computing is to connect the client to the cloud [21].

Moreover, Christensen [22] provided three archetypes of components for the next generation of mobile applications including context enablement, REST-based cloud computing and the combination of smart mobile devices. These three components form a mobile cloud transmission model. The study [21] rebuilt a concept model on vertical view as can be seen from Figure 2.1.

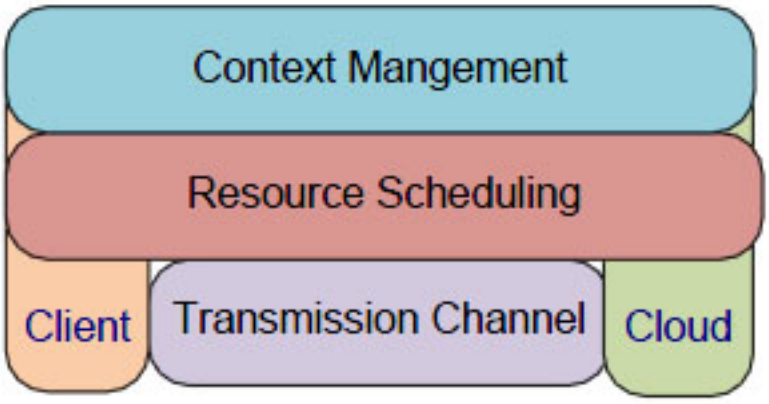


Figure 2.1 - Mobile cloud computing concept model. Reprinted from [21]

The client and the cloud are represented as the left and right entities. There are three components between the client and the cloud: the transmission channel, resource scheduling and context management. Resource scheduling and context management include components of both the client and the cloud. The precondition of this model is context-awareness on the client side and it delivers elastic, on-demand services for clients on the cloud side. Various wireless transport protocols can be referred to as the transmission channel. Resource scheduling components are used to address resources such as storage resources, computing resources and so on, which are referred to as the schedule resources. For context management, mobile device features are enabled by context to let users ascertain additional information from their mobile devices with no need for explicit user input. Context parameters can be tracked by context management and adapted to modification of context conditions.

2.1.2 Architecture of mobile cloud computing

The organisation systems of mobile cloud computing is referred to as mobile cloud computing architecture. Guan, Ke, Song and Song [21] discussed architecture schemes containing agent-client schemes and collaborating schemes.

2.1.2.1 Agent-client scheme

To help mobile devices overcome restrictions, such as processing power and data storage limitations, almost all resources are provided by the cloud side that create an agent for each mobile device. The agent is used to communicate between mobile devices and the cloud.

2.1.2.2 Collaborating scheme

In this scheme, mobile devices are considered as part of the cloud. The cloud server functions may control and schedule collaboration among mobile devices.

2.1.3 Partition of application and offloading

In order to achieve the migration and offloading of applications, each application should be divided into components and developers should consider resource consumption and data dependency of application partition.

In the study [23], a two-step approach was presented to partition or divide applications between a server and a mobile phone. First, an application's behaviour is represented as a data flow graph. In the second step, this graph finds a partitioning algorithm to provide an objective function. There are two types of partitioning algorithms that are given. The first type is ALL, which is used for the computed offline partitioning to consider different types of mobile phones and network conditions, while the second type, K-step, is used for on-the-fly partitioning. A mobile device connects to the server and the server provides resources and satisfies requirements issued by the mobile phone.

In another study [24], the author presented a two-step Hybrid Application Partitioning Strategy. The first step is to partition the workload by using the by-scale strategy. The by-scale strategy is used to break down the workload at a certain scale level into cluster nodes. The second step is to partition the cluster nodes, which is broken down

at a large scale level from the first step, by using the by-region strategy. The by-region strategy is used to scan the cluster nodes with the surround nodes.

Offloading is used to increase the energy efficiency of mobile devices [3] and involves sending the processing of mobile devices to powerful machines to perform the computation. Making offloading of computation more attractive depends on the amount of data transmitted, the amount of performing computations, and the wireless bandwidth available. Mobile image processing, for example, which sends images over wireless networks to the cloud, uses a significant amount of energy. The study [3] was used in a framework of Heterogeneous Network (HetNet) [25] to make the offload decision. The offload decision is made taking under consideration the offloading gain and the cost of energy.

In a study conducted by Eduardo Cuervo and colleagues [26], they presented the MAUI system, which increases energy efficiency by using fine-grained code offload. At the same time, the MAUI system reduces energy waste. The authors proposed three steps. The first step is to use code portability from the MAUI system to make two smart phone application versions. One of them computes on the mobile device and another version computes on the cloud. The second step is to use the combination of programming reflection and type safety to find and automatically extract the program state needed. The final step is to use the results from step two to determine the cost of network shipping.

2.1.4 Issues

There are issues with the mobile cloud computing environment today based on users and mobile application developers' opinions as Hung et al. [27] and Kumar and Lu [3] report.

2.1.4.1 Energy efficiency of mobile cloud based applications

Mobile devices are expected to handle many different sizes of data such as videos, speech, images and so on with a limitation of bandwidth wireless connection. If a user keeps using more extensive and continuous data transferring, the problem of energy efficiency arises because wireless network communication consumes the large amount of energy [28]. Therefore, offloading cannot always save energy [29].

A simple model was given by the study [3] to provide energy analysis of computation offloading. Suppose a task of computation requires *C* instructions. *M* is defined as the speed in instructions per second for a mobile device while S is defined as the speed in instructions per second for the cloud server. *D* is defined as bytes of data exchange between the cloud server and the mobile device and *B* is defined as the transmission rate of the mobile device. In the mobile device, mobile computation consumes P_c watts while a mobile being idle consumes P_i watts and a mobile sending and receiving data consumes P_{tr} watts. The mobile device consumes $P_c \times (\frac{C}{M})$ watts for performing the computation while the server consumes $\left[P_i \times \left(\frac{c}{s}\right)\right] + \left[P_{tr} \times \left(\frac{D}{B}\right)\right]$watts for performing the computation. So the amount of energy saved is

$$P_c \times \left(\frac{C}{M}\right) - P_i \times \left(\frac{c}{s}\right) - P_{tr} \times \left(\frac{D}{B}\right)$$

Therefore, the mobile device can benefit from offloading when a large amount of computation *C* and a small amount of communication *D* are present. This model agrees with the one in study [4] in which cloud computing was shown to be energy efficient when the mobile device can offset the communications energy consumption. As a result, a challenge arises when a developer needs to determine an offloading decision in order to make a cloud-based application more energy efficient.

2.1.4.2 The re-design and deployment of applications

Traditional client-server applications have many advantages in the cloud. However, the requirement of application partitioning and deployment of services is to enable fine-grain, dynamic client-server collaboration [27]. Moreover, the performance of web applications or cloud applications appears to be slower than stand-alone applications due to real-time data on an application. For example, a chess game application, which is based on mobile cloud computing and needs to send and receive data from the cloud, causes slower performance [3]. Additionally, client-server applications reduce effects on the deployment of personal applications. Also, the server is owned by someone else and needs to be maintained [27].

2.1.4.3 The condition of network and service availability

A model of Quality of Service (QoS) is designed basically for high bandwidth and real-time traffic such as video conference [28]. However, the performance of mobile cloud applications depends on the cloud service and wireless network [3, 30, 31]. The quality of the network between mobile devices and the cloud depends on latency, bandwidth, etc. It is hard to ensure the smoothness of a mobile application especially in poor network conditions [28]. For example, in a national park users may not be able to use mobile cloud applications for organisation, retrieval or identification of data. It is also difficult to make mobile cloud computing available in subways, tunnels and building basements [3, 30, 31]. The reliability of the cloud depends on a low number of service outages and high amounts of data storage. As an example of a failure of this technology, the mobile Sidekick service of T-Mobile and Microsoft crashed and lost customers' data and contacts [3].

Therefore, the most challenge for wireless network of mobile cloud computing is the wireless connectivity which can meet mobile cloud computing requirements with respect for scalability, availability and energy efficiency [32].

2.2 Android

Android is a mobile operating system designed for mobile devices such as smart phones, tablets and netbooks. The concept and platform was created by Android Inc., and Android was bought by Google in 2005. The original goal of Android was to create an operating system that was small, easily upgraded and flexible for handsets used in businesses or industry. The intended use of Android 1.x and 2.x is for smart phone devices whereas Android 3.x is designed for high-end support of tablet computers [33].

2.2.1 Architecture and background of Android

In this section, the Android platform architecture will be discussed to provide an understanding of its key concepts. The Android platform stack is divided into three layers as you can see in Figure 2.2. Each layer in an Android architecture contains several program components and provides different services in each layer [34].

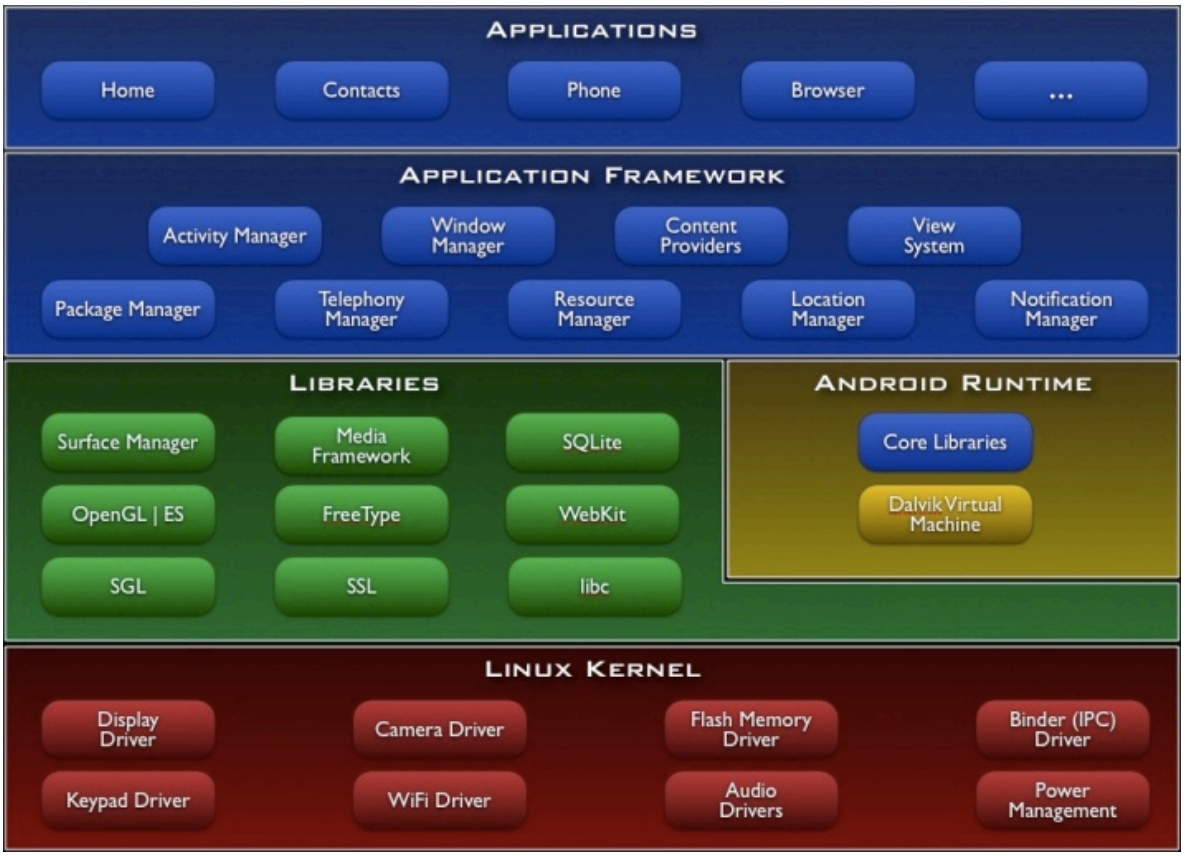


Figure 2.2 - The system architecture of the Android platform. Reprinted from [35]

2.2.1.1 Linux Kernel

The Linux Kernel is a base of the whole Android OS. The Linux Kernel is designed to stores all the essential mobile hardware drivers. The mobile hardware drivers, which are programs, are controlled by the Linux Kernel to communicate with the mobile hardware. Moreover, the Linux Kernel is used to take care of the operation of the core system, which includes power and memory management, networking, and process management. The Linux Kernel can be referred as a layer of abstraction between other software layers and the hardware. [34-36]

2.2.1.2 Libraries

Libraries in this section are referred to as C/C++ runtime libraries. They run directly on the Linux Kernel and make core services (such as display management, video and audio media playback, graphics support and so on) available for the Android runtime and applications. [34-36]

2.2.1.3 The Android runtime

The Dalvik Virtual Machine (DVM), which is a type of Java Virtual Machine, is the main component of the Android runtime and there is a core set of Java libraries and APIs supporting the DVM. This means there are detailed documents and open source files for application developers to use. Each DVM is located in the process of the Linux Kernel, which has priority in the threading of system-level and memory management. The DVM can support multi-threading which means that Android applications can be run on its own process. [34-36]

2.2.1.4 Application framework

The application development is directly supported by the Android application framework. It uses the set of Java namespaces and classes to develop an application. The application framework provides application developers with a wide Android Manager Provider range including namespaces and classes for creating system notifications; managing the telephone, camera, acceleration detection, the user interface, etc. [34-36]

2.2.1.5 Applications

The top layer in the Android architecture is applications where are installed in an Android mobile. The system treats all Android applications and standard Android applications (such as SMS application, dialler, web browser and contact manager) and libraries of native code, which are able to be called and loaded by the Java Native Interface to develop applications. [34-36]

2.2.2 Mobile applications

When an application is developed, a developer writes a program in Java. Once an Android application is installed, the Android application is located in its security sandbox. So, each application can only access the resources that it requires [37].

There are four kinds of application components including activities, services, content providers and broadcast receivers. In the activity component, an e-mail application, for example, displays an e-mail being composed on a mobile screen while the other activities may show a list of new e-mails or reading an e-mail. In the service component, it

is used to perform the long running operation or work for remote processes. This component runs in the background. For example, a user may run a radio application while he or she uses another application. In the content provider component, a shared set of application data is managed by the content provider [37]. In the broadcast component, a reactive object is launched to handle a specific task [38].

There are several categories of mobile applications. Applications for general information provide mobile users information in a particular area. Applications using personal data to login require personal information to access the server. For example, a bank account application would require the customer username and password. The encryption of this information is required on a channel of communication between the mobile device and the bank's database. Network communication applications provide communication between a user and other users. Commercial applications let users purchase goods, make payments or engage in other commercial activities. Games entertainment applications that users choose to play in their free time [39].

2.2.3 Mobile cloud applications

There are two types of cloud-based applications including web applications(or browser-based applications) and native applications [40].

Browser-based applications involve the access of mobile applications by using a native browser. They use standard web technologies such as CSS, HTML and JavaScript to execute on the server. The advantages of browser-based applications are that there is no need to maintain the server, there is flexibility to run on any platform, there is no need to process application approvals and they use the same architecture as traditional web applications. However, there are some disadvantages of browser-based applications. For example, the network latency can cause poor user experience. Also the graphical interface may be of poor quality based on the application or mobile device. Moreover, users do not use the applications without a network connection. On the other hand, native applications use Java and Android APIs to develop native applications to provide applications that are the most flexible and provide the best user experience. Moreover, low-level hardware can be accessed.

2.2.3.1 Programming concepts

There are two sides of mobile cloud applications including cloud servers and mobile application frameworks. A cloud server is the server-side infrastructure component that is in the cloud. Mobile-oriented features (such as real-time push, data synchronisation and so on) are provided by the system. There are two programming concepts including the channel server and Mobile Service Bean. The channel server is a gateway used to integrate objects of on-device models and data with the systems of server-side backend storage. With the channel server, developers do not have to worry about low-level state management, synchronisation or mobile-oriented issues. The Mobile Service Bean provides a request/response service, which is based on a synchronous invocation mechanism [40].

In a mobile application framework there are many components designed to support applications in terms of mobile data, mobile model-view-controller (MVC) and the mobile cloud. For example, data-oriented services such as simple Remote Procedure Calls (RPCs), real-time push notifications and data synchronisation are provided by the mobile data framework [40].

2.3 Previous research

To easy to understand, we divide previous research in this section into 2 sub-sections which are 1) energy efficient approaches and 2) measurement studies of mobile cloud computing.

2.3.1 Energy Efficient Approaches

Pathak, Hu and Zhang [41] created an energy profiler for applications on smartphones called “eprof”. Eprof’s architecture has three components: code instrumentation and logging, power modelling and energy accounting, and profile presentation. Firstly, eprof collects processes, threads, subroutines and application system calls. Secondly, it tracks, logs and draws back energy activities to smartphone hardware components. Thirdly, eprof draws the energy activities matching with the entities responsible for the activities. Eprof was used in case studies to understand the energy use by mobile applications. A result of using eprof shows that free applications use 65%-75% of energy consumption for third-party advertisement.

Addressing the challenges of energy efficiency of communication on smart phones, Sharma, Navda, Ramjee, Padmanabhan and Belding [42] developed Cool-Tether to provide energy efficiency on WAN links, which are based on GRPS/EDGE/3G and WiFi radios. Firstly, they identified the properties in detail and the specific architectural elements of the two main energy components, which are associated with WAN and WiFi links, to be addressed in the system design. The Cool-Tether system contains three key components including a cloud-based gatherer, an energy-aware striper and a reverse-infrastructure mode WiFi LAN. Finally, they evaluated the Cool-Tether system by comparing the system with the COMBINE system [43]. The result showed that Cool-Tether can save over 50% of energy while on Wi-Fi mode.

Chun, Ihm and Maniat is [44] presented the design and implementation of the CloneCloud system, which chooses parts of an unmodified application that can benefit from remote computation to transfer the chosen parts of the unmodified application to the cloud. There are two key parts in the design of CloneCloud including partitioning and distributed execution. The aim of the partitioning mechanism in CloneCloud is to choose which parts of an application's computation to migrate to the cloud and which parts to run locally. To produce a partition, the CloneCloud partitioning framework combines static program analysis with dynamic program profiling. CloneCloud uses static analysis to identify legal choices for migrating and to locate re-integration points in the code. At the code level, CloneCloud transfers only the point at the boundaries of application methods and at the virtual machine layer method's boundaries. The core-system library method and the native method boundaries cannot be transferred. For example, a method uses a local resource such as a GPS, a microphone or a camera in the mobile device. This method has to run locally. Moreover, a method may create and access lower states than a VM and the method may share the native state. Therefore, the method has to be collocated with the same machine because CloneCloud does not send native states. Moreover, the dynamic profiler collects the data that will be used to make a cost model. In this study, execution time and energy consumed at the mobile device are used for the cost metric. For the execution time cost metric, CloneCloud collects the execution time of methods and computes migration costs by simulating migrations at each method. For energy consumption, the Monsoon power monitor was used to measure energy providing a model

to estimate energy cost. Then, the optimiser aims to choose which method of an application to transfer to the clone from a mobile device. Two analysers are used to inform the optimisation regarding the methods to be transferred. CloneCloud uses a distributed execution mechanism to carry out a specific partition of processes of an application that runs inside an application layer virtual machine. The distributed execution uses the thread migrator to stop a migrant thread from collecting the migrant thread's state. Then the thread migrator sends the state to the node manager for data transfer. A phone will mark states that are migrated to the clone for resuming and merging later. Hence, CloneCloud can increase the execution speed and reduce energy consumption.

Cuervo, Balasubramanian and Cho [26] use the advantages of two approaches in their architecture to reduce energy consumption. The first approach addresses how to partition a program to be sent to the cloud. Therefore, this approach can reduce power consumption. The second approach is to use a virtual machine migration to migrate a program to the cloud. Thus, programmers do not to need to change a program to obtain advantages from remote execution. MAUI is a system that is placed between applications and the server. The MAUI system requires an application to have two versions of a program: one for running locally and the other to run on the server. Before the application starts computation, MAUI profiles each version of the application and calculates the energy cost of each method including CPU and memory cost. The energy cost of transmission over the network is also measured including its bandwidth and delay. Information from the previous step helps MAUI decide which methods run locally and which methods run on the cloud. When the application starts processing, MAUI chooses the best way for each method to operate to maximise energy saved. While another approach [45] does not estimate the computation time and the energy consumption in an application like MAUI, it lets the application run locally within a timeout. If the computation of the application is not completed within the timeout period, the computation will be offloaded to the cloud. Given the propensity of online mobile games to request constant connectivity and to consume high levels of energy, the battery power has become one of the main concerns with smart phones.

Bhojan and Akhihebbal [46] introduced ARIVU to conserve the power consumed by the network connection, display and processor of smart phones. ARIVU is a middleware

that reduces client's energy consumption for multiplayer online role-playing and first-person shooting games without decreasing the quality of game play. ARIVU contains two components including the Resource Data Collector (RDC) and the Resource Controller (RC). The RDC collects necessary raw data from the client side and samples game states. Then the RC uses raw data from the RDC to analyse a client's game state.

2.3.2 Measurement studies of mobile cloud computing

A measurement study [47] found mobile networking technologies such as 3G and WiFi used a large amount of energy because of tail energy. Tail energy is energy spent to keep a network interface in the same power state after a data transfer is complete. However, if a data transfer occurs within the tail time, there is no energy spent. Therefore, TailEnder was developed based on the study [47] to reduce energy consumption while still meeting user expectations. In e-mail applications, for example, TailEnder schedules a data transfer to minimise tail energy.

Namboodiri and Ghose [4] tried to answer the question of "when is the usage of non-cloud-based applications less preferable in terms of energy consumption than cloud-based applications?" Three types of applications including word processing, multimedia and gaming were used in this study. The local word processing application required no communications and used local resources for running the application. On the other hand, the cloud-based word processing application needed some network connectivity and used little local computation. Multimedia applications had two options. The first option was that the multimedia application played files remotely from servers over the network. The second option was to let the multimedia application download files and store them so they could be played when the multimedia application is offline. Finally with game applications, the amount of processing required depends on the nature of the game. In this case online games involved network communications to support their interactivity requirements. The HTC Desire phones were used in this study to run sample applications and the PowerTutor [14] tool was used to measure energy consumption. Both WiFi and 3G access were turned off for all trials involving device-based (local) applications. The QuickOffice application was used as the local word processing application and Google Docs was used as the cloud-based word processing application. Local video files were played to test a local multimedia

application and YouTube ran as a sample of a cloud-based multimedia application. For gaming, they used the same game application but for non-cloud-based applications they played chess with the computer locally and remotely. The results show that cloud-based applications consumed more energy than non-cloud-based applications. Especially, the cloud-based multimedia applications were the most energy inefficient because playing an online video required significant local processing and communication to send a stream of data. They reported that the WiFi interface's power consumption was significant for cloud-based applications while non-cloud-based applications turned the WiFi interface off. Then, the authors analysed the energy consumption of a device used in their earlier study, which provides the factors which relate to the energy consumption of cloud-based applications and non-cloud-based applications. They discovered that cloud-based application scans were more energy efficient than non-cloud-based applications if the decreased local computation can offset the energy consumption of network communications. Furthermore, they found that the energy consumption of CPU execution has to be bigger than the energy consumption of network connectivity to be energy efficient. Next, the authors introduced GreenSpot, which is an algorithm, designed to help mobile devices to choose cloud or non-cloud versions to run an application. First, GreenSpot checks the available versions of the application. If only one version is available, GreenSpot uses that version. If both versions are available, GreenSpot evaluates the two options to select the most energy efficient one for the application.

Lagerspetz and Tarkoma [48] analysed and measured the trade-off between mobiles and the cloud for mobile search and from the point of view of energy use. They developed a mobile search and synchronisation application based on the Dessy project [49]. A collection of files is read or indexed by the mobile search and synchronisation application. This method allows the application to search for key words in files. Cloud technologies make indexing faster and achieve more accurate search results. Also, the results showed that cloud technologies improved the energy use in mobile devices. In Kumar and Lu's paper [3], they explained offloading, which is used to increase the amount of energy saved in mobile devices. The concept of offloading involves sending the processing of mobile devices to a powerful machine to perform the computation. This can result in a high level of energy efficiency. They examined energy consumption from offloading by carrying out

a simple analysis. A chess game and an image processing application were used as examples to test offloading. They used a formula that they developed to measure energy consumption. As a result the chess game serves as an example of the advantages of offloading when the amount of computation is very large while the amount of bytes exchanged between the server and the mobile system is small. Therefore, the attractiveness of offloading of computation depends on the amount of data being transmitted and the amount of computation required; the wireless links and transmission use most of the energy. Mobile image processing, for example, which sends images over a wireless network to the cloud, uses large amounts of energy.

Chapter 3 Empirical Measurement

In previous chapter, we reviewed the partition of application and offloading which involved energy efficiency on mobile devices. In this chapter, we have proposed results of the empirical measurement of energy consumption on mobile hardware components of our chosen mobile application. Due to the empirical measurement, the results of energy consumption in each mobile hardware component contain collected data from the non-cloud-based version and the cloud-based version of our chosen application. The first two results which are energy consumptions of LCD and GPS have the similar trend of consuming energy in both versions. However, the last two results of energy consumption which are CPU and 3G communication show different trend in the two versions. CPU and 3G communication of the cloud-based version have the larger amount of energy consumption compared with the non-cloud-based version.

The remainder of this chapter is organised as follows. Firstly we will explain a chosen Android application, which is Osmand, for this empirical measurement under consideration and its characteristics. Osmand in general, non-cloud-based Osmand and cloud-based Osmand will be proposed to understand nature of Osmand and how it works. Next, the methodology of the empirical measurement will be described. The existing technologies which were used in the empirical measurement will be also defined. Finally results of the measurement study on a smart phone running an Android platform will be shown.

3.1 Osmand: Map & Navigation

In this empirical measurement, one particular navigation application was chosen to study in-depth because there is no navigation application has been studied before and useful information from in-depth study can explain how the navigation application consumes energy. Osmand (Open Street Map Automated Navigation Directions) running on an Android mobile platform [50] was chosen as our testing application (from the view point of energy efficiency) because Osmand is a popular open source navigation project on Google

code for Android and BlackBerry developers [51]. Osmand is a step-by-step navigation application that has been downloaded by more than 6,000 people worldwide on their smart phones [52, 53]. Also Open Street Map(OSM) map databases, which is created and provided free geographic data to everybody [54, 55] are used to display maps in Osmand, and they can be stored on a local device's memory card. Moreover, Osmand provides offline and online modes for finding route results. Therefore, we can compare the energy efficiency of non-cloud-based Osmand and cloud-based Osmand as the main functions in Osmand can work in both offline and online modes. To use Osmand, a user clicks on the Osmand icon on a mobile screen. Then the Android system shows Osmand on the screen as you can see in Figure 3.1. There are four buttons: Map, Search, Favorites and Settings. The user can click on the Search button to start looking for a route to a particular destination or choose the Map button to show the user's current position on the map.

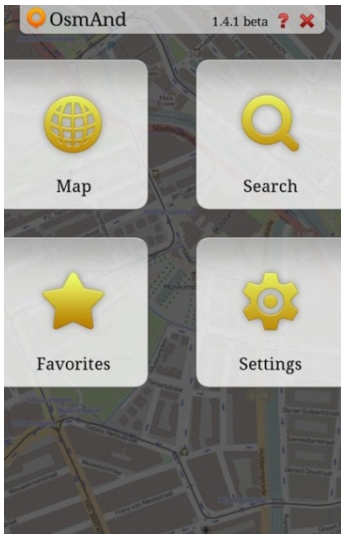


Figure 3.1 - Osmand's main menu

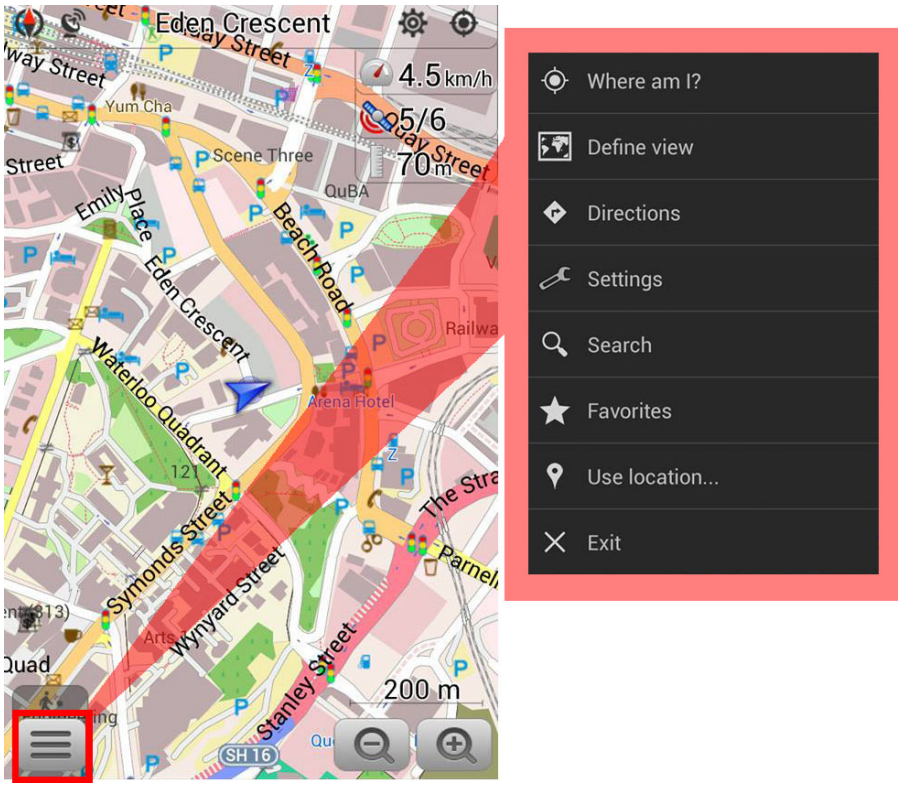


Figure 3.2 - Osmand's sub-menu

Next the user can choose a search button in a sub-menu to choose the destination address as you can see from Figure 3.2. After the user selects a destination using the Search button on the main menu or the sub-menu, Osmand starts searching for a route from the user's current position, which is referred to in this study as the starting point of the trip. The method used to find a route result differs depending on navigation services. When the result is found, it is overlaid on the map.

Osmand has offline and online modes for its navigation services. We call offline mode in Osmand as non-cloud-based Osmand while online mode is called cloud-based Osmand. Here below are the explanations of non-cloud-based and cloud-based Osmands.

3.1.1 Non-cloud-based Osmand

A user can use offline mode of Osmand by changing Osmand's navigation service at Setting Menu. Offline mode or we call it as non-cloud-based Osmand uses the 'OsmAnd' utility as its navigation service. This utility uses the A* algorithm approach and the A* algorithm calculates the moving cost between the starting point and the destination point

[56, 57]. First, the algorithm calculates the moving cost (by based on the destination point) between the starting point and nodes which surround the starting point. Then the algorithm chooses the lowest moving cost from the starting point to one of the surrounding nodes (we refer it as node A). Then the algorithm, again, calculates the moving cost between node A and nodes which surround node A. The algorithm repeats this process until reaching the destination point. More detail about how A* algorithm work will be in section 4.3. When a result is found, Osmand overlay the result on the map.

3.1.2 Cloud-based Osmand

A user also can change to use online mode by changing Osmand's navigation service at Setting Menu. Online mode or we call it as cloud-based Osmand uses CloudMade as its navigation service. Cloud-based Osmand works by sending a request file to search a route that contains the coordinates of the particular starting point and destination point to CloudMade on this url [58]. Then, CloudMade processes the result and sends it back to the local device. Osmand in the local device parses the result and shows on the mobile screen.

3.2 Methodology

Experimental methodology was considered to take part in the empirical measurement (in Chapter 3) as its methodology because the concept of experimental methodology is to find relationship between factors in a subject [59, 60]. Strong evidence is provided by using the experimental methodology for causal interpretations [61]. Thus, doing experimental methodology can provide answers of Research Question 1 (What are the characteristics of a cloud-based application on the battery draining of smart phone devices compared to a non-cloud-based application by using existing technologies and tools?) with proof of results from experiment that are able to explain the characteristics of a cloud-based application in a smart phone device which causes energy used. Moreover, we considered simulation methodology to be used in the evaluation in Chapter 5 to answer Research Question 3 (What are the results of comparing the energy use of the device-based application vs. the cloud-based application using a simulation environment?) because hypotheses which are a cloud-based prototype build using energy efficient recommendations in this thesis can be tested by simulation to save the cost and time [62]. Simulation methodology is the use of a

mathematical/logical model to test the performance of a new system without the use of the system in the real world [63-67].

In Chapter 3, experiments in the empirical measurement were designed using an approach of energy profilers for smart phones [41]: the energy inside applications is explored by using three steps. In the first step, we tracked and recorded application's activities for further analysis, and in the second step, the energy activities of the smart phone's hardware components are tracked. Thirdly, collected data from the previous two steps are correlated for further analysis. The collected data can allow determining the causes of energy waste from applications' activities and smart phone hardware components.

In the first step, Logcat [68, 69] is used in the empirical measurement to profile and collect applications' activities. The Android logging system (also called Logcat) was developed to collect and view the system debug output of Android applications including error messages and messages that developers write from applications. To use Logcat, we need to set up the Android development environment in Eclipse which is a multi-language software development tool [70, 71] first. Eclipse can be written in many languages such as Java, PHP, Perl, Ruby and so on. The easiest way to set up the Android development environment is to download the Eclipse IDE with built-in Android Developer Tools (ADT) which is a plug-in for Eclipse [72]. ADT contains Android Software Development Kit (Android SDK) with the API libraries and the essential developer tools which are included Logcat and Traceview [73].

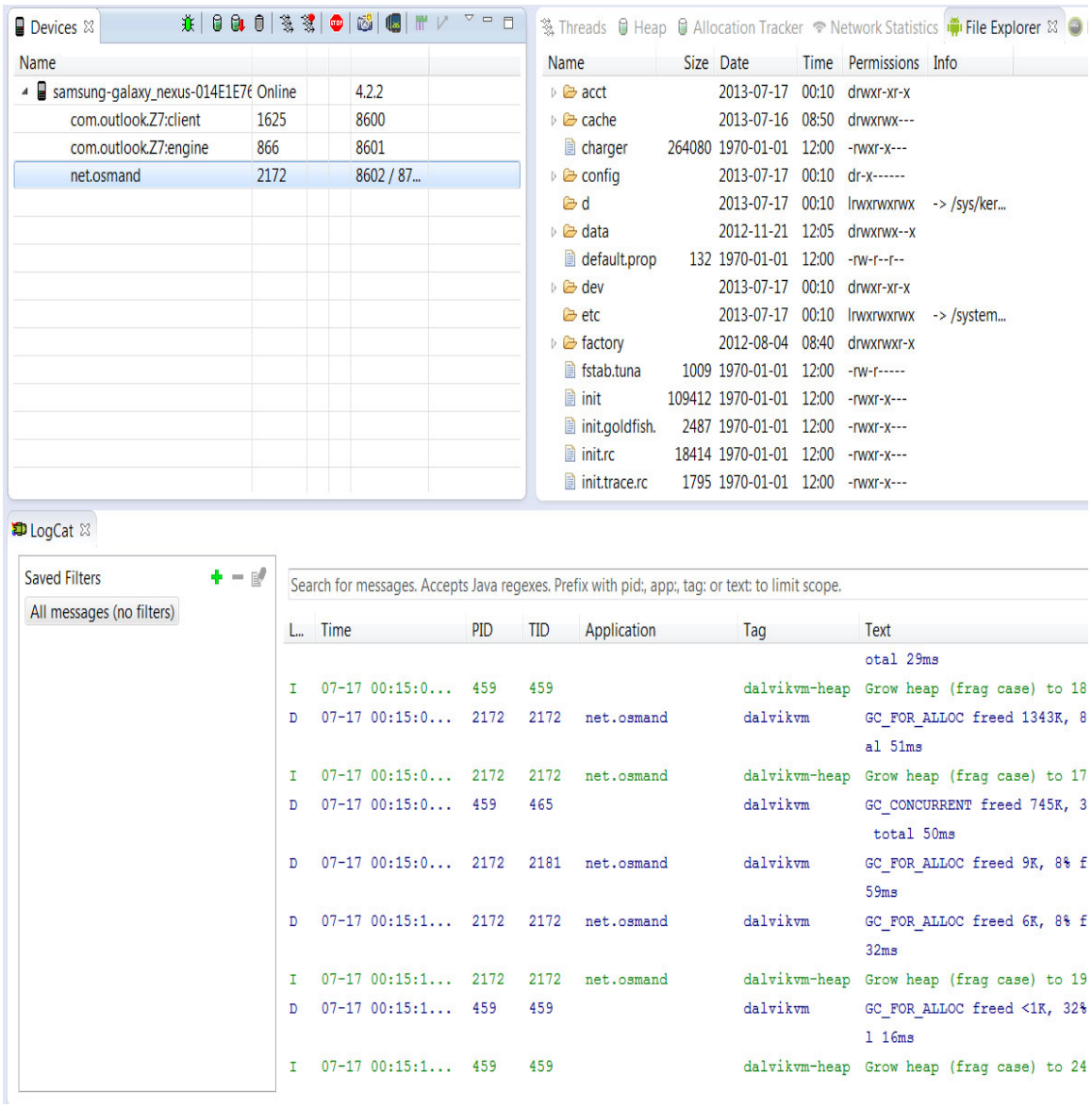


Figure 3.3 - DDMS with Logcat

Logcat can be used from the Dalvik Debug Monitor Server (DDMS) in Eclipse as it seen in Figure 3.3 which shows a screenshot of DDMS with Logcat when a smart phone is connected to a computer. DDMS is a debugging tool which provides Android developers the system debug output (from Logcat), heap usage for a process, tracking memory, the network traffic and so on [74].

In the second step, it is to track the energy activities of the smart phone's hardware components. In the study [13, 26], Monsoon [75] was used to measure energy in smart phones. Monsoon is the power monitor hardware which works together with Power Tools

software to give an energy measurement for mobile devices. However, Monsoon costs $771 USD and a buyer has to be responsibility for shipping cost, tax, and export charges. This will be costly for the thesis. Moreover, the measurement results from Monsoon do not show energy consumption of each hardware components in a smart phone [76]. Fortunately, there is an alternative way to measure energy consumption in a smart phone. The study [13] developed an Android application names 'PowerTutor' which has energy models based on experimental results from Monsoon the power monitor. Also, the study [4] used PowerTutor to measure energy consumption of Android applications. Therefore, PowerTutor was used in the second step to track the energy activities of the smart phone's hardware components.

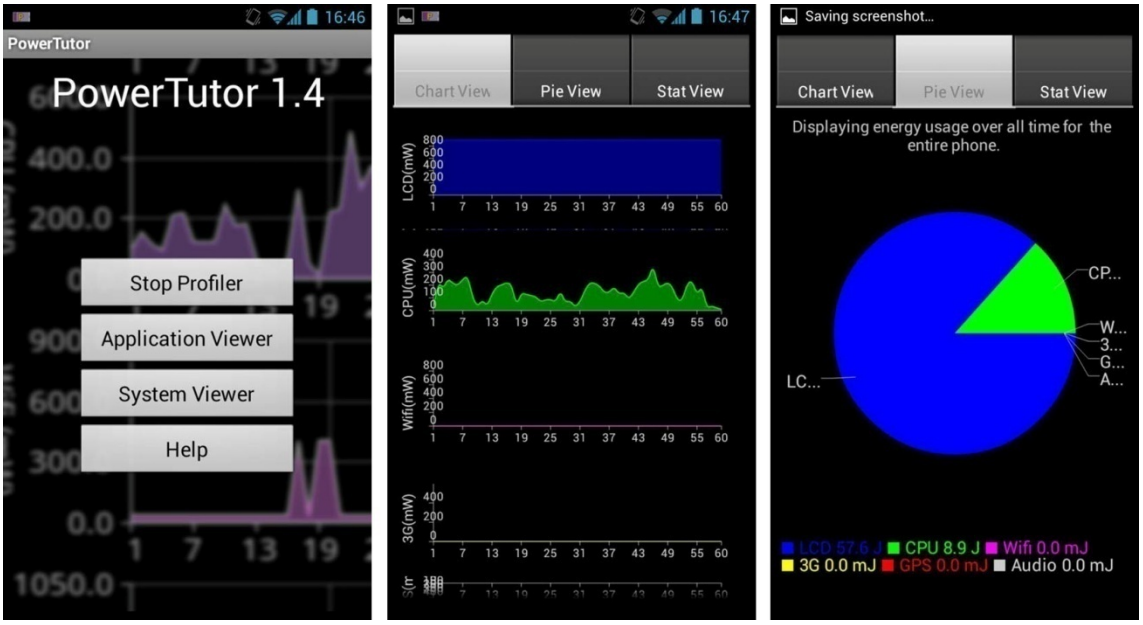


Figure 3.4 - Screenshots of PowerTutor

PowerTutor [14] is an Android application that is used to collect energy consumption statistics for hardware components including CPU, 3G, LCD and GPS for Android applications. There are a few states for each smart phone hardware component that cause energy consumption. Information of energy consumption is given by PowerTutor in term of Watt per second. The power model of LCD displays [13], for example, is determined by the brightness level and whether it is in the on or off state.GPS is measured by the power states of the GPS device, which include active, sleep and off states. Figure 3.4 shows screenshots of PowerTutor; the screenshot on the left side is the main menu of

PowerTutor, while the screenshot in the middle shows the energy consumption by displaying various charts and the screenshot on the right side shows the energy consumption as a pie chart. However, in the energy measurement results from PowerTutor, PowerTutor does not determine which threads in the CPU component consume energy. Hence, Traceview is used to perform energy accounting of an application and find out which CPU threads cause energy consumption and energy waste [41]. In Traceview, execution logs and application performance are profiled and shown as a graphic, a timeline panel and a profile panel [77, 78].

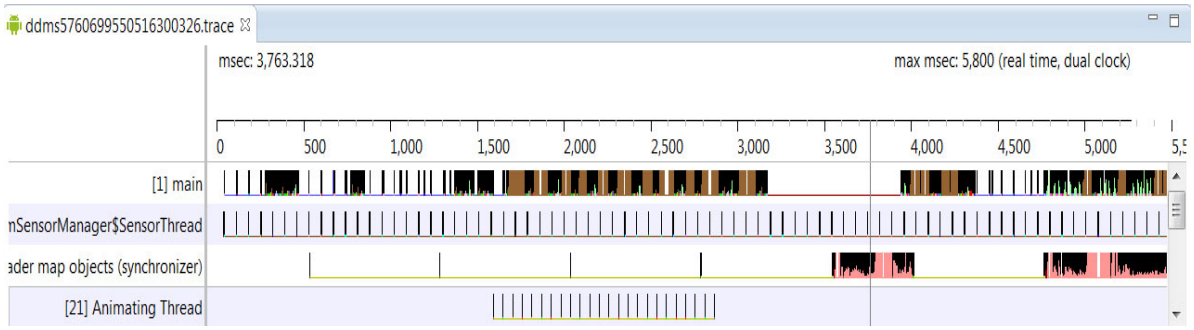


Figure 3.5 - Screenshot of Traceview timeline panel

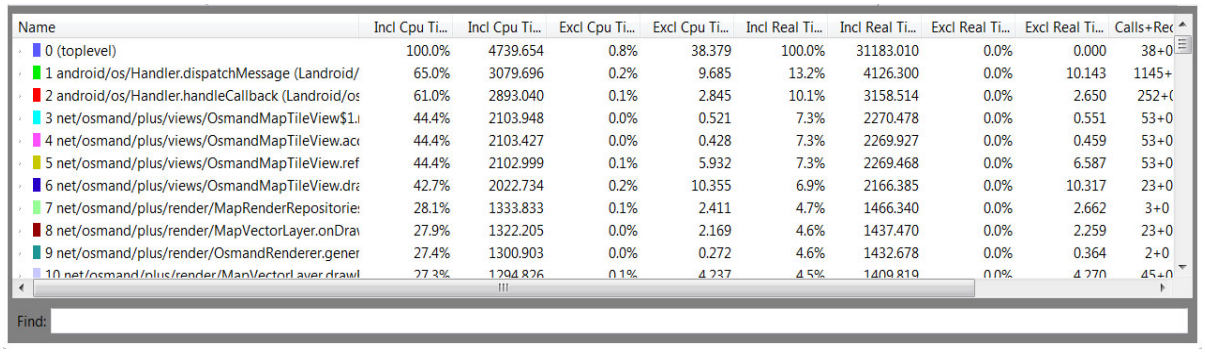


Figure 3.6 - Screenshot of Traceview profile panel

Figure 3.5 shows the timeline panel of the traffic usage of an application. On the left side, there are names of threads such as the main thread, the sensor thread, etc., which are shown on separate rows. Bars on the right side next to the names of threads are thread executions. Each bar shows the methods that are used for the thread. Also, each method has its own colour and in Figure 3.6, it shows the profile panel, which displays the time usage for each method.

A Samsung Galaxy Nexus smart phone was used in this measurement study. It has a Dual-core 1.2 GHz Cortex-A9 as its CPU and runs on Android platform version 4.2.2 which is also called Jelly Bean [79]. The smart phone tested had Osmand and PowerTutor installed. The phone also needed to connect with a computer installed with Eclipse IDE and built-in ADT before collecting data to use Logcat [74].

3.3 Measurement Results

Here are the results of experiments on a smart phone in the empirical measurement using the method that we mentioned in section 3.2. As mentioned earlier, the only differences between non-cloud-based and cloud-based Osmand are their navigation services and how these services find route results. We refer to finding route results as the "calculating route" activity.

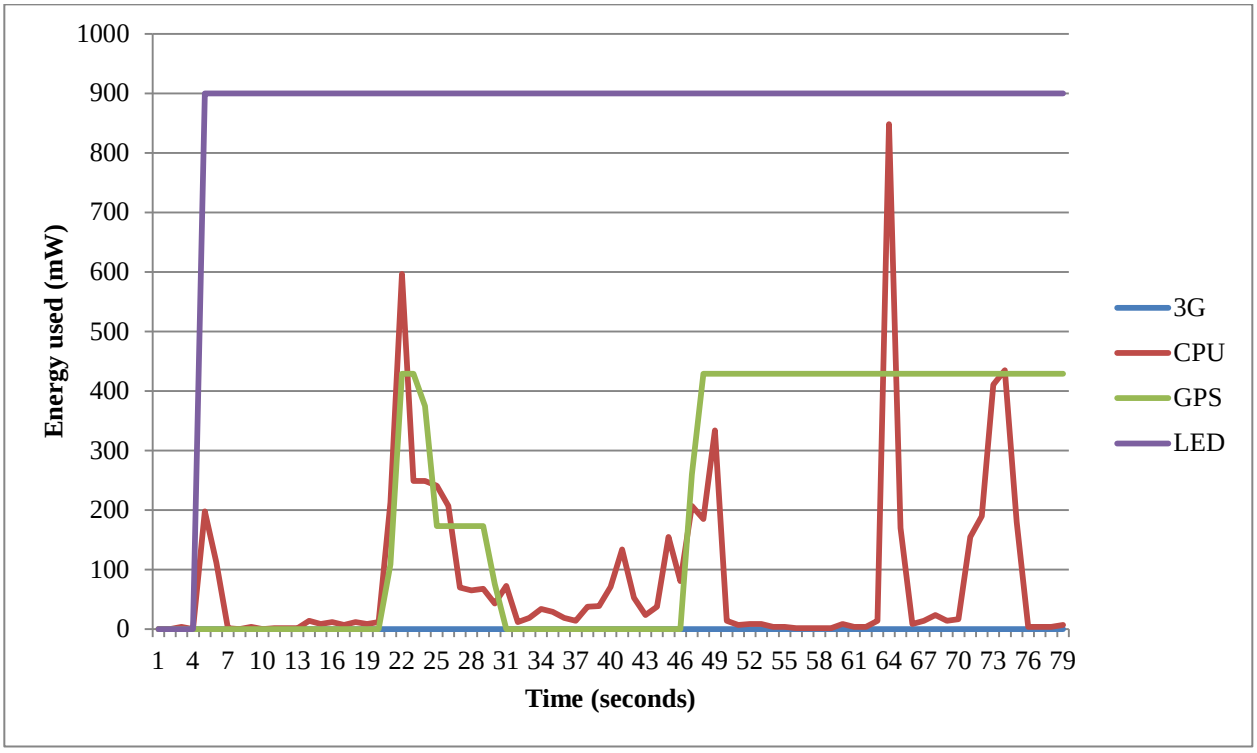


Figure 3.7 - Energy usage of hardware components of non-cloud-based Osmand

Non-cloud-based Osmand uses OsmAnd as its navigation service and OsmAnd uses the A* algorithm [56, 57] to find route results. Therefore, energy consumption of hardware components in non-cloud-based Osmand involves the CPU, LCD and GPS while 3G was

turned off as can be seen in Figure 3.7. On the other hand, cloud-based Osmand uses CloudMade as its navigation service and network communication is needed for cloud-based Osmand to receive route results from CloudMade. So, energy consumption of hardware components in cloud-based Osmand involves the CPU, 3G, LCD and GPS as can be seen in Figure 3.8.

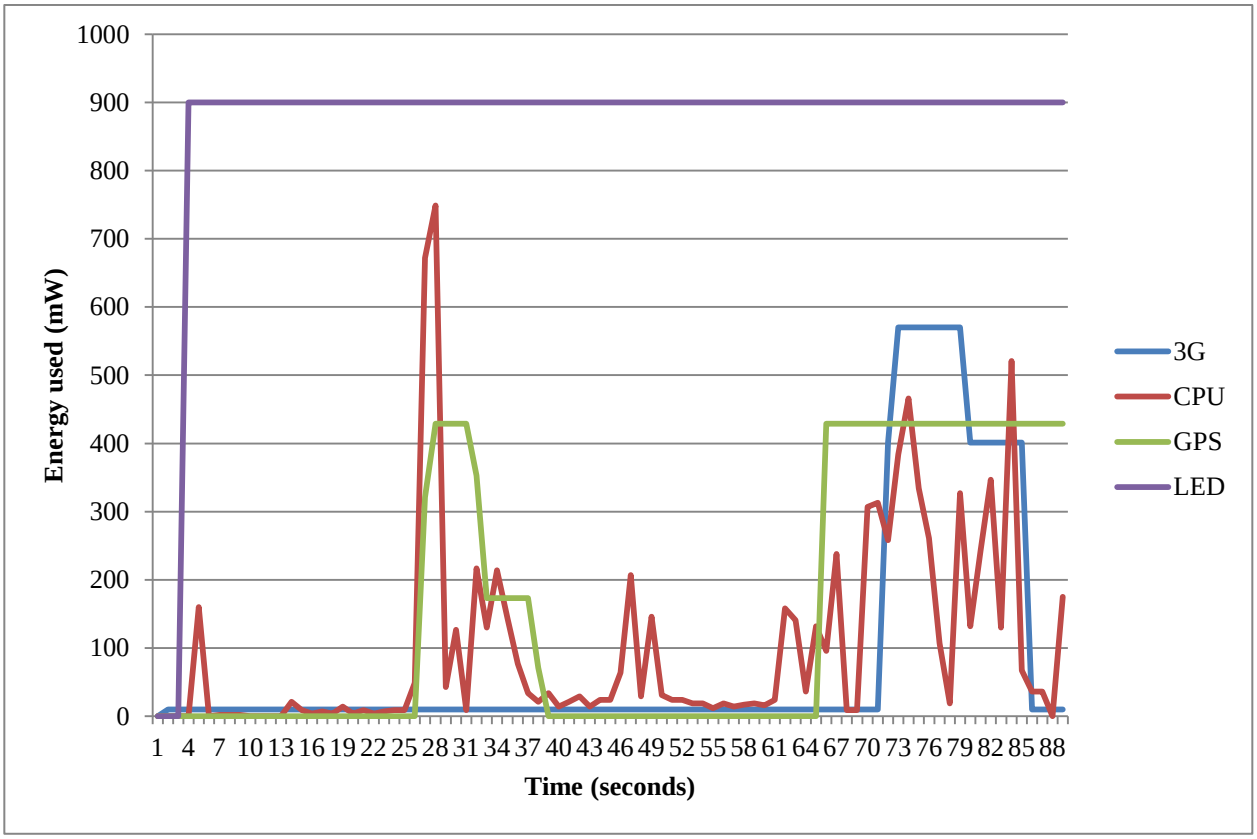


Figure 3.8 - Energy usage of hardware components of cloud-based Osmand

3.3.1 LCD energy consumption

For this measurement, the tested smart phone was set at full brightness; Figure 3.7 shows the energy used for hardware components of the non-cloud-based Osmand. At second 4 and onwards, Osmand begins working and displays on the mobile screen. As a result, the LCD consumes 900mW per second to display Osmand on the screen. Moreover, Figure 3.8 shows the energy used for hardware components of cloud-based Osmand. Osmand starts working and is shown on the screen at second 3, so the LCD consumes 900mW per second. Therefore, LCDs in non-cloud-based Osmand and cloud-based Osmand consume the same

amount of energy to display the application. When Osmand stops working or the tested smart phone is in the sleep mode, the LCD consumes 0mW per second.

3.3.2 GPS energy consumption

As shown in Figure 3.7 (non-cloud-based Osmand), from seconds 19 to 22, a user selects to show their current position on a map. During this interval in the active state, the GPS consumes 429mW per second. Then after second 22, the user selects the search button to look for a destination and as Osmand does not need coordinates for this function, the GPS moves to the sleep state and consumes 173mW per second. As it is not being used at this point, the GPS stays in that sleep state for five seconds before it turns off. When non-cloud-based Osmand starts navigating, the GPS switches to the active state and consumes 429mW per second from second 47 onwards as you can see in Figure 3.7.

As seen in Figure 3.8 (cloud-based Osmand), from seconds 26 to 30, Osmand shows the user's current position on a map. This causes the GPS to consume 429mW per second to detect the coordinates. Next at seconds 31 to 64, Osmand uses the search function and does not need the GPS, so the GPS switches to the sleep state and then it turns off. When the navigation starts working at second 65, the GPS turns on and consumes 429mW per second.

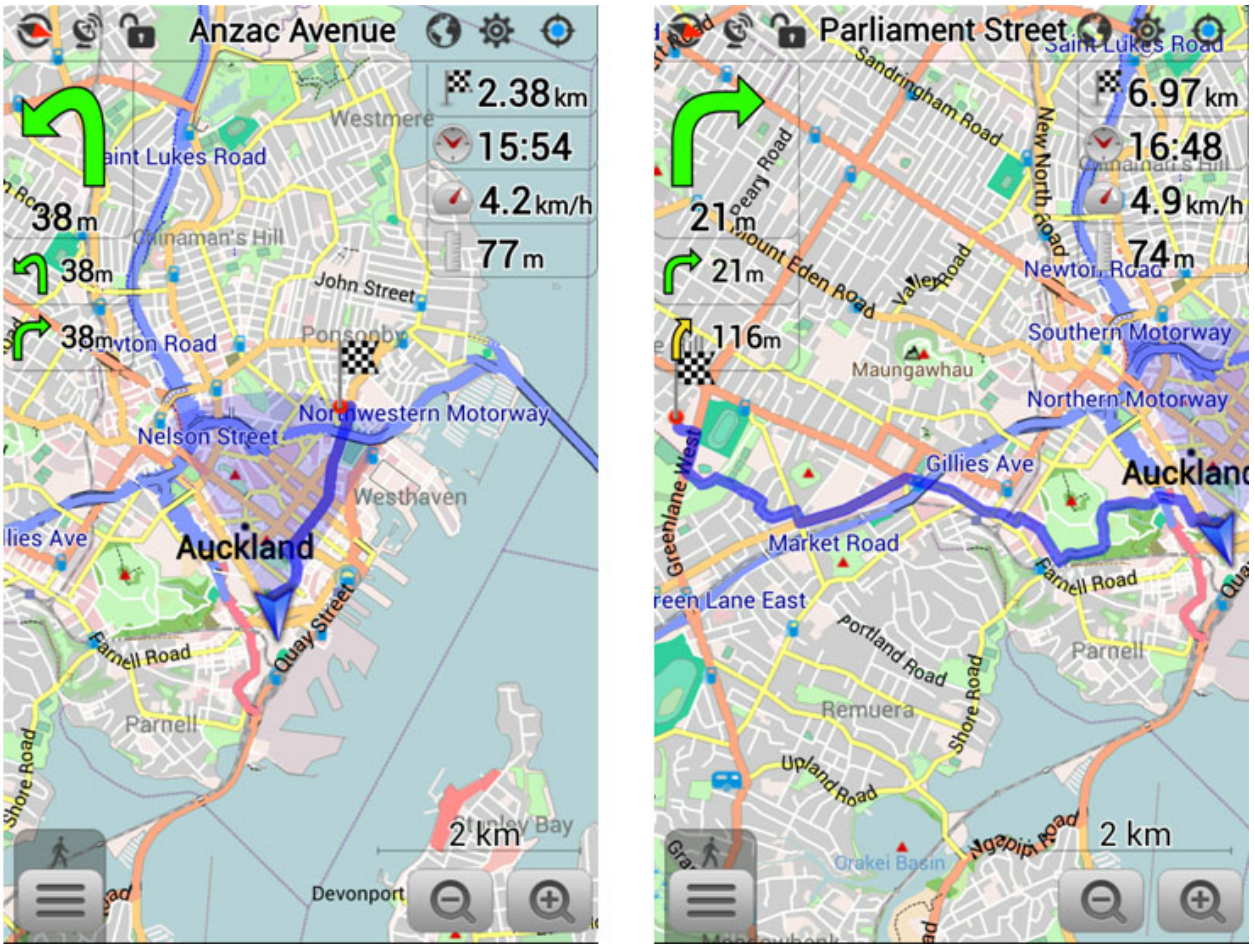
As seen in Figures 3.7 and 3.8, the GPS consumes similar amounts of energy in the non-cloud-based and cloud-based Osmands.

3.3.3 CPU energy consumption

First, CPU energy consumption of non-cloud-based Osmand will be explained and then CPU energy consumption of cloud-based Osmand will follow.

3.3.3.1 CPU energy consumption of non-cloud-based Osmand

In the calculating route activity for non-cloud-based Osmand, the CPU consumes varying amounts of energy from 500mW to 1500mW depending on the complexity of the map between the starting point and destination point as it can be seen in Figure 3.9.



(a) (b)

Figure 3.9 - Potential routes between starting points and destination points

In Figure 3.9(a), the starting point is on Parliament Street and the destination point is the New World supermarket in Victoria Park. Moreover, the route distance between on Parliament Street and the New World supermarket is 2.38km. The CPU of non-cloud-based Osmand consumes about 1412mW for calculating route activity of Figure 3.9(a).

However, in Figure 3.9(b), the destination point is changed to Greenlane Hospital and the starting point is the same point as Figure 3.9(a). In addition, the route distance from on Parliament Street to Greenlane Hospital is 6.97km. The CPU of non-cloud-based Osmand consumes approximately 599mW for calculating route activity of Figure 3.9(b).

From the finding, the route in Figure 3.9(a) is shorter than the route in Figure 3.9(b) but calculating route activity of Figure 3.9(a) uses the larger amount of CPU energy consumption than calculating route activity of Figure 3.9(b). It is because the route in

Figure 3.9(a) has more possible ways from the starting point to the destination point than the route in Figure 3.9(b). Therefore, CPU in Figure 3.9(a) has heavier computation to search for the route than CPU in Figure 3.9(b). It can be concluded that the complexity of maps affects the amount of CPU energy consumption.

Figure 3.10 - Timeline panel of each thread's execution of non-cloud-based Osmand for calculating route activity

The CPU execution is logged by Traceview while PowerTutor measures energy consumption. The CPU execution logs correspond to the calculating route activity in Figure 3.9. The left side of Figure 3.10 shows the threads referring to the calculating route activity of non-cloud-based Osmand. The bar charts next to the threads show the traffic of thread usages.

When an Android application starts working, a thread of execution called the "main" is automatically created by the system for the application. The main thread's role is to dispatch events to the suitable user interface widgets. Additionally, the main thread communicates the application to components of the Android UI toolkit [80]. Thus, there is heavy traffic during the main thread's execution. Furthermore, the calculating route thread of non-cloud-based Osmand in Figure 3.10 performs in a local device. This requires heavy computation and there is a lot of usage traffic. After the calculating route thread begins the searching route process, a loader map object thread is launched to prepare data objects to display on the mobile screen. Also, a garbage collection thread is created due to the heavy computation load in the three previously mentioned threads.

The energy consumptions of each thread were calculated by estimating call counts or CPU utilization per routines [41, 81] from Traceview. Drawing data from Traceview in Figure 3.10, the energy consumption of each thread in the calculating route activity of non-cloud-based Osmand is summarised in Figure 3.11.

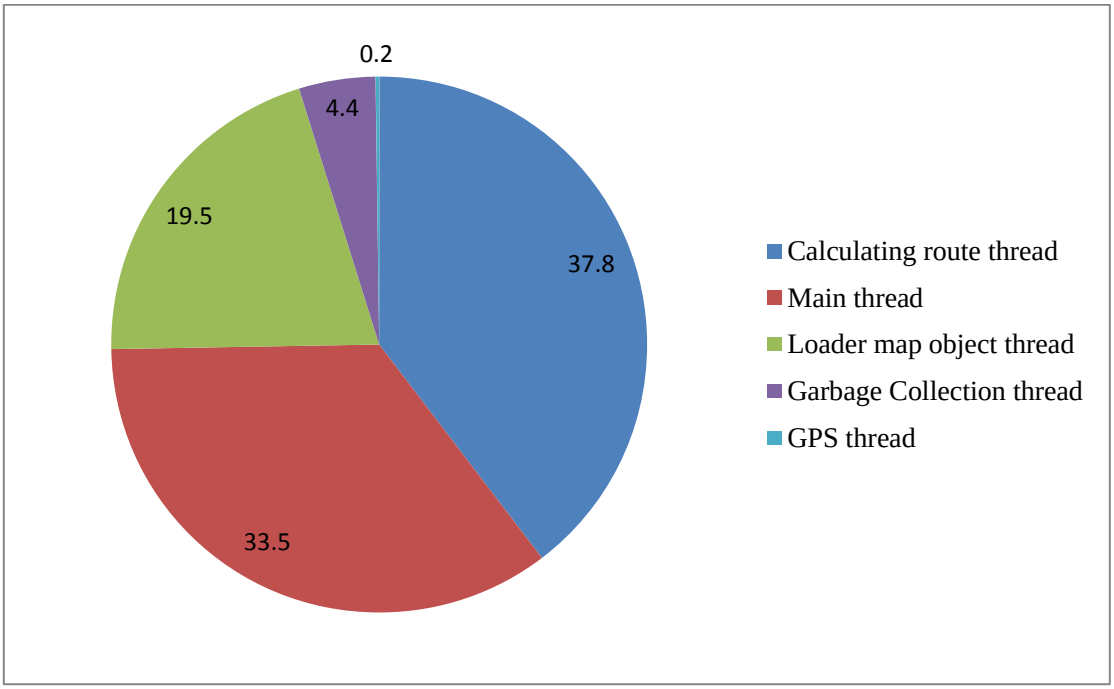


Figure 3.11 - Energy consumption of each thread in the calculating route activity of non-cloud-based Osmand

Figure 3.11 shows a summary of the thread energy usage during the calculating route activity of non-cloud-based Osmand. Due to the heavy computation required to search for a route result, the calculating route thread accounts for approximately 37.8% of total CPU energy consumption. This supports our finding in Figure 3.9 that the complexity of maps influences the amount of CPU energy consumption. Also, the main thread accounts for about 33.5% of total CPU energy consumption is to create events for Osmand. While non-cloud-based Osmand searches for route results, the loader map object starts preparing map objects for overlay on the map. This accounts for approximately 19.5% of total CPU energy consumption. Due to the heavy computation required in these three threads, the garbage collection thread is used to reclaim memory that is no longer in use; it accounts for 4.4% of total CPU energy consumption. The sensor thread, which uses GPS to detect the user's location, accounts for 0.2% of total CPU energy consumption.

Hence, the calculating route thread of non-cloud-based Osmand was analysed more in-depth to determine which functions involved energy consumption of the calculating route thread in Figure 3.12.

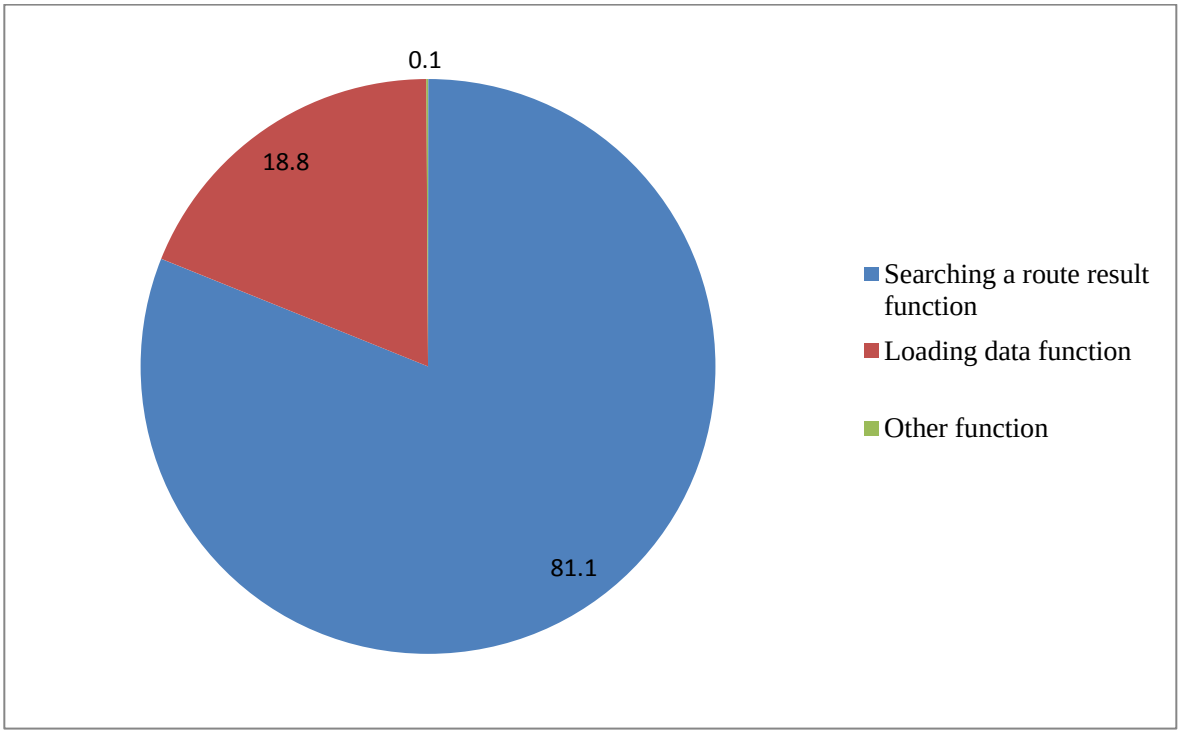


Figure 3.12 - Energy consumption of functions involved in the calculating route thread of non-cloud-based Osmand

The navigation service 'OsmAnd' for non-cloud-based Osmand uses the A* algorithm [56, 57], so the calculating route thread is computation intensive. As a result, the searching route result function accounts for over 80% of the total energy of the calculating route thread, while the loading data function accounts for about 18.8% of the total energy of the calculating route thread.

3.3.3.2 CPU energy consumption of cloud-based Osmand

In the calculating route activity of cloud-based Osmand, the CPU uses the range energy consumption between 900mW and 2300mW. Two examples from our measurement results show that the CPU of cloud-based Osmand consumes 2250mW for calculating route activity from Parliament Street to the New World supermarket in Victoria Park as seen on Figure 3.9(a). Additionally, the CPU of cloud-based Osmand consumes 1509mW for

calculating route activity from Parliament Street to Greenlane Hospital, which is the route displayed on Figure 3.9(b).

Our study shows that the CPU of cloud-based Osmand consumes more energy that the CPU of non-cloud-based Osmand, which consumes only 599mW to find a route result from Parliament Street to Greenlane Hospital and 1412mW to find a route result from Parliament Street to the New World supermarket in Victoria Park.

Thus, Traceview becomes useful for studying the execution threads inside CPU when PowerTutor cannot inform the execution thread inside the CPU. Traceview can profile execution logs and application performance, which are shown as a graphic.

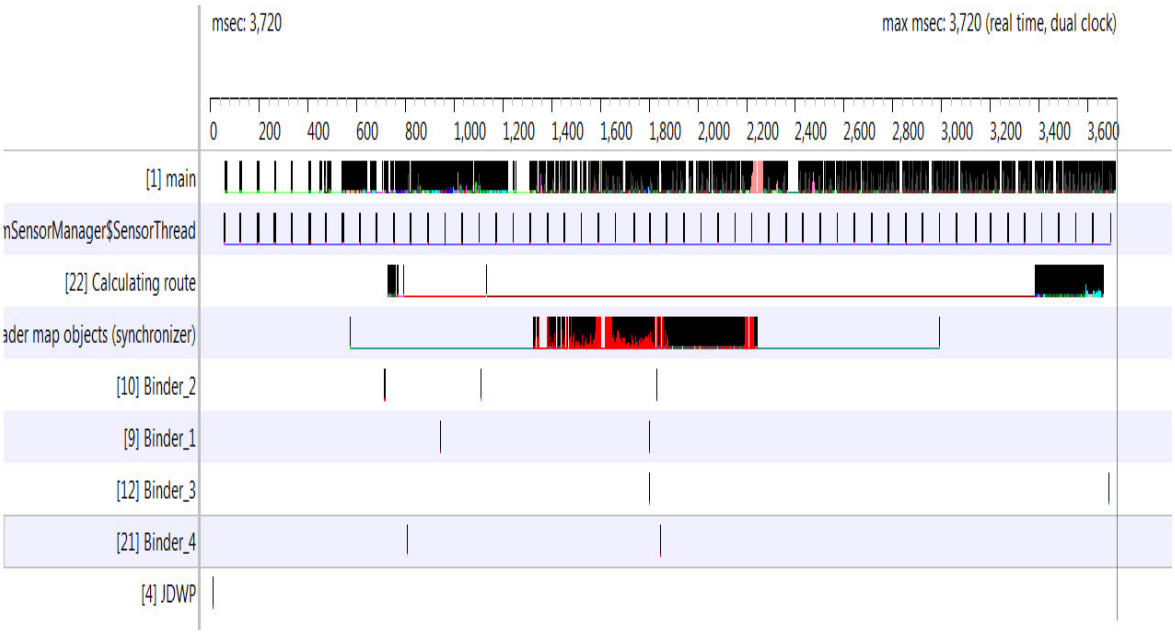


Figure 3.13 - Timeline panel of each thread's execution of cloud-based Osmand in the calculating route activity

The CPU execution logs correspond to the calculating route activity of cloud-based Osmand, which is shown in Figure 3.13. The left side of Figure 3.13 shows the threads that are used in the calculating route activity of cloud-based Osmand. The bar charts next to the threads show the thread traffic usages.

As cloud-based Osmand sends the task of searching route results to CloudMade, there is a smaller amount of traffic usage for the calculating route thread compared with traffic usage in the calculating route thread of non-cloud-based Osmand in Figure 3.10. The

traffic usage of the calculating route thread is only at the start when cloud-based Osmand sends a request file to CloudMade to search for route results and at the end when cloud-based Osmand receives route results back from CloudMade and parses the results. On the other hand, the main thread and the loader map object thread work the same as the threads of non-cloud-based Osmand. From the Traceview data in Figure 3.13, we can summarise the energy consumption of each thread in the calculating route activity.

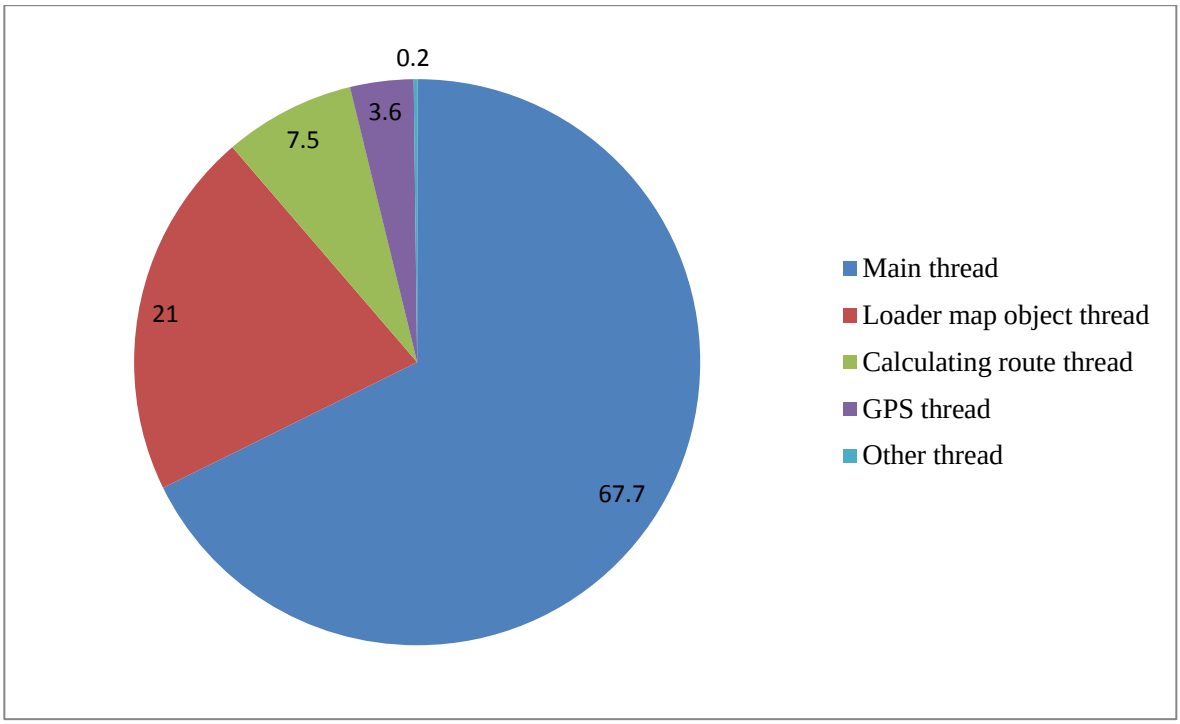


Figure 3.14 - Energy consumption of each thread in the calculating route activity of cloud-based Osmand

Figure 3.14 shows the summary of thread energy usage of the calculating route activity of cloud-based Osmand. The main thread accounts for the largest amount of energy consumption—approximately 67.7% of the total CPU energy use of cloud-based Osmand. The loader map object accounts for about 21%, while the calculating route thread accounts for only 7.5% of the total CPU energy consumption. The sensor thread, which relates to the GPS and the other threads, consumes 3.6% and 0.2%.

In the calculating route activity of cloud-based Osmand, the main thread consumes more energy and functions differently than the main thread of non-cloud-based Osmand.

Therefore, the main thread and the calculating route thread were analysed in-depth to compare them with non-cloud-based Osmand as it seen in Figure 3.15.

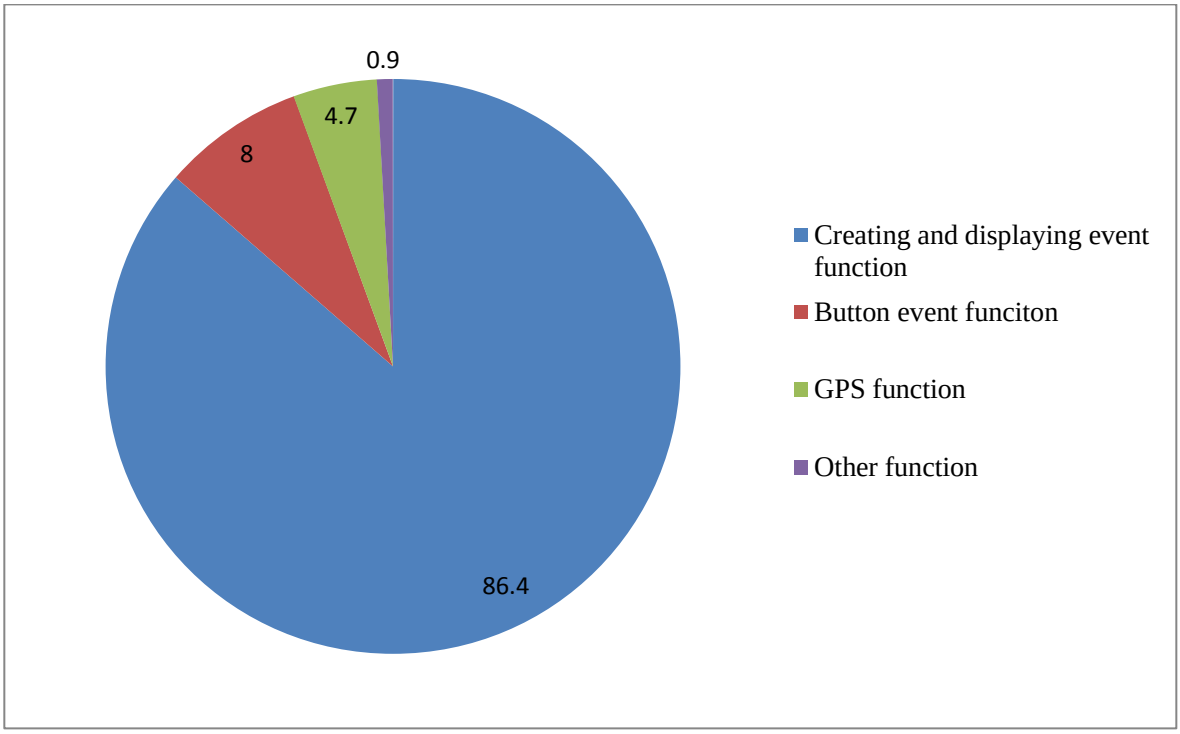


Figure 3.15 - Energy consumption of functions related to the main thread

The main thread of cloud-based Osmand accounts for 86.4% of the total energy consumption of the main thread for creating and displaying event on the mobile screen. The button event accounts for 8% of the total energy consumption while the system sensor function accounts for about 4.7%. The other functions only account for 0.9%. From this information, we can summarise that programming and source code make Osmand's display function in cloud based Osmand consume the large amount of energy and cause energy inefficiency for cloud-based Osmand. However, this is out of our study scope, so we do not study further in application side to improve display function of cloud-based Osmand.

Next, the calculating route thread will be analysed in-depth as seen in Figure 3.16.

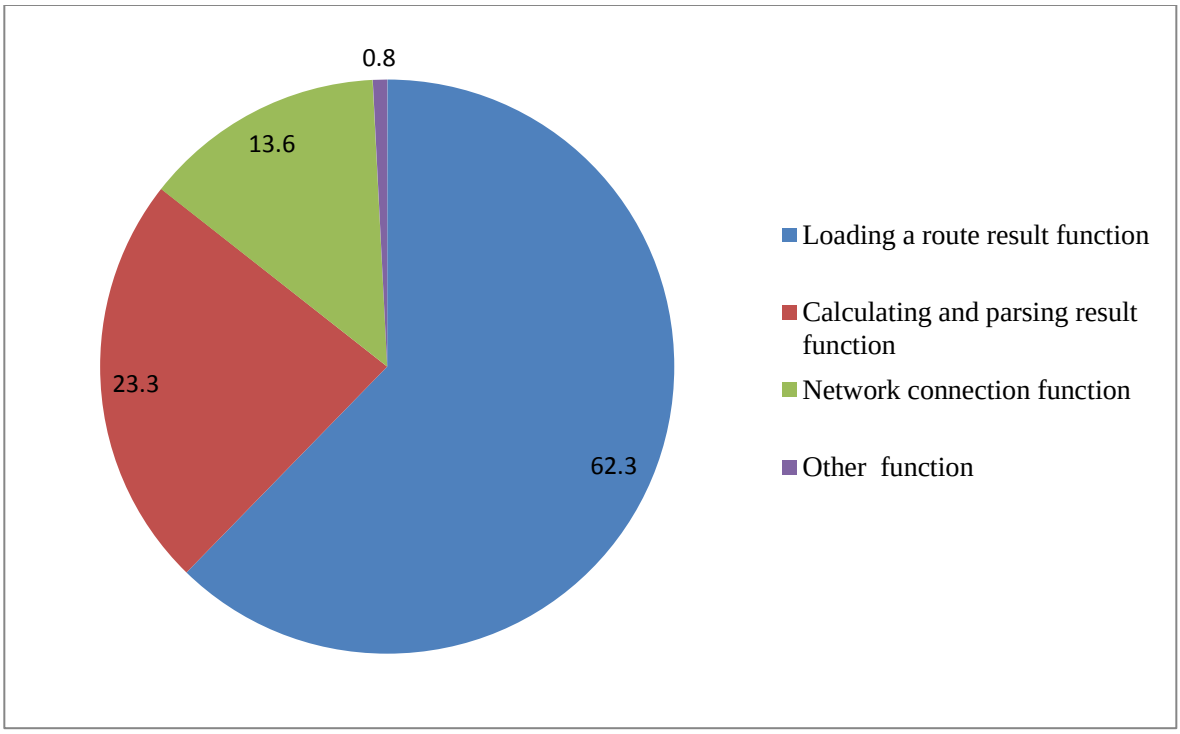


Figure 3.16 - Energy consumption of functions involved in the calculating route thread

As cloud-based Osmand uses the navigation service CloudMade, there is a small amount of energy used for the calculating route thread and loading the route result function accounts for 62.3% of the total energy consumption of the calculating route thread. 23.3% is used for calculating and parsing a route result function, while the network connection (which is 3G communication in this study) accounts for 13.6%. The other functions consume 0.8%.

3.3.3.3 Summary

Based on the results of the empirical measurement, the complexity of maps has influence on the CPU energy consumption of non-cloud-based Osmand because non-cloud-based Osmand uses A* algorithm [56, 57] for calculating route activity which makes heavy computation for CPU of non-cloud-based Osmand. As a result, this causes the energy consumption of CPU for non-cloud-based Osmand. On the other hand, cloud-based Osmand has less computation for calculating route activity because cloud-based Osmand has CloudMade processes the calculating route activity for cloud-based Osmand. However, the CPU of cloud-based Osmand consumes more energy than the CPU of non-cloud-based

Osmand because the main thread in the calculating route activity of cloud-based Osmand uses a large amount of energy to generate and display maps on the mobile screen. As a result, the CPU of cloud-based Osmand is more energy in-efficient than the CPU of non-cloud-based Osmand.

3.3.4 3G communication energy consumption

Cloud-based Osmand requires network connection to send a request file to CloudMade and receive a route result back from CloudMade. Hence, 3G communication energy consumption is considered as another factor that causes energy consumption in cloud-based Osmand. However, 3G communication does not apply to non-cloud-based Osmand as a factor which causes energy consumption.

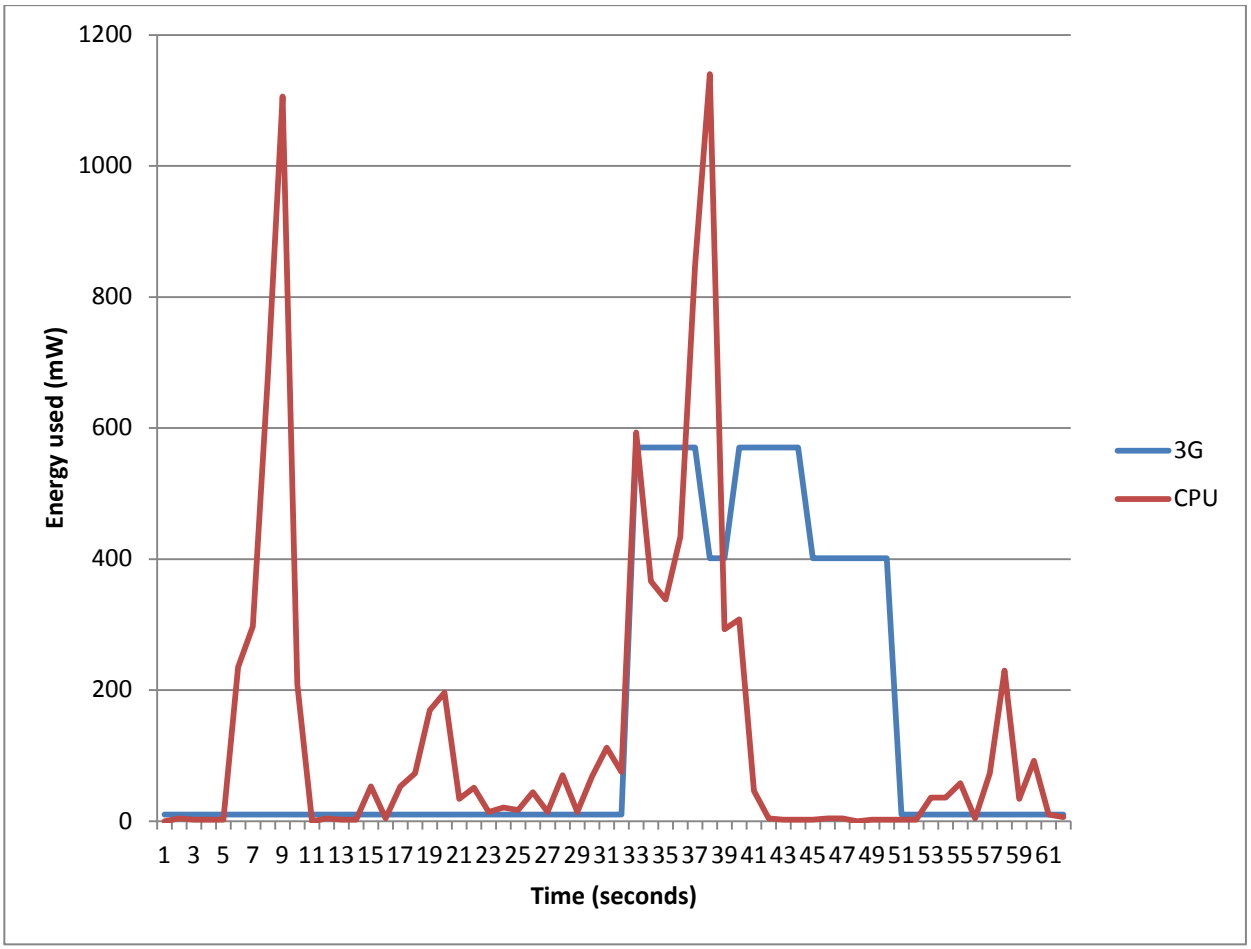


Figure 3.17 - Energy usage of CPU and 3G communication of cloud-based Osmand

Figure 3.17 shows the results of the CPU and 3G communication energy usage of cloud-based Osmand. Based on Figure 3.17, cloud-based Osmand loads a map and shows a user's location from seconds 8 to 27. Between seconds 28 and 32, the user searches for a destination address. Cloud-based Osmand sends a request file to CloudMade at second 33 and it consumes 570mW to send the file. After sending the request file and with no other files arriving, the 3G interface still consumes 570mW per second until second 37. Next, the energy consumption of the 3G interface drops to 401mW. At second 40, cloud-based Osmand receives the route result back from CloudMade, and this consumes 570mW. Then, the 3G interface consumes 570mW per second for another 4 seconds without file transmitting. Next, the 3G interface consumes 470mW for another 6 seconds. Finally, the 3G interface consumes 10mW per second at second 51. As a result from Figure 3.17, the 3G interface consumes over 5000mW for network transfers. More explanation about how 3G consumes energy will be in the section 4.4.

3.3.5 Summary

The results from our measurement study found that both non-cloud-based Osmand and cloud-based Osmand consume the same amount of energy for the GPS and LCD components. For CPU of non-cloud-based Osmand, the complexity of maps is a factor that causes the CPU energy consumption. However, the CPU of cloud-based Osmand uses more energy compared with the CPU of non-cloud-based Osmand because of the main thread use over 60% of the total energy to create and display events on the mobile screen. Additionally, 3G communication causes cloud-based Osmand to consume over 5000mW for network transfers.

Chapter 4 Analytical Characterization on Energy Consumption

In previous chapter, we showed results of the empirical energy measurement on mobile hardware components of non-cloud-based Osmand and cloud-based Osmand. In this Chapter we formulate the analytical models to characterise the energy consumption of each mobile hardware component involved in the earlier empirical measurement (LCD, GPS, CPU and 3G). Then, knowledge from the analytical models of CPU and 3G is used to design the energy efficient recommendations and EESManager is developed based on the recommendations to improve energy efficiency on cloud-based Osmand and a local device.

The remainder of this chapter is organised as follow. The LCD energy model will be firstly analysed and explained how PowerTutor gets the LCD energy consumption in section 3.3.1 and follow by the GPS energy model. Next, the CPU energy model will be described and also we will explain how A* algorithm affects the CPU energy consumption of non-cloud-based Osmand based on the results of the previous empirical measurement. Then, we will analyse and explain the overview of the 3G communication energy model. Furthermore, we will give 5 possible 3G communication energy models of file transmission for cloud-based Osmand. Finally, the energy efficient recommendations and EESManager will be presented.

4.1 LCD energy model

In the section 3.3.1 in Chapter 3, the LCD energy consumption from the empirical measurement was presented. In this section, we explain how PowerTutor [13, 14] measures the LCD energy consumption in a smart phone. The LCD power model is measured from the on or off state of the LCD and the brightness level of the LCD. Let E_{LCD} be the energy consumption of the LCD of a smart phone device and it can be written down as shown in Table 4.1.

Table 4.1 - The LCD power model, Adapted from Zhang, Tiwana, et al. [13] and PowerTutor's source code [17, 19, 20]

Model	$E_{LCD} = LCD_{on} \times (\beta_{bright} \times brightness + \beta_{backlight})$		
Category	**System variable**	**Range**	**Power coefficient**
LCD	LCD_{on}	0,1	n.a
	Brightness	0-255	β_{brignt} : 2.40276 $\beta_{backlight}$: 287.9606

Where LCD_{on} is the state of LCD. When LCD_{on} is 1, it means LCD is in the on state otherwise it is in the off state. $brightness$ is the brightness level of the LCD mobile screen which can range from 0 to 255, while β_{brignt} and $\beta_{backlight}$ are power coefficients of the LCD power model.

From the measurement results of LCD energy consumption in Chapter 3 (section 3.3.1), we set the tested smart phone at full brightness. So, $brightness$ is 255 for the brightness level and we can calculate LCD energy consumption by using the LCD energy model as below:

$$E_{LCD} = LCD_{on} \times (\beta_{bright} \times brightness + \beta_{backlight})$$

$$E_{LCD} = 1 \times (2.40276 \times 255 + 287.9606)$$

$$E_{LCD} = 900.6644\text{mW}$$

4.2 GPS energy model

In the section 3.3.2 in Chapter 3, we presented the GPS energy consumption from the empirical measurement. In this section, we explain how PowerTutor [13, 14] measures the GPS energy consumption in a smart phone. PowerTutor uses a GPS power model, which is influenced by GPS states: the on state, the sleep state and the off state. However, the number of satellites available or the signal strength has a small influence on energy

consumption. Let E_{GPS} be the GPS energy consumption of a smart phone device and we can write it down as shown in Table 4.2:

Table 4.2 - The GPS power model, Adapted from Zhang, Tiwana, et al. [13] and PowerTutor's source code [16, 19, 20]

Model	$E_{GPS} = (\beta_{Gon} \times GPS\_on) + (\beta_{Gsleep} \times GPS\_sleep)$		
Category	**System variable**	**Range**	**Power coefficient**
GPS	GPS_on	0,1	β_{Gon}: 429.55
	GPS_sleep	0,1	β_{Gsleep}: 173.55

Where GPS_on is the on state of GPS and GPS_sleep is the sleep state. When 1 means that the state is active and 0 means that the state is inactive. The GPS energy model has β_{Gon} and β_{Gsleep} as its power coefficient.

From the measurement results of the GPS energy consumption in Chapter 3 (section 3.3.2) when the GPS is used to show a user's current position on Osmand, we can calculate the GPS energy consumption by using the GPS energy model as below:

$$E_{GPS} = (\beta_{Gon} \times GPS\_on) + (\beta_{Gsleep} \times GPS\_sleep)$$

$$E_{GPS} = (1 \times 429.55) + (0 \times 173.55)$$

$$E_{GPS} = 429.55\text{mW}$$

If Osmand does not need coordinates from the GPS, the GPS changes to the sleep state and it consumes energy as we can calculate below:

$$E_{GPS} = (0 \times 429.55) + (1 \times 173.55)$$

$$E_{GPS} = 173.55\text{mW}$$

If there is no need for coordinates while the GPS is in the sleep state for six seconds, the GPS moves to the off state [16, 19, 20].

4.3 CPU energy model

In the section 3.3.3 in Chapter 3, the CPU energy consumption from the empirical measurement was presented. In this section we explain how PowerTutor [13, 14] measures the CPU energy consumption in a smart phone.

PowerTutor measures the CPU energy consumption influenced by CPU utilisation and CPU frequency using the system frequency file in the /sys file system [13, 15]. Table 4.3 indicates the CPU power model used in PowerTutor and the variables in Table 4.3 show the difference of power use of the CPU during active and idle states. Let E_{CPU} be the CPU energy consumption of a smart phone device and it can be written down as shown in Table 4.3: where $frequency_h$ is the high level of the CPU frequency while $frequency_l$ is the low level of the CPU frequency. Utilization in CPU can range in between 1 and 100, it has β_{uh} and β_{ul} as its power coefficients for the high and low levels of utilization. CPU consumes β_{CPU} mW when CPU is in the on state. Therefore, the CPU energy consumption can be calculated by using the CPU energy model.

Table 4.3 - The CPU power model, Adapted from Zhang, Tiwana, et al. [13] and PowerTutor's source code [13, 15, 19, 20]

Model	$E_{CPU} = (\beta_{uh} \times frequency_h + \beta_{ul} \times frequency_l) \times utilization + \beta_{CPU} \times \text{CPU\_on}$		
Category	**System variable**	**Range**	**Power coefficient**
CPU	Utilization	1-100	β_{uh}: 4.3388 β_{ul}: 3.4169
	frequency_l frequency_h	0,1 0,1	n.a.
	CPU_on	0,1	β_{CPU}: 121.46

In the empirical measurement, we found that the CPUs in non-cloud-based and cloud-based Osmands have different energy consumption for the calculating route activity because non-cloud-based Osmand performs on a local device whereas cloud-based Osmand performs in the cloud. So, CPU energy consumption while calculating route activity will be the focus of this analysis, because non-cloud-based and cloud-based Osmands share the same functions in other activities. Therefore, there is no difference in CPU energy consumption with other activities.

For non-cloud-based Osmand, the complexity of maps has influence on the CPU energy consumption of non-cloud-based Osmand as it can be seen in section 3.3.3.1 because non-cloud-based Osmand uses the A* algorithm approach to search for a route result. So we will explain how A* algorithm works to understand how the complexity of maps involves the CPU energy consumption. The A* algorithm calculates the moving cost between the starting point and the destination point [56, 57].

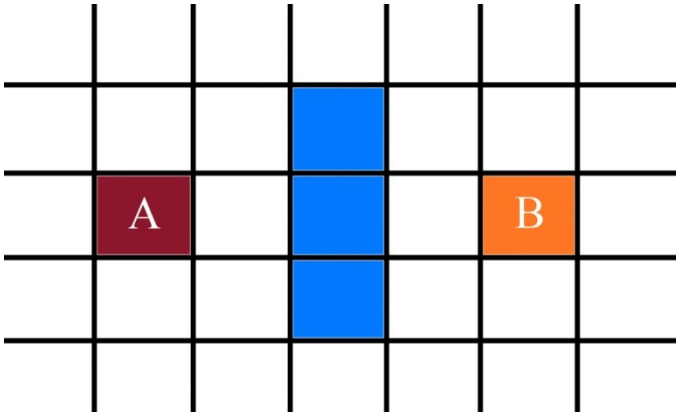


Figure 4.1 - Introduction of A* algorithm approach, Adapted from Lester [57]

Figure 4.1 shows how the A* algorithm works. Let A be a starting point and B a destination point, and the blue block is a wall that separates A and B. The algorithm finds a route from A to B by using path scoring. Assume that the path scoring uses the following equation:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the cost of moving from A to any node n, while $h(n)$ is the estimated cost of moving from node n to the destination, which is referred as B in Figure 3.3. On the

other hand, $f(n)$ is calculated by adding $g(n)$ and $h(n)$. For the next step, the A* algorithm uses Equation 1 as a key to determine the shortest path by starting from A.

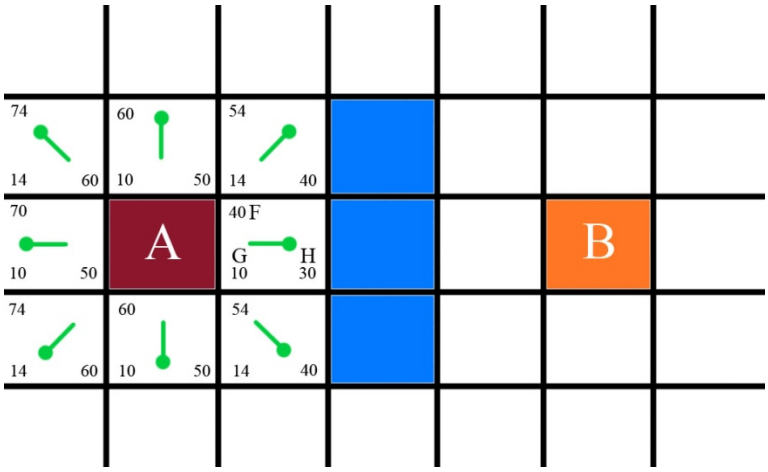


Figure 4.2 - Path scoring of the A* algorithm approach adapted from Lester [57]

Figure 4.2 shows the process of scoring the path of the A* algorithm. As seen in Figure 4.2, there are eight squares surrounding the starting point A. Let us assign $g(n)$ of the horizontal and vertical squares a cost of 10 and $g(n)$ of the diagonal squares a cost of 14, while $h(n)$ can be estimated by calculating the distance to destination point B by moving only horizontally and vertically. For example, in the square that contains letters G, H and F, $g(n)$ is 10 and $h(n)$ is 30. Therefore, $f(n)$ is 40 for the square.

In the next step, the algorithm chooses the lowest score of $f(n)$ from all eight squares surrounded starting point A. In Figure 4.2, the square that has $f(n)$equal to 40 is chosen as the next node; we refer to this as node1. Then the eight squares surrounding node1 are calculated to find $f(n)$. However, if we look at Figure 4.2, there is a blue wall next to node1. So, the algorithm does not calculate $f(n)$ and ignores the wall. Afterwards, node2, which has the lowest F value, is chosen and the process will be repeated until it reaches destination point B.

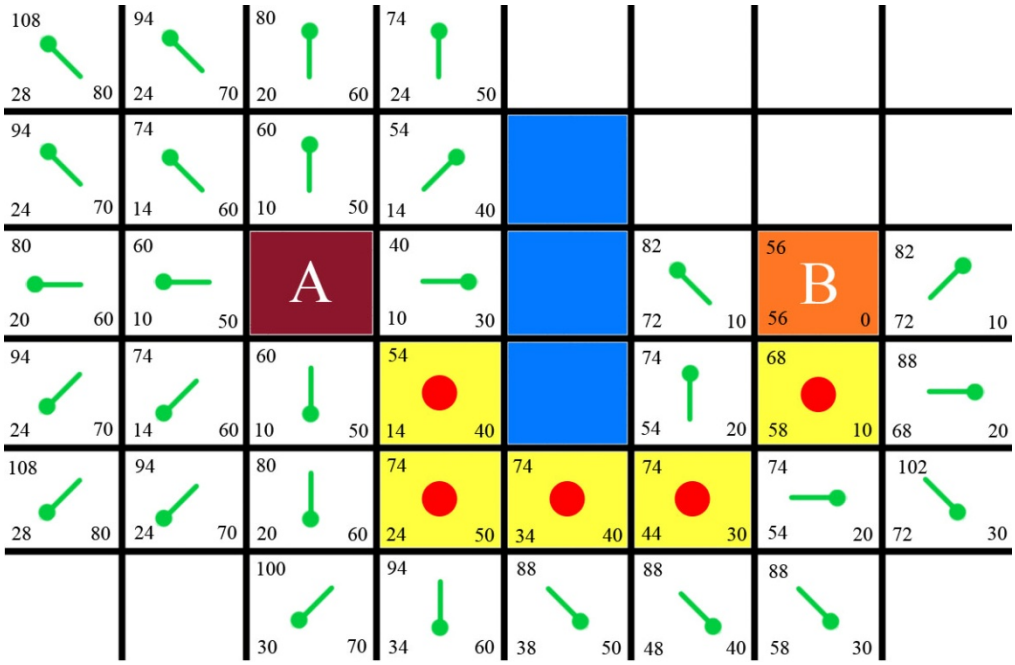


Figure 4.3 - Path determining of the A* algorithm approach, Adapted from Lester [57]

Figure 4.3 shows how the A* algorithm approach determines the shortest path. This process starts from destination point B and moves backward by choosing the lowest F value from one node to the next, which will take the route back to starting point A as you can see from Figure 4.3. From the A* algorithm, we can conclude that the A* algorithm influences heavy computation for calculating route activity which causes the CPU energy consumption in non-cloud-based Osmand. This supports in our finding in section 3.3.3.1 in Chapter 3 that more than 80% of energy consumption of the calculating route thread in non-cloud-based Osmand was used for searching a route result. On the other hand, only 7.5% of the CPU energy consumption of cloud-based Osmand was involved in the route searching process.

4.4 3G communication energy model

As cloud-based Osmand uses CloudMade as its navigation service, cloud-based Osmand requires wireless network communications. For our empirical measurement, we used 3G communications. The 3G power model, which is used in PowerTutor [13] is influenced by the data rate between a local device and the cloud, and the size of transmitted files. However, signal strength is not considered in the 3G power model [13].

Let E_{3G} be the energy consumption of 3G communications of a smart phone device and we can write it down as shown in Table 4.4:

Table 4.4 - The 3G communication power model, Adapted from Zhang, Tiwana, et al. [13] and PowerTutor's source code [14, 18-20]

Model	$E_{3G} = (\beta_{3G_{idle}} \times 3G_{idle}) + (\beta_{3G_{FACH}} \times 3G_{FACH}) + (\beta_{3G_{DCH}} \times 3G_{DCH})$		
Category	**System variable**	**Range**	**Power coefficient**
3G	Data rate	0-∞	n.a.
	Size of files	0-∞	n.a
	$3G_{idle}$	0,1	β_{3G_idle}: 10
	$3G_{FACH}$	0,1	β_{3G_FACH}: 401
	$3G_{DCH}$	0,1	β_{3G_DCH}: 570

Where $3G_{idle}$ is the IDLE state, $3G_{FACH}$ is the FACH state of the 3G interface which has the 3G interface sharing a channel of communication to the base station and the data rate for this state is only a few hundred bytes per second [13], and $3G_{DCH}$ is the DCH state of the 3G interface which uses high-speed data rates for network transmission [13]. Moreover, β_{3G_idle}, β_{3G_FACH} and β_{3G_DCH} are power coefficients for those states.

From the measurement results of the 3G communication energy consumption in Chapter 3 (section 3.3.4) when there is no file transmitting over the network, the 3G interface stays in the IDLE state. It can be calculated energy consumption for this state as below:

$$E_{3G} = (\beta_{3G_{idle}} \times 3G_{idle}) + (\beta_{3G_{FACH}} \times 3G_{FACH}) + (\beta_{3G_{DCH}} \times 3G_{DCH})$$

$$E_{3G} = (10 \times 1) + (401 \times 0) + (570 \times 0)$$

$$E_{3G} = 10\text{mW}$$

If there is a file presented and we assume that the 3G interface moves to the FACH state. We can calculate energy consumption as below:

$$E_{3G} = (10 \times 0) + (401 \times 1) + (570 \times 0)$$

$$E_{3G} = 401\text{mW}$$

On the other hand, if the 3G interface moves, the energy consumption is shown as below:

$$E_{3G} = (10 \times 0) + (401 \times 0) + (570 \times 1)$$

$$E_{3G} = 570\text{mW}$$

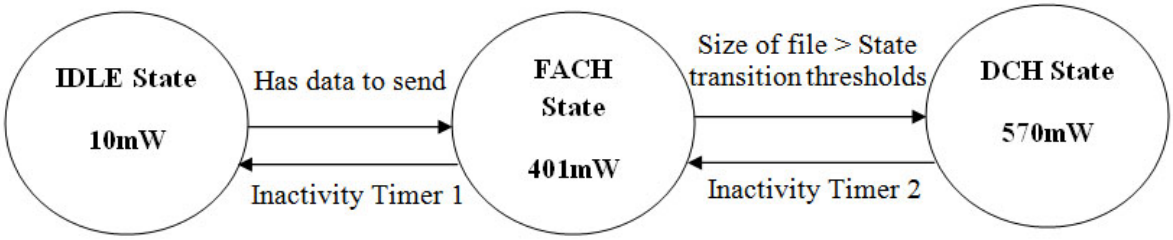


Figure 4.4 - 3G power states (From Zhang, Tiwana, et al. [13])

Figure 4.4 shows how PowerTutor works and measures 3G communication energy consumption. The 3G communication power model from Table 4.4 and Figure 4.4 is based on the size of files. Before 3G communication energy consumption of cloud-based Osmand is explained, the parameters that are used for explaining 3G communication energy consumption of cloud-based Osmand using the 3G power model are listed in Table 4.5.

Table 4.5 - 3G power model parameters for cloud-based Osmand

Symbol	**Meaning**
$time_{DCH}$	Tail time of the DCH state
$time_{FACH}$	Tail time of the FACH state
E_{3G}	3G communication energy consumption
E_{idle}	Energy consumption while in the IDLE state
$E_{request}$	The energy consumption of sending a request file
E_{result}	The energy consumption of receiving a route result file

$size_{request}$	The sizes of a request file
$size_{result}$	The sizes of a route result file
$size_{threshold}$	The size of the state transition threshold

As seen in Figure 4.4, when the 3G interface does not have any files transferred over the network it remains in the IDLE state and consumes 10mW per second. If there is a file sent over the network to a mobile and the file size is less the state transition threshold or $size_{threshold}$, the 3G interface enters the FACH state and consumes 401mW per second to receive the file until the 3G interface finishes the transmission. However, if the file is bigger $size_{threshold}$, the 3G interface enters the DCH state and consumes 570mW per second. In the DCH state, the 3G interface uses high-speed data rates for communication activities [13], so it consumes more energy than when in the FACH state.

When the 3G interface finishes its task and there is no activity required, the interface stay in the same state for fixed periods of time. We call this "tail energy" which is the energy spent to keep hardware components—in this case the 3G interface—in the same power state after finishing their task [13, 41, 47]. For example, the 3G interface is in the DCH state while sending a file. When the interface finishes its task, and if there is no file to send or to receive for Inactivity Timer 2 or $time_{DCH}$ seconds, the 3G interface changes to the FACH state. The interface remains in that state for Inactivity Timer 1 or $time_{FACH}$ seconds and if there is still no activity present, the 3G interface returns to the IDLE state. If there is a file presented while the 3G interface stays in the DCH state, there is no energy cost of the 3G communication for this file. On the other hand, if a file that is bigger than $size_{threshold}$ is presented to the 3G interface while tail energy forces the 3G interface to stay in the FACH state, the interface will enter the DCH state and the interface consumes 570mW for file transmission. However, if the file is less than $size_{threshold}$, there is no energy cost. This finding supports the measurement study [47] that when a smart phone sends the next packet within tail time, there is no cost of transmission and no tail energy penalty. The recommendation of the measurement study [47] is to use multiple transfers to minimise tail energy.

For cloud-based Osmand, it works by sending a request file that contains a starting point, a destination point and an intermediate location list to CloudMade and this consumes $E_{sending}$. Next CloudMade searches for a route result. When this is complete, CloudMade sends the route result back to the local device and this process consumes $E_{receiving}$. The local device then calculates and parses the result into a step-by-step route that is overlaid on a map. So, there are two files involved in this process— the request file and the route result, and these files cause 3G communication energy consumption.

Based on our study, the size of route results or $size_{result}$ is always bigger than $size_{threshold}$. Thus the 3G interface consumes 570mW per second to receive the route results, and we can express the 3G communication energy consumption in the case of cloud-based Osmand as:

$$E_{3G} = E_{request} + E_{result} + \left(time_{FACH}^{request} + time_{FACH}^{result}\right)401 + \left(time_{DCH}^{request} + time_{DCH}^{result}\right)570 + E_{idle} \quad (1)$$

Based our study, we can illustrate five main models of 3G communication energy usage as seen in Figures 4.5 to 4.9.

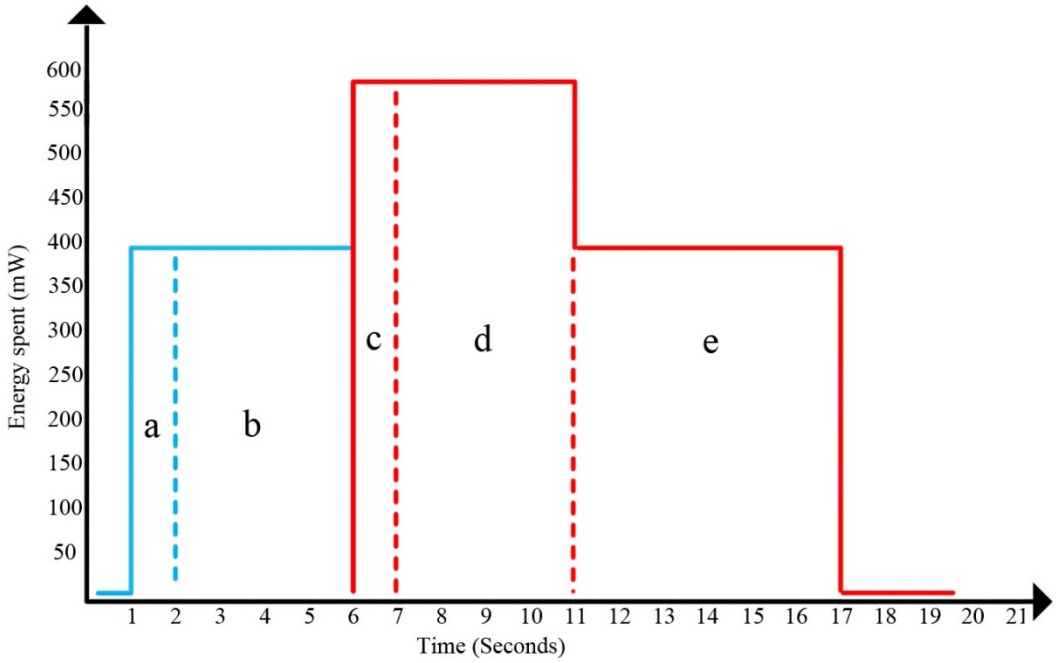


Figure 4.5 - Model 1

Figure 4.5 shows the energy used for 3G communications for model 1 when a request file or $size_{request}$ is less the state transition threshold or $size_{threshold}$ and a route result is sent back within $time_{FACH}$ seconds, where (a) is the energy consumed to send the request file to CloudMade, (b) is the energy consumption of the tail energy for the FACH state, (c) is the energy consumed to receive the route result from CloudMade, (d) is the energy consumption of the tail energy for the DCH state, and (e) is the energy consumption of the tail energy for the FACH state and then the cellular interface enters the IDLE state.

We can express the 3G communications energy consumption from Figure 4.5 using formula(1), where $E_{request}$ represents the energy consumption of (a), $time_{DCH}^{request}$ is 0 because $E_{request}$ is in the FACH state, and $time_{FACH}^{request} < time_{FACH}$because the route result is transferred back to cloud-based Osmand within $time_{FACH}$ seconds. This also means E_{idle} is 0mW. Cloud-based Osmand receives the route result from CloudMade, and it consumes E_{result}, which is referred to as (c). $time_{DCH}^{result}$ is $time_{DCH}$ and $time_{FACH}^{result}$ is $time_{FACH}$. Therefore, 3G communications energy consumption is expressed:

$$E_{3G} = E_{request} + E_{result} + \left(time_{FACH}^{request} + time_{FACH}\right)401 + (time_{DCH})570 \qquad (2)$$

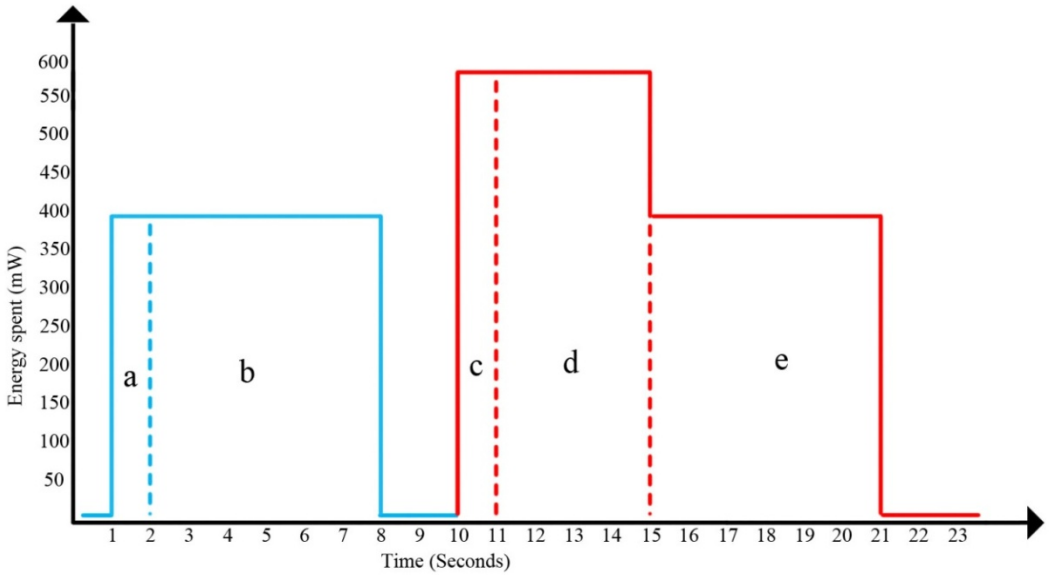


Figure 4.6 - Model 2

Figure 4.6 shows the energy used for 3G communications for model 2 when $size_{request}$ is less than $size_{threshold}$ and a route result is sent back after

$time_{FACH}$ seconds, where (a) is the energy consumed to send the request file to CloudMade or $E_{request}$, (b) is the energy consumption of the tail energy for the FACH state, (c) is the energy consumed to receive the route result from CloudMade or E_{result}, (d) is the energy consumption of the tail energy for the DCH state, and (e) is the energy consumption of the tail energy for the FACH state and then the cellular interface enters the IDLE state.

We can use formula(1) to express the 3G communications energy consumption shown in Figure 4.6, where $E_{request}$ is referred as (a) because $size_{request}$ is less than $size_{threshold}$; $time_{DCH}^{request}$ is 0 because $E_{request}$ is in the FACH state while $time_{FACH}^{request}$ is $time_{FACH}$ because cloud-based Osmand receives the route result from CloudMade after $time_{FACH}$ seconds, which consumes E_{result} to receive the result. Also E_{idle} is addressed in the 3G communications energy consumption because the transmission of the route result happens after $time_{FACH}$ seconds, so the 3G interface enters the IDLE state before continuing. $time_{DCH}^{request}$ is $time_{DCH}$ and $time_{FACH}^{result}$ is $time_{FACH}$. Therefore, 3G communications energy consumption is expressed as formula(3):

$$E_{3G} = E_{request} + E_{result} + \left(time_{FACH}^{request} + time_{FACH}\right)401 + (time_{DCH})570 + E_{idle} \quad (3)$$

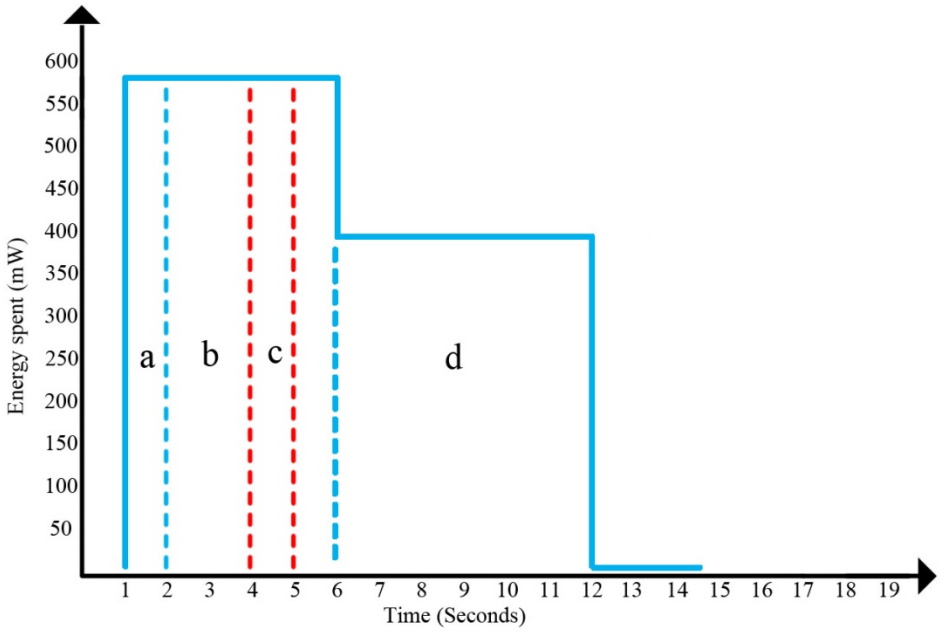


Figure 4.7 - Model 3

Figure 4.7 shows the energy used for 3G communications for Model 3 when $size_{request}$is bigger than $size_{threshold}$ and a route result is sent back within $time_{DCH}$, where (a) is the energy consumed to send the request file to CloudMade or $E_{request}$, (b) is the energy consumption of the tail energy for the FACH state, (c) is the action of receiving the route result from CloudMade but the energy cost of this action or E_{result} does not appear due to the benefit of the tail energy, and (d) is the energy consumption of the tail energy for the FACH state and then the cellular interface enters the IDLE state.

We can use formula(1) to express the 3G communications energy consumption from Figure 4.7, where $E_{request}$ is referred as (a) because $size_{request}$ is bigger than $size_{threshold}$ and $time_{DCH}^{request}$ is $time_{DCH}$ while E_{result} is 0 because of a tail energy advantage and $time_{FACH}^{request}$ is $time_{FACH}$. On the other hand, there is no $time_{FACH}^{result}$ or $time_{DCH}^{result}$ because E_{result} does not occur. Hence, we can show the 3G energy consumption of Model 3 as:

$$E_{3G} = E_{request} + (time_{FACH})401 + (time_{DCH})570 \quad (4)$$

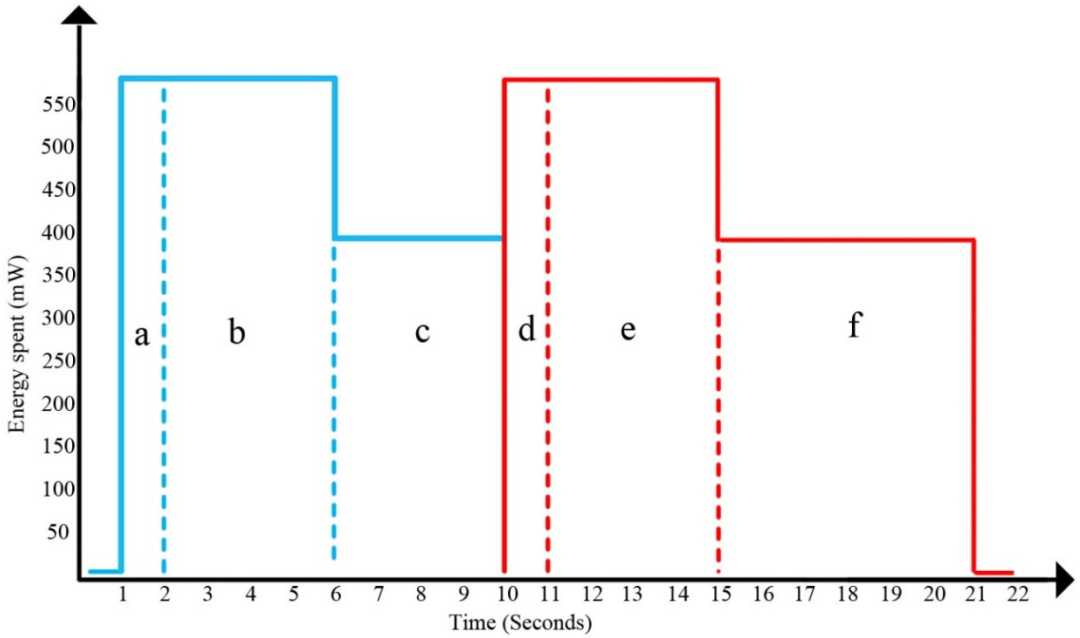


Figure 4.8 - Model 4

Figure 4.8 shows the energy used for 3G communications for Model 4 when $size_{request}$ is bigger than $size_{threshold}$. The cellular interface enters the DCH state. Osmand receives a route result within $time_{FACH}$ seconds. (a) is the energy consumed to transfer the request file, (b) is the energy consumption of the tail energy for the DCH state, (c) is the energy consumption of the tail energy for the FACH state, (d) is the energy consumed to receive the route result, (e) is the energy consumption of the tail energy for the DCH state, and (f) is the energy consumption of the tail energy for the FACH state and then the cellular interface enters the IDLE state.

We can explain the energy consumption of 3G communications shown in Figure 4.8 using formula(1), where $E_{request}$represents the energy consumed to transfer the request file or (a). $time_{DCH}^{request}$ is $time_{DCH}$ and $time_{FACH}^{request} < time_{FACH}$ because the route result is transferred back to cloud-based Osmand within $time_{FACH}$ seconds. Thus, E_{idle} is 0mW and cloud-based Osmand receives the route result from CloudMade, which consumes E_{result}. $time_{DCH}^{result}$ is $time_{DCH}$ and $time_{FACH}^{result}$ is $time_{FACH}$. Therefore, 3G communications energy consumption is formulated as:

$$E_{3G} = E_{request} + E_{result} + \left(time_{FACH}^{request} + time_{FACH}\right)401 + (2 \times time_{DCH})570 \quad (9)$$

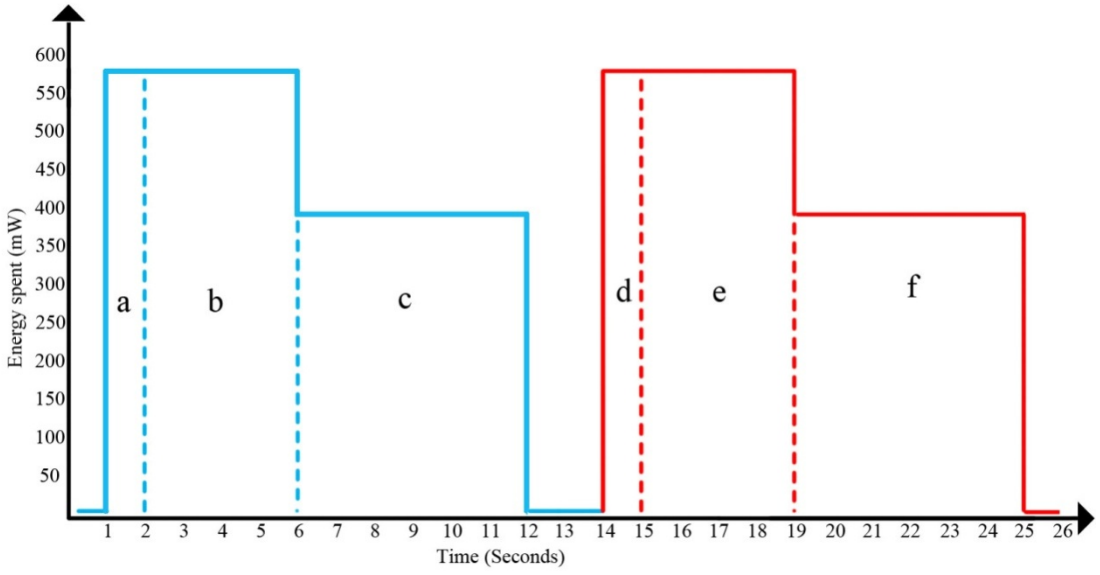


Figure 4.9 - Model 5

Figure 4.9 shows the energy used by 3G communications when $size_{request}$ is bigger than $size_{threshold}$ and a route result is sent back after $time_{FACH}$ seconds. So the 3G interface enters the IDLE state, and (a) is the energy consumed to send the request file to CloudMade, (b) is the energy consumption of the tail energy for the DCH state, (c) is the energy consumption of the tail energy for the FACH state and then the 3G interface enters the IDLE state, (d) is the energy consumed to receive the route result from CloudMade, (e) is the energy consumption of the tail energy for the DCH state, and (f) is the energy consumption of tail energy for the FACH state and then the cellular interface enters the IDLE state.

We can explain the energy consumption of 3G communications as shown in Figure 4.9 using formula(1) where $E_{request}$ is represented as the energy consumed to transfer the request file or (a). $time_{DCH}^{request}$ is $time_{DCH}$ and $time_{FACH}^{request}$ is $time_{FACH}$ because the route result is transferred back to cloud-based Osmand after $time_{FACH}$ seconds. Thus, E_{idle} is considered in the 3G communications energy consumption. Cloud-based Osmand receives the route result from CloudMade, which consumes E_2. $time_{DCH}^{result}$ is $time_{DCH}$ and $time_{FACH}^{result}$ is $time_{FACH}$. Therefore, 3G communications energy consumption is expressed as:

$$E_{3G} = E_{request} + E_{result} + (2 \times time_{FACH})401 + (2 \times time_{DCH})570 + E_{idle} \quad (11)$$

Then we calculated the 3G communications energy consumption in the five case studies in Table 4.6.

Table 4.6 - 3G communications energy consumption for five models

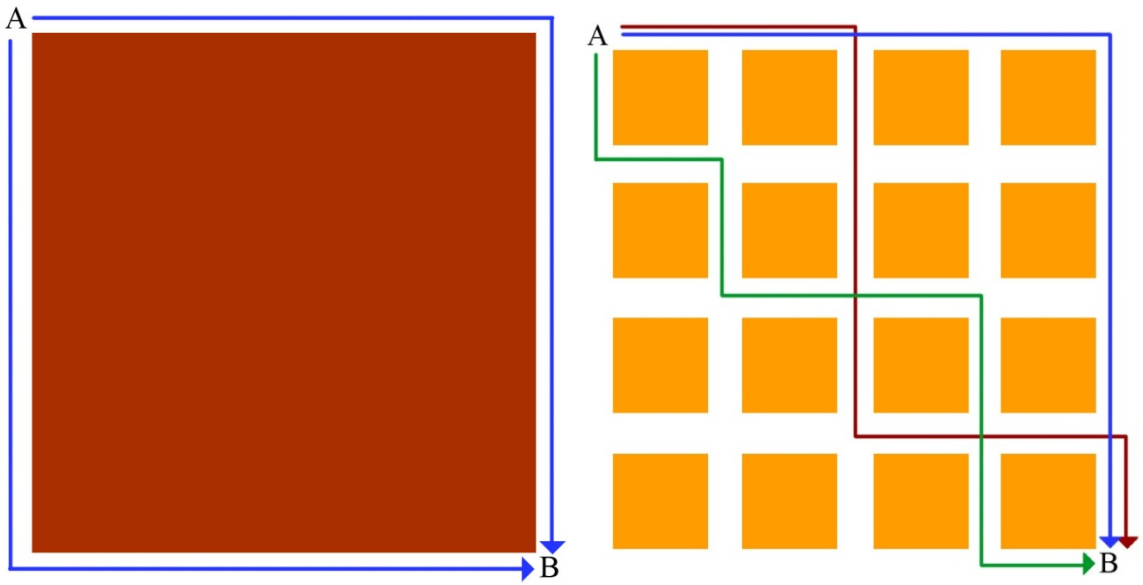
Models	**Size of request file**		**Time when receiving route result**			**Energy spent (mW)**
	$< size_{threshold}$	$> size_{threshold}$	**Within $time_{FACH}$ after sending a request file**	**Within $time_{DCH}$ after sending a request file**	**After being idle**	
1	✓		✓			5657 - 8063mW

2	✓				✓	8063mW + E_{idle}
3		✓		✓		5826 – 8106mW
4		✓	✓			8507 – 10512mW
5		✓			✓	10512mW + E_{idle}

Based on the results from the measurement study, the 3G interface consumes at least 5600mW to search for a route result. It uses only 971mW to transmit a request file and route result over the network and the rest of the 3G communications energy consumption is tail energy.

4.5 Designing a recommendation prototype

The goal of this study was to make recommendations for improving the energy efficiency of cloud-based Osmand and a local device. Based on the empirical measurement and the analytical characterization on energy consumption in the previous section in this chapter, our first recommendation for cloud-based Osmand is that a route result should be allowed to transfer over the network if it is within $time_{DCH}$ to minimise tail energy. In this study, we set $time_{DCH}$ as 4 seconds. This recommendation is based on the finding from our empirical measurement and the study [11]. However, if $time_{DCH}$ is over before the route result is transferred, cloud-based Osmand should switch to offline mode to continue searching for the route result and reduce 3G communication energy consumption. If cloud-based Osmand searches the route result on a local device, it needs to consider the complexity of maps because from our empirical measurement, we found that the complexity of maps affects the CPU energy consumption. Thus, if the map between a starting point and destination point is complex, the CPU computation by the local device is heavier compared with a simple map. This is explained by Figure 4.10.



(a) (b)

Figure 4.10 - The complexity of maps

Figure 4.10 models the complexity issue of maps. Assume A is a starting point and B is a destination point. The map in Figure 4.10(a) is simple because the map only contains a block between A and B. Thus, routes between A and B can be only two ways and if using path scoring of the A* algorithm [12-13] to find the shortest route from A to B, the computation is not heavy. On the contrary, a map such as Figure 4.10(b) is complex due to the 16 blocks between A and B. This makes CPU in a local device calculates more, which causes more CPU energy consumption, to find the shortest route because there are so many possible routes from A to B that are necessary to calculate. Hence our second recommendation is to allow a route result to be transferred over the network if a map is complex. This recommendation would reduce the CPU energy consumption of cloud-based Osmand in cases of complex maps.

4.5.1 EESManager

An Energy Efficient Scheduling Manager or EESManager was developed using the two recommendations mentioned above. We designed EESManager for Osmand and it works using an online feature, a map and a delay of route results. The conditions below are checked and a determination is made whether switching to an offline feature or not uses

less energy consumption. The parameters used in device side algorithm and cloud side algorithm are listed in Table 4.7.

Table 4.7 - EESManager algorithm parameters

Symbol	Meaning
$time_{DCH}$	Tail time of the DCH state
$size_{threshold}$	The size of the state transition threshold
$size_{request}$	The sizes of a request file
$size_{result}$	The sizes of a route result file
$node_{map}$	Number of nodes on a map between a starting point and a destination point
$node_{complex}$	The minimum number of nodes on a map between a starting point and a destination point that make a map complex

Device side algorithm

1: **if** Osmand uses online feature **Then**

2: Get $size_{request}$ and $node_{map}$

3: **if** $size_{request} < size_{threshold}$ **then**

4: put the 3G cellular interface in the DCH state

5: **end if**

6: Send the request file and $node_{map}$ to CloudMade

7: **if** Osmand does not receive a route result within $time_{DCH}$ seconds

8: **if** $node_{map} < node_{complex}$ **then**

9: Osmand use offline feature

10: **else**

11: Start receiving the route result from CloudMade

12: **end if**

13: **end if**

Cloud side algorithm

1: **if** receive a request file and $node_{map}$ from Osmand **Then**

2: compute a route result

3: **if** the route result is not sent within $time_{DCH}$ seconds **then**

4: **if** $node_{map} < node_{complex}$ **then**

5: stop processing

6: **else**

7: send the route result

12: **end if**

13: **end if**

The device side algorithm of EESManagerstarts by a user using Osmand. The device side algorithm of EESManager checks which version of Osmand is being used. If Osmand uses an online feature, the EESManager algorithm is used. If not, Osmand uses an available offline feature.

The next phase of the device side algorithm of EESManager is used to obtain parameter values to perform cloud-based Osmand. A request file and a route result assign $size_{request}$ and $size_{result}$. From the empirical measurement, we found that $size_{result}$ is always bigger than the state transition threshold, $size_{request}$ is dependent on coordinates of the starting point and destination point, and an intermediate list. We assign $node_{map}$ as the numbers of nodes on a map between a starting point and a destination point. If $node_{map}$ is greater than $node_{complex}$, it means that the map is complex. If $node_{map}$ is less than $node_{complex}$, the map is not complicated. The request file is sent to CloudMade and if $size_{request}$ is smaller than the state transition threshold, EESManager switches the

3G cellular interface to the DCH state. If the route result is not completed within $time_{DCH}$ seconds, there are two options from which to choose: First if $node_{map}$ is less than $node_{complex}$, Osmand switches to an offline feature. Second if $node_{map}$ is bigger than $node_{complex}$, Osmand continues to receive the route result from CloudMade.

At the same time, the cloud side algorithm of EESManager begins with CloudMade receiving a request file from Osmand. Then the cloud side algorithm of EESManager lets CloudMade find a route result. If the route result is completed within $time_{DCH}$ seconds, CloudMade sends the route result back to Osmand. If not, EESManager chooses from two options: if $node_{map}$ is less than $node_{complex}$, CloudMade stops processing. If $node_{map}$ is greater than $node_{complex}$, CloudMade sends the route result back to Osmand.

Chapter 5 Evaluation and results

In this chapter, we report on an extensive simulation evaluation of the EESManager by using Android Emulator [82]. The proposed EESManager will be validated and evaluated through different navigation scenarios for its performance on improving energy efficiency for local devices. We compare the results of original cloud-based Osmand and cloud-based Osmand with the EESManagerin different navigation scenarios to determine whether the EESManager can improve energy efficiency of cloud-based Osmand and save energy consumption for local devices. The simulation evaluation has confirmed that the better energy efficiency of cloud-based Osmand with EESManager by comparing to the original cloud-based Osmand and EESManager also helps a local device reducing CPU energy consumption.

The remainder of this chapter is organised as follows. Firstly we will explain a simulation environment which is Android emulator. Also, simulation configuration, the emulator control of network and the geo location provider emulation are presented to understand how Android emulator was configured. Next, the simulation methodology of this evaluation will be described. The existing technologies which were used in the evaluation will be also defined. Finally results of the evaluation on the Android emulator and summary of the results will be shown.

5.1 Simulation environment

In Chapter 5, simulation methodology is used to evaluate EESManager because testing the hypothesis with simulation can save us time and money. So, we used an Android emulator [82] for the simulation environment because the Android emulator has similar Android smart phone environment compared to other simulation tools.

The Android emulator was used to simulate and evaluate EESManager, which is a virtual mobile device emulator that can run on computers. Android developers can use the emulator without using a real Android smart phone device to test, develop and prototype Android applications. When a developer runs a testing application on the emulator, the

service of the Android platform can be used by the Android emulator to invoke other applications or use Android features such as playing audio and video, or storing and retrieving data. Debug capabilities are also included in the emulator, so developers can simulate the interruptions of an application, and study the effects of network latency and package lost on the data network. Therefore, all the hardware and software features of a typical mobile device are included in the Android emulator.

5.1.1 Simulation configuration

To set up the simulation environment, here is a list of requirements; 1) Java 1.6, 2) Eclipse IDE with built-in ADT (Android Develop Tools), 3) Android NDK, 4) Osmand and 5) PowerTutor. First to run the Android emulator, Java 1.6 and Eclipse IDE with built-in ADT [72] are needed to be installed in a target computer. The Android emulator is provided by the Android Software Development Kit (Android SDK) [83, 84] which is in Eclipse IDE with built-in ADT. Next, Osmand is required the Android SDK platforms of Android 1.6 (API 4) and Android 2.2 (API 8) [85] which can be downloaded in Android SDK Manager as seen in Figure 5.1. Then, the Android emulator also requires an Android Virtual Device which is a configuration of Android emulator that can let a developer models and creates a configuration of an actual device included hardware and software options [86]. In this simulation, we used Android Virtual Devices Manager (AVD Manager) [87] to create an Android virtual device as seen in Figure 5.2.

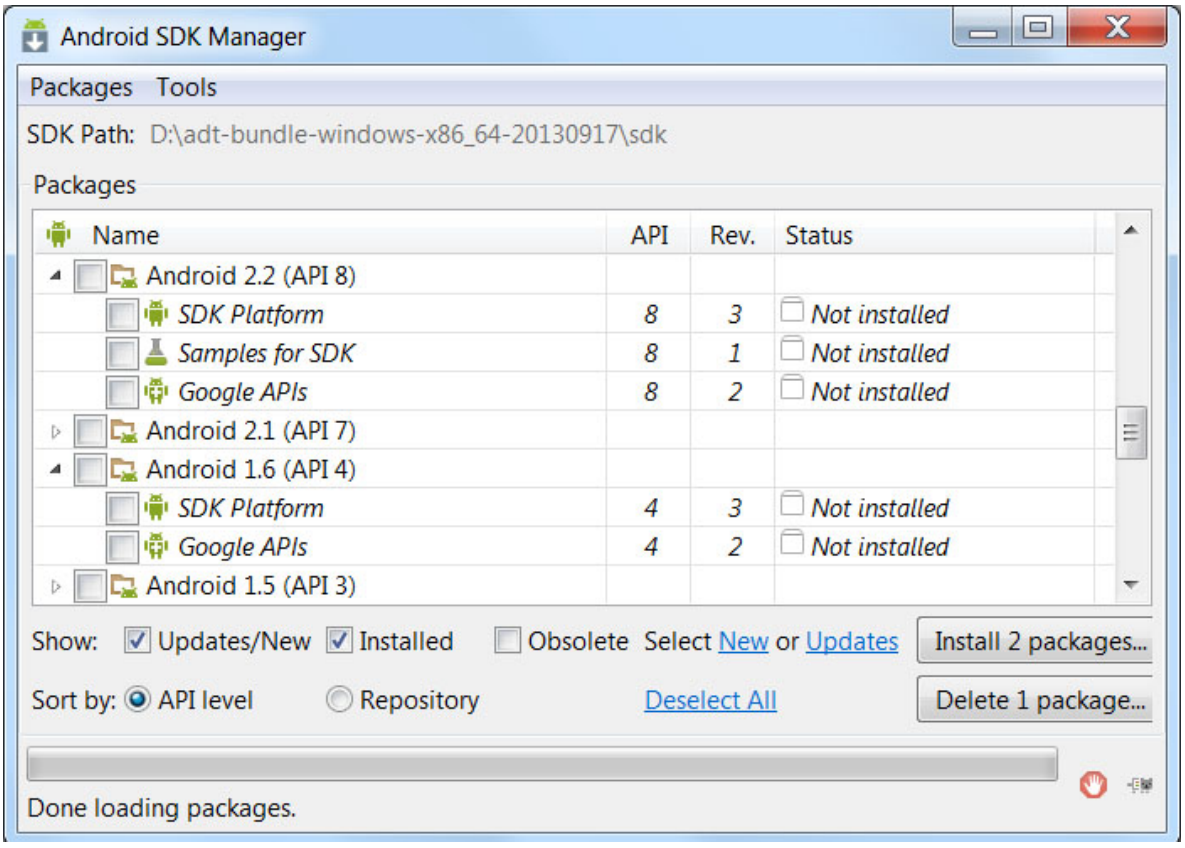


Figure 5.1 - The screenshot of Android SDK Manager

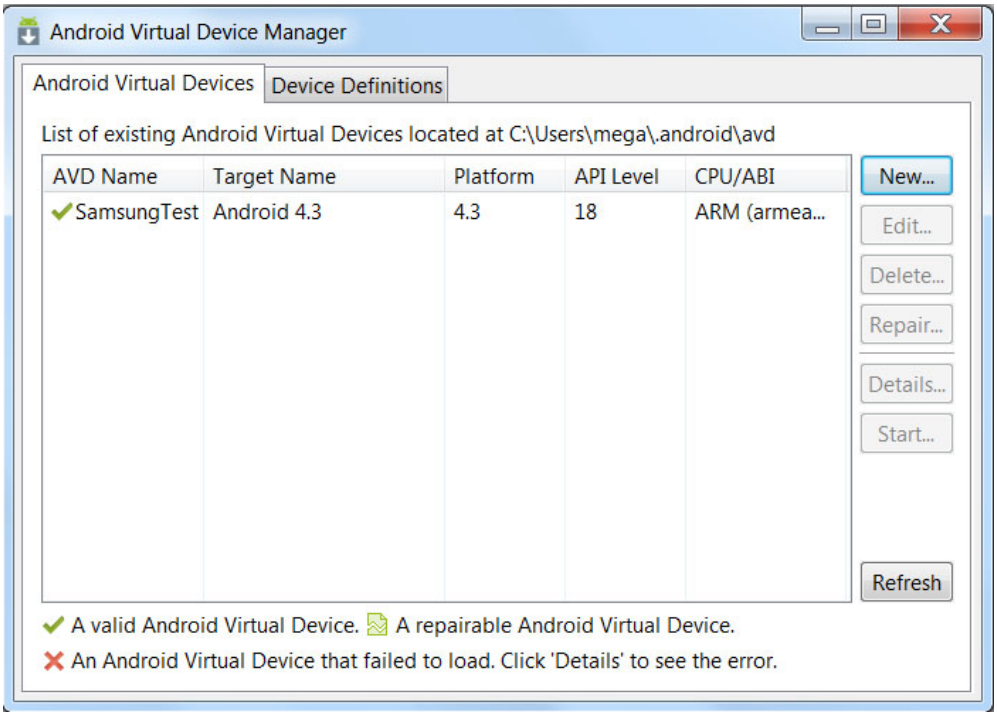


Figure 5.2 - AVD Manager

In this evaluation, the simulation was run on Ubuntu 12 Operating System because Osmand can be downloaded easily using repositories in Ubuntu. Ubuntu is based on the Linux kernel in desktop environment and it is under a free and open source software license [88] and Ubuntu contains the majority of software packages which are under a free software license. Repositories are referred as software archives that store Ubuntu programs and repositories provide a high level of security while repositories install new software onto Ubuntu using network connection [89]. In this thesis, Osmand was downloaded using command line below

```
$ repoinit-u git://github.com/osmandapp/OsmAnd-manifest.git
$ repo sync
```

Figure 5.3 - Android emulator with Osmand icon on the screen

Then, Osmand was imported to Eclipse's workspace and Osmand can run through the Android emulator as seen in Figure 5.3. Moreover, PowerTutor is needed to be installed in the Android emulator to collect the energy consumptions of hardware components of

Osmands. PowerTutor can be found and downloaded at [90]. To install PowerTutor, we opened the terminal and typed “directory” where platform-tools folder in Android SDK tools is and used the command line below

```
adb install PowerTutor.apk
```

Finally, the simulation which has Osmand and PowerTutor installed in the Android emulator was run to evaluate EESManager. In addition, the Android emulator provides navigation and control keys. The mapping between the emulator keys and the keys of a keyboard is summarised in Table 5.1.

Table 5.1 - The mapping of emulator keyboard (Adapted from [82])

Emulated Device Key	**Keyboard Key**
Home	HOME
Menu (left softkey)	F2 *or* Page-up button
Star (right softkey)	Shift-F2 *or* Page Down
Back	ESC
Call/dial button	F3
Hang-up/end call button	F4
Search	F5
Power button	F7
Audio volume up button	KEYPAD_PLUS, Ctrl-F5
Audio volume down button	KEYPAD_MINUS, Ctrl-F6
Camera button	Ctrl-KEYPAD_5, Ctrl-F3
Switch to previous layout orientation (for example, portrait, landscape)	KEYPAD_7, Ctrl-F11
Switch to next layout orientation (for example, portrait, landscape)	KEYPAD_9, Ctrl-F12
Toggle cell networking on/off	F8

Toggle code profiling	F9 (only with *-trace* startup option)
Toggle full screen mode	Alt-Enter
Toggle trackball mode	F6
Enter trackball mode temporarily (while key is pressed)	Delete
DPad left/up/right/down	KEYPAD_4/8/6/2
DPadcenter click	KEYPAD_5
Onion alpha increase/decrease	KEYPAD_MULTIPLY(*) / KEYPAD_DIVIDE(/)

5.1.2 Using the emulator console

The Android emulator supports varied options and developers can use those options to control behaviour or appearance of an application when the emulator is launched. A control console is provided in each emulator and developers can connect and use the console for simulation. Before using the emulator control console, the path needs to be set up. Firstly, a developer opens the terminal and types “directory” where Android SDK tools is as it can be seen from Figure 5.4.

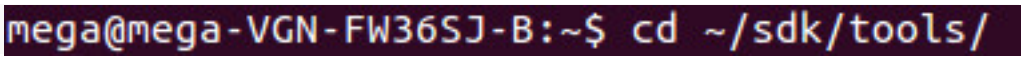


Figure 5.4 - Setting up path for the emulator control console

Next, the console of the target emulator is needed to connect as you can see from Figure 5.5.

Figure 5.5 - Command for connecting the target console

Then, the terminal shows a result as can be seen from Figure 5.6.

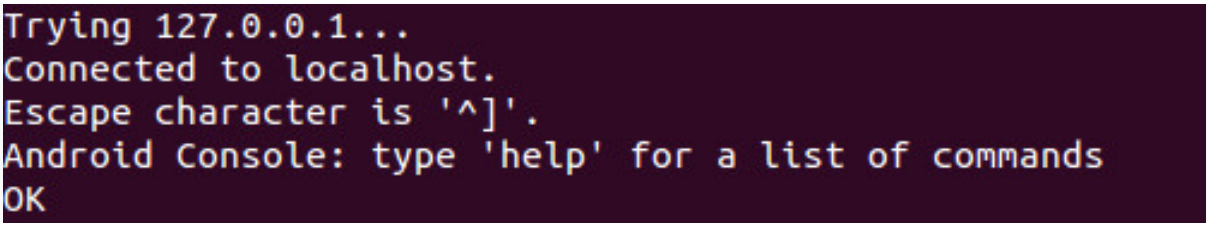


Figure 5.6 - The result of connecting the target console

5.1.1.1 Network emulation

We needed to simulate the network connection between Osmand based on a mobile device and CloudMade, so network conditions affect directly the environment simulation. The Android emulator console can be used for checking the network status, delay and speed, and simulating network conditions [82]**.**

A developer can simulate various network latency levels by using the Android emulator. Therefore, an application can be tested in more typical condition environments. A latency level or range can be set at emulator start up or the developer uses the emulator console to change the latency level while the emulator is running the application [82]**.**

In this simulation, we set the latency level at the emulator control console after the console shows the result of connecting the console as can be seen in Figure 5.6. The *netdelay* command with a supported <*delay*> value from Table 5.2 is used when the emulator is running a test application and the developer connects to the emulator control console. Here is command that we used:

```
network delay umts
```

Table 5.2 - The format of network <delay> (numbers are in milliseconds), Adapted from [82]

Value	Description	Comments
Gprs	GPRS	(min 150, max 550)
Edge	EDGE/EGPRS	(min 80, max 400)
Umts	UMTS/3G	(min 35, max 200)

None	No latency	(min 0, max 0)
<num>	Emulate an exact latency (milliseconds)	
<min>:<max>	Emulate a specified latency range (min, max milliseconds)	

5.1.1.2 Geo Location Provider Emulation

We also needed to set the geo location in Osmand to run the simulation environment. The Android emulation provides the geographic location by using the emulator control. The easiest way to put a GPS coordinate location is, firstly, to click Window -> Open Perspective -> Other. The Open Perspective Window will be shown as you can see in Figure 5.7 and then DDMS is selected. The DDMS window is shown.

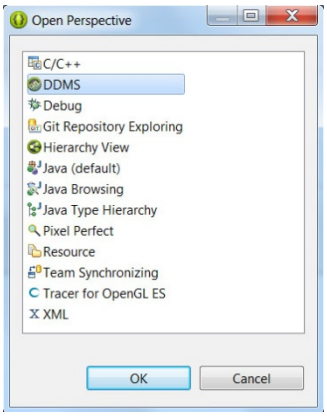


Figure 5.7 - Open Perspective

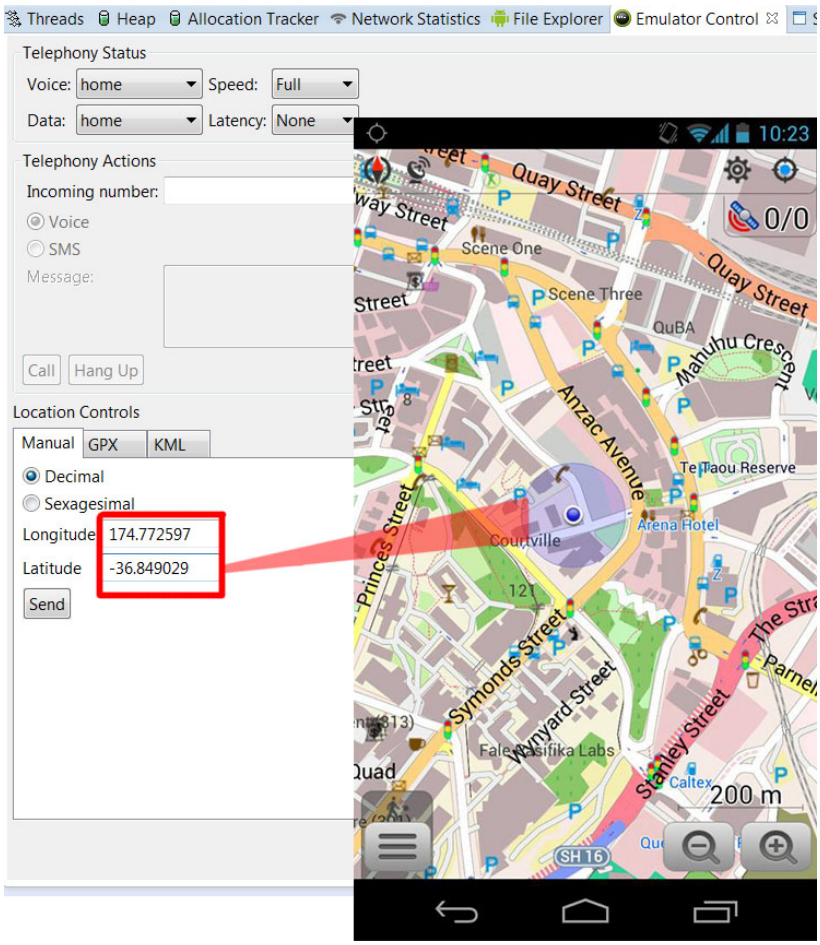


Figure 5.8 - Location Controls in Emulator Control Tab

From DDMS, we choose the Emulator Control Tab and scroll down to Location Controls. Longitude and latitude must be put in the boxes marked with a red square in Figure 5.8. When the developer clicks the send button, Osmand will show the longitude and latitude from Location Controls on the screen as seen in Figure 5.8.

The Android emulator helps developers to test an application without using an actual Android smart phone device. However, the emulator still has limitations and it does not support some functions used in the actual device. Below is a list of the functional limitations [82].

The Android emulator has no support for:

- Calling and answering real phone calls
- USB connections

- Device-attached headphones
- Ascertainment of network connected state
- Ascertainment of battery charge level and AC charging state
- Ascertainment of SD card insert/eject
- Bluetooth

5.2 Simulation methodology

We only focus on improving cloud-based Osmand, so evaluation will be the comparison of original cloud-based Osmand and cloud-based Osmand with EESManager. Original cloud-based Osmand and cloud-based Osmand with EESManager were used to run simulations on the Android emulator 10 times for each chosen route for each version of Osmand. We chose our chosen routes under conditions of distances and complexity of maps. Hence, there are four routes chosen for simulations which were:

- Map scenario 1 which is from Wynyard Quarter to the Hilton Hotel which is represented a short distance and simple map,
- Map scenario 2 which is from Orakei Yacht Sales to Mission Bay Beach which is represented a long distance and simple map,
- Map scenario 3 which is from the Statesman Apartment to Auckland's Central Library which is represented a short distance and complex map,
- And map scenario 4 which is from Andrew Simms Chrysler Jeep Dodge to Greenlane Clinical Centre which is represented a long distance and complex map.

To collect results of these evaluations, the PowerTutor [14], which was installed in the Android emulator, was used to measure CPU and 3G energy consumption, while Logcat [68, 69], which is part of the Eclipse IDE with built-in Android Developer Tools (ADT) was used to record Osmand's activities. The matching data from both PowerTutor and Logcat will show the results of EESManager's performance.

Moreover, in the simulation we make some assumptions as seen in Table 5.3: we define $time_{DCH}$ as four seconds and a map is treated as a complex map if $node_{complex}$ contains more than 25 nodes between a starting point and a destination point.

The complexity of maps is explained by an example from Figure 4.7 in Chapter 4. Moreover, $size_{threshold}$ is assumed as 270 bytes.

Table 5.3 - Parameters of simulation

Symbol	Meaning	assumptions
$time_{DCH}$	Tail time of the DCH state	4 second[13, 18-20]
$node_{complex}$	The minimum number of nodes on a map between a starting point and a destination point that make a map complex	25 nodes
$size_{threshold}$	The size of the state transition threshold	270 bytes [13, 18-20]

5.3 The results

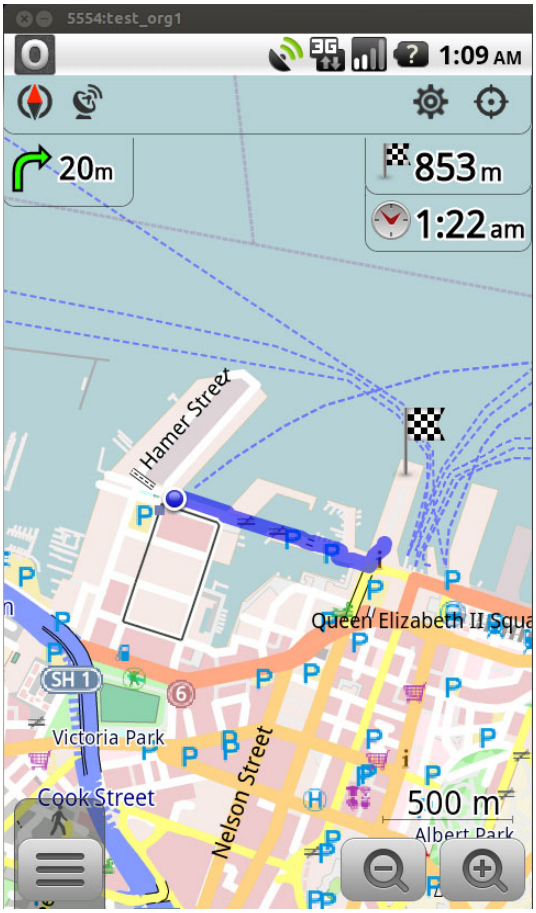


Figure 5.9 - Map scenario 1 from Wynyard Quarter to the Hilton Hotel

As shown in Figure 5.9, the route in this scenario starts at Wynyard Quarter and ends at the Hilton Hotel. In this study a complex map is considered as a map that contains more than 25 nodes between a starting point and a destination point. The map in Figure 5.9 contains 21 nodes. EESManager will enforce the use of the local device if a route result is not found and sent to the local device within four seconds. However, if a route result is sent back within four seconds, EESManager will let the local device receive the route result. The starting point in this simulation is determined to be 0.853kilometres away from the destination. Thus, this scenario presents the map which is simple and has short distance. The analysis was based on the collected data for this simulation is shown in Figure 5.10.

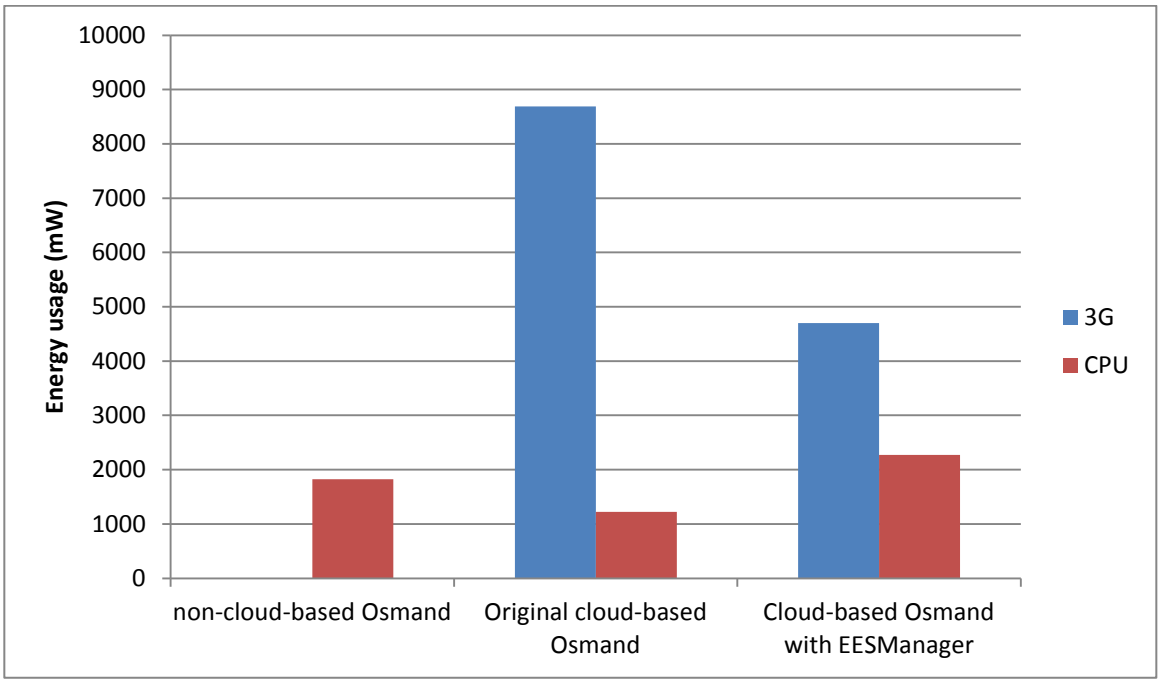


Figure 5.10 - Results of testing with a starting point of Wynyard Quarter and end point of the Hilton Hotel

As shown in Figure5.10 are the results of testing between non-cloud-based Osmand, original cloud-based Osmand and cloud-based Osmand with EESManager. Results show that on average 3G communication energy consumption can be decreased approximately 4000mW by using EESManager compared with the original cloud-based Osmand, but it causes cloud-based Osmand with EESManager to consume more CPU energy than the non-cloud-based Osmand and the original cloud-based Osmand by 445mW for non-cloud-based Osmand and just over 1050mW for original cloud-based Osmand. As a result, EESManager

can save energy consumption nearly by 30% compared with original cloud-based Osmand in this scenario.

Figure 5.11 - Map scenario 2 from Orakei Yacht Sales to Mission Bay Beach

As shown in Figure 5.11, a simulation route is run from Orakei Yacht Sales as a starting point to Mission Bay Beach as a destination point. The map in Figure 5.11 has eight nodes. If CloudMade cannot find a route result within four seconds, a local device is used by EESManager instead of CloudMade. On the other hand, the local device will receive the route result if CloudMade sends the result back within four seconds. The route result is determined to be 2.93kilometers in length. Hence, this scenario is represented a simple map which has long distance than the map in Figure 5.9. Results from collected data for this simulation are shown in Figure 5.12.

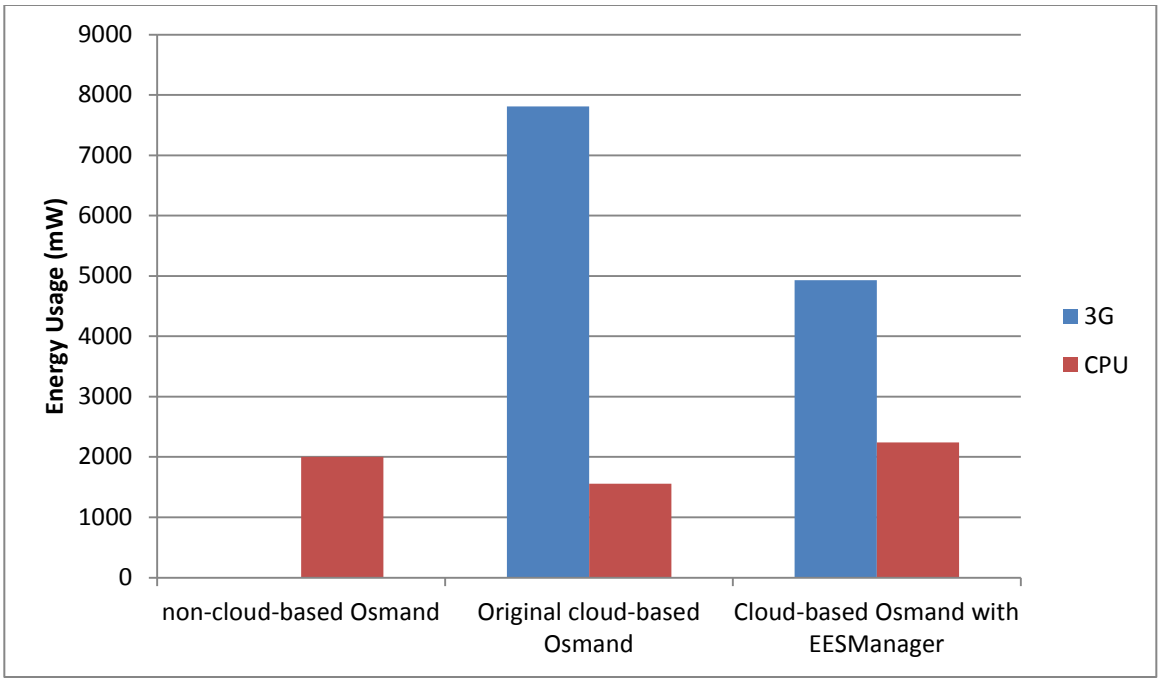


Figure 5.12 - Results of testing with a starting point of Orakei Yacht Sales and a destination point of Mission Bay Beach

As shown in Figure 5.12 are the results of testing between non-cloud-based Osmand, original cloud-based Osmand and cloud-based Osmand with EESManager. Results show that, on average, 3G communication energy consumption can be reduced nearly 3000mW by EESManager but it causes more CPU energy consumption in cloud-based Osmand with EESManager compared to non-cloud-based Osmand, which is by 238mW, and the original cloud-based Osmand which is by just over 680mW. In consequence, EESManager can save overall energy consumption by 23% compared with original cloud-based Osmand in the scenario.

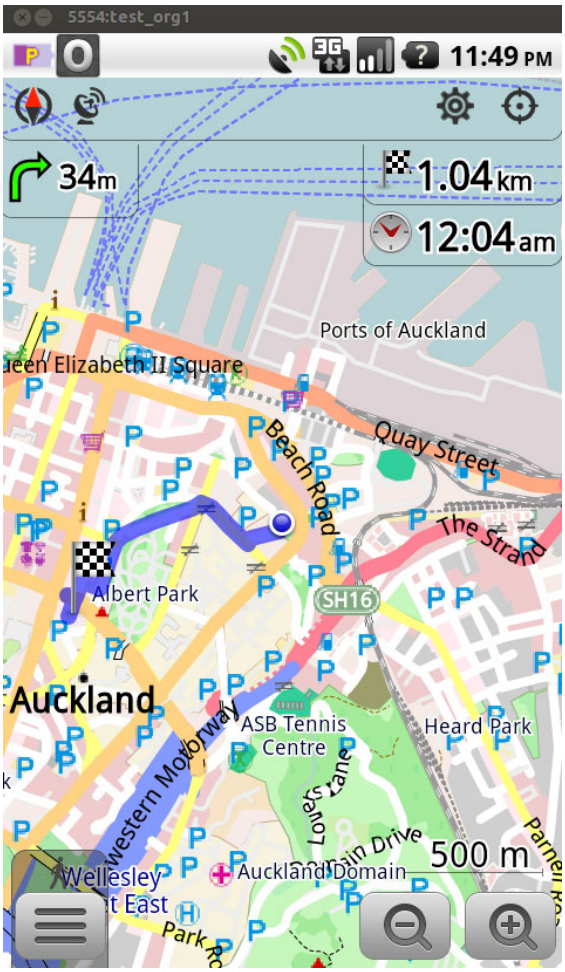


Figure 5.13 - Map scenario 3 from the Statesman Apartments to Auckland's Central Library

As seen in Figure 5.13, a simulation begins at The Statesman Apartments and ends at Auckland's Central Library. The route result is determined to be 1.04kilometers in length. The map in Figure 5.13 is a complex map because it contains 59 nodes between the starting point and the destination point. Thus, this scenario presents the map which is complex and has short distance. If a route result is sent back within four seconds, EESManager will let the local device receive the route result. However, if a route result is not found and sent to the local device within four seconds, EESManager will allows CloudMade to finish searching for the route result to save CPU energy consumption of the local device because the computation capabilities of CloudMade are more powerful than the local device's and the process of finding a route result would be heavy for a local device. Results from the collected data for this simulation are shown in Figure 5.14.

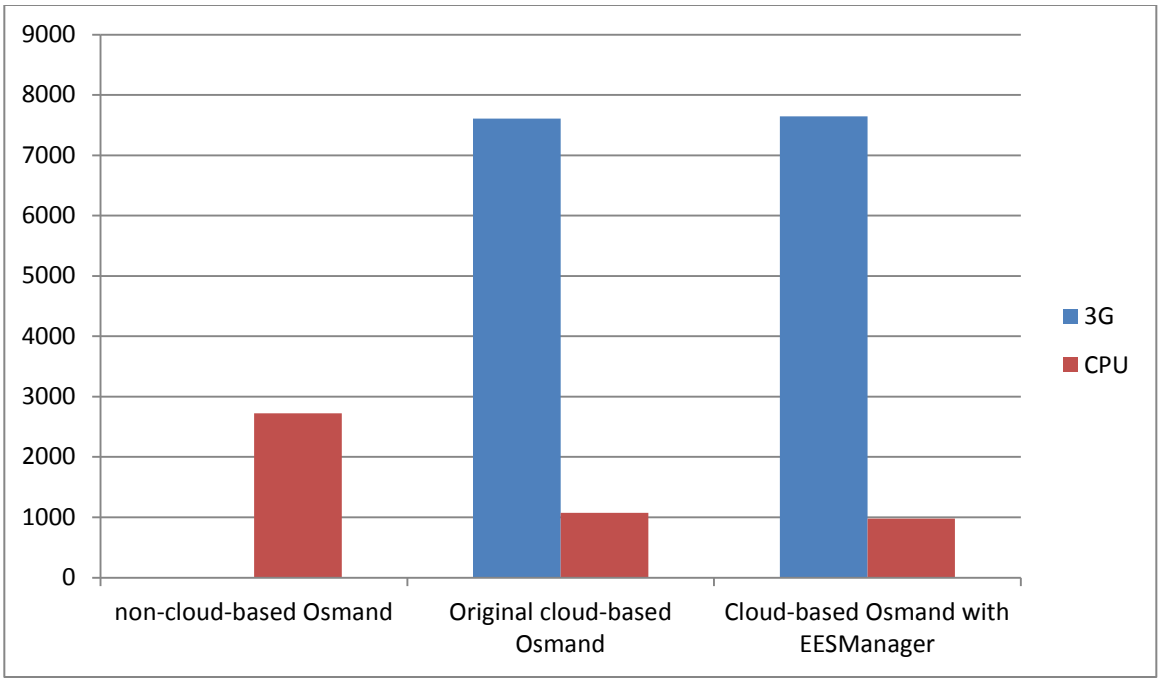


Figure 5.14 - Results of testing with a starting point the Statesman Apartments to Auckland's Central Library

As shown in Figure 5.14 are the results of testing between non-cloud-based Osmand, original cloud-based Osmand and cloud-based Osmand with EESManager. Results indicated that both original cloud-based Osmand and cloud-based Osmand with EESManager consume similar amounts of CPU and 3G communication energy. However, EESManager determines to let CloudMade finish its task and the local device receives the route result if the route result is not sent back within four seconds. EESManager causes less CPU energy consumption compared with non-cloud-based Osmand by 1750mW. As a result, EESManager can save CPU energy consumption approximately 64% compared with non-cloud-based Osmand in this scenario.

Figure 5.15 - Map scenario 4 from Andrew Simms Chrysler Jeep Dodge to Greenlane Clinical Centre

As shown in Figure 5.15, a simulation is used to run from a starting point at Andrew Simms Chrysler Jeep Dodge to Auckland's Central Library. The route length is determined to be 3.38kilometers.The map in Figure 5.15 has 48 nodes on the map so this means that the map is complex. Therefore, this scenario represents the map which is complex and has longer distance compared with the map in Figure 5.13. Due to the map containing more than 25 nodes, the local device would require high processing power to search for a route result if a route result is not found and sent to the local device within four seconds. Therefore, to reduce the energy consumption of CPU in the local device EESManager allows CloudMade to finish searching for the route result because of the powerful computation ability of CloudMade. Results from the collected data for this simulation are shown in Figure 5.16.

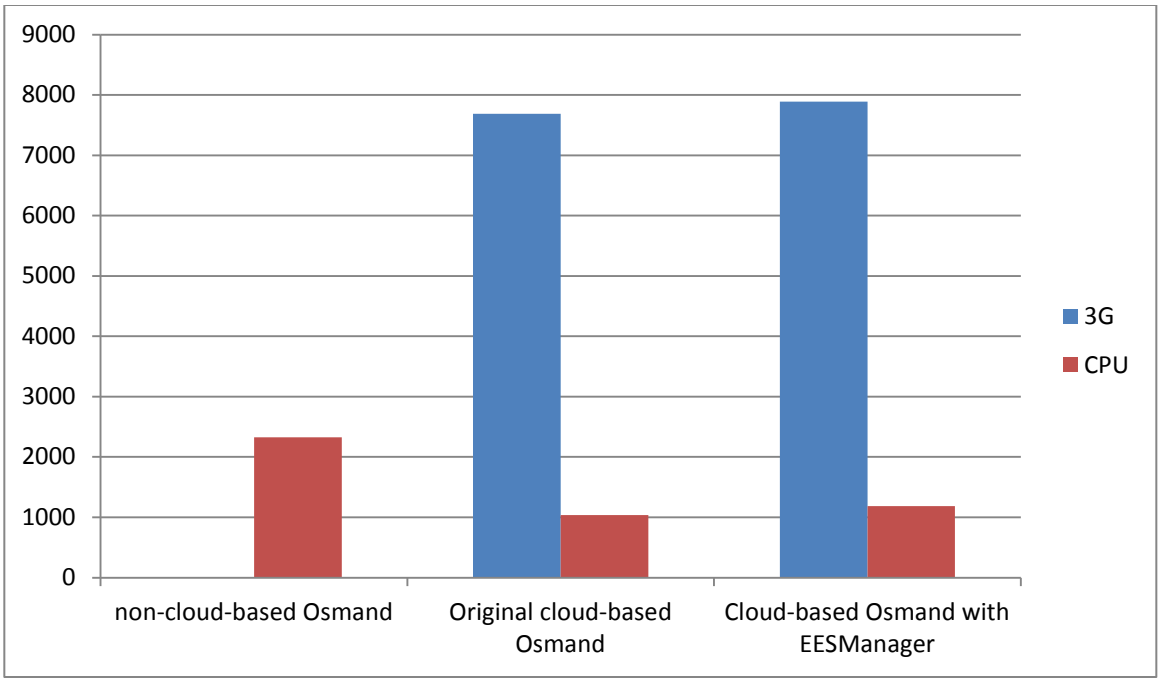


Figure 5.16 - Results of testing with a starting point at Andrew Simms Chrysler Jeep Dodge and the destination point at Greenlane Clinical Centre.

As seen in Figure 5.16are the results of testing between non-cloud-based Osmand, original cloud-based Osmand and cloud-based Osmand with EESManager. Results showed that both the original cloud-based Osmand and cloud-based Osmand with EESManager have similar CPU and 3G communication energy consumptions. On the other hand, CloudMade are allowed to finish searching for a route result and send the result back to the local device if the route result is not sent back within four seconds. EESManager makes CPU of cloud-based Osmand consumes less energy consumption compared with non-cloud-based by about 1142mW. Consequently, EESManager can help the local device save CPU energy consumption by 49% compared to non-cloud-based Osmand in this scenario.

From all results in Figure 5.9 – Figure 5.16, we summarise the energy consumption of three versions of Osmand as can be seen in Table 5.4.

Table 5.4 - The CPU and 3G communication energy consumption from the simulated results

Map scenario	Energy consumptions of non-cloud-based Osmand		Energy consumptions of original cloud-based Osmand		Energy consumption of cloud-based Osmand with EESManager	
	CPU	3G	CPU	3G	CPU	3G
Map Scenario 1	1824.7mW	0mW	1220mW	8691.3mW	2270.3mW	4700mW
Map Scenario 2	2003.5mW	0mW	1555.8mW	7808.1mW	2242.1mW	4931mW
Map Scenario 3	2726mW	0mW	1076.5mW	7607.6mW	976.5mW	7647.7mW
Map Scenario 4	2325.3mW	0mW	1037.7mW	7686.8mW	1182.9mW	7888.3mW

5.4 Summary

On one hand, as shown in Figures 5.9 – 5.12, EESManager uses an offline version of Osmand if CloudMade could not finish finding route results within four seconds because the maps between the starting points and the destination points are simple which contain less than 25 nodes. As a result, EESManager can reduce 3G communication energy consumption of cloud-based Osmand in simulation testing compared with original cloud-based Osmand. However, EESManager caused more CPU energy consumption when compared with non-cloud-based Osmand and the original cloud-based Osmand. Overall, EESManager can reduce by 26.6% of the total of energy consumption of the original cloud-based Osmand.

On the other hand, as seen in Figures 5.13-5.16, EESManager allows CloudMade to finish searching for the route after four seconds to save CPU energy consumption if Osmand did not receive route results within four seconds and these maps are complex because the process of finding a route result on a complex map and it would require heavy

computation for a local device. As a result, EESManager does not have any different results compared with the original cloud-based Osmand in both simulation testing results shown in Figures 5.14 and 5.16 but the results of cloud-based Osmand with EESManager compared with the result of non-cloud-based Osmand show that cloud-based Osmand with EESManager consumes less CPU energy than non-cloud-based Osmand. As a result, EESManager can help the local device reduce CPU energy consumption by 57% compared to non-cloud-based Osmand.

Consequently, EESManager can improve cloud-based Osmand's energy efficiency and a local device when 1) route results are not sent to Osmand within tail time and maps are simple and 2) route results are sent to Osmand within tail time. Furthermore, there is no study on energy efficient of mobile navigation applications in the public before and EESManager was designed for only Osmand. Hence, we cannot compare the performance of this thesis and the different previous research.

Chapter 6 Conclusions and Future work

The purpose of this thesis was to improve the energy efficiency of a selected mobile application which is Osmand and a local device for this study. We were striving to discover what characteristics of the selected mobile application cause in term of energy used and to contribute new knowledge and solutions to build a cloud-based prototype by using energy efficient recommendations.

6.1 Conclusions

Inspired from the impact of cloud-based applications on the battery life of mobile devices, we have chosen Osmand as our selected mobile application, and identifies cloud-based Osmand's characteristics of energy used compared with non-cloud based Osmand. Also, we try to tackle these problems by EESManager.

The first conclusion we have found that the energy consumptions of GPS and LCD for both non-cloud based Osmand and cloud-based Osmand are the same. For CPU, we found that non-cloud-based Osmand consumes more CPU energy consumption for the calculating route result activity than other activities in non-cloud-based Osmand and the complexity of maps relates the CPU energy consumption of the calculating route result activity. In cloud-based Osmand, we found that cloud-based Osmand uses a large amount of energy consumption in CPU to create and display application's events on the mobile screen compared with non-cloud based Osmand. This is because programming technique and source code of Osmand influence the CPU energy consumption of cloud-based Osmand. Furthermore, the network connection is required for cloud-based Osmand to send and receive files between the clouds. In 3G communication in cloud-based Osmand, tail energy was a factor that causes energy wasted. For this study, we focused on developing and improving 3G communication energy consumption and tail energy usage. We also focused on reducing CPU energy consumption by determining what feature, which is online or offline features, would be suited for each map scenario.

The second conclusion we have developed EESManager based on knowledge from the earlier measurement study and the consideration of the factors which are limiting the tail energy and the complex of maps. We have implemented EESManager in cloud-based Osmand running on the Android emulator. We chose 4 routes to run simulation ten times in non-cloud-based Osmand, the original cloud-based Osmand and cloud-based Osmand with EESManager for each route. The results from simulations show that EESManager can improve on the energy efficiency of cloud-based Osmand and the local device in two cases. The first case is when CloudMade can find route results and send the results back to the local device within the tail time. The second case is when CloudMade cannot find route results and send them back within the tail time and also map scenario are needed to be simple.

6.2 Future work

Considering the work covered in this thesis within a constraint in the limited time period and the development of the energy measurement technologies for applications, it would be useful to highlight some future areas to be further investigated.

In this thesis EESManager was designed for Osmand, while the ideas of EESManager algorithm should be used and implemented for other type of mobile applications such as applications which have heavy process and applications which involve file transmission to gain benefits of energy saved in the future work. Furthermore, an application running on the Android was only focused in this study. Hence, applications in other mobile platform such as iOS and Windows Mobile should be studied on energy efficiency similar to this thesis.

The sizes of data which were involved in the network connection of this thesis are small compared with other types of data such as video files, images and so on. The study of the large mobile data affecting the energy efficiency on the local device needs to be address in the future work. Also, the complexity of the A* algorithm should be studied in-depth and analysed to find the relationship between the complexity of the A* algorithm and energy consumption.

References

[1] *Battery life concerns mobile users*. Available: http://edition.cnn.com/2005/TECH/ptech/09/22/phone.study/

[2] J. Newman. *iPhone 4S users satisfied with phone, but not its battery life*. Available: http://www.pcworld.com/article/245272/iphone_4s_users_satisfied_with_phone_but_not_its_battery_life.html

[3] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?," *Computer*, vol. 43, pp. 51-56, 2010.

[4] V. Namboodiri and T. Ghose, "To cloud or not to cloud: A mobile device perspective on energy consumption of applications," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2012 IEEE International Symposium on a*, 2012, pp. 1-9.

[5] K. Kumar, "Application-based energy efficient mobile and server computing," 2011.

[6] P. Asrani, "Mobile Cloud Computing," *International Journal of Engineering and Advanced Technology (IJEAT)*, vol. 2, 2013.

[7] D. Huang, "Mobile cloud computing," *IEEE COMSOC Multimedia Communications Technical Committee (MMTC) E-Letter*, vol. 6, pp. 27-31, 2011.

[8] F. Özlü. (2012). *Mobile Cloud Computing*. Available: http://www.slideshare.net/Fatihzl/mobile-cloud-computing-16045326

[9] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 4-4.

[10] K. Pentikousis, "In search of energy-efficient mobile networking," *Communications Magazine, IEEE*, vol. 48, pp. 95-103, 2010.

[11] P. Simoens, F. De Turck, B. Dhoedt, and P. Demeester, "Remote display solutions for mobile cloud computing," *Computer*, vol. 44, pp. 46-53, 2011.

[12] S. Robinson, "Cellphone energy gap: desperately seeking solutions, Strategy Analytics," ed, 2009.

[13] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, *et al.*, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2010, pp. 105-114.

[14] (2009). *PowerTutor*. Available: http://ziyang.eecs.umich.edu/projects/powertutor/

[15] msg555. *PowerTutor / src / edu / umich / PowerTutor / components / CPU.java*. Available: https://github.com/msg555/PowerTutor/blob/master/src/edu/umich/PowerTutor/components/CPU.java

[16] msg555. *PowerTutor / src / edu / umich / PowerTutor / components / GPS.java*. Available: https://github.com/msg555/PowerTutor/blob/master/src/edu/umich/PowerTutor/components/GPS.java

[17] msg555. *PowerTutor / src / edu / umich / PowerTutor / components / LCD.java*. Available: https://github.com/msg555/PowerTutor/blob/master/src/edu/umich/PowerTutor/components/LCD.java

[18] msg555. *PowerTutor / src / edu / umich / PowerTutor / components / Threeg.java*. Available: https://github.com/msg555/PowerTutor/blob/master/src/edu/umich/PowerTutor/components/Threeg.java

[19] msg555. *PowerTutor / src / edu / umich / PowerTutor / phone / DreamConstants.java*. Available: https://github.com/msg555/PowerTutor/blob/master/src/edu/umich/PowerTutor/phone/DreamConstants.java

[20] msg555. *PowerTutor / src / edu / umich / PowerTutor / phone / DreamPowerCalculator.java*. Available: https://github.com/msg555/PowerTutor/blob/master/src/edu/umich/PowerTutor/phone/DreamPowerCalculator.java

[21] L. Guan, X. Ke, M. Song, and J. Song, "A survey of research on mobile cloud computing," in *Computer and Information Science (ICIS), 2011 IEEE/ACIS 10th International Conference on*, 2011, pp. 387-392.

[22] J. H. Christensen, "Using RESTful web-services and cloud computing to create next generation mobile applications," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, 2009, pp. 627-634.

[23] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: enabling mobile phones as interfaces to cloud applications," in *Middleware 2009*, ed: Springer, 2009, pp. 83-102.

[24] X. Luo, "From augmented reality to augmented computing: a look at cloud-mobile convergence," in *Ubiquitous Virtual Reality, 2009. ISUVR'09. International Symposium on*, 2009, pp. 29-32.

[25] L. Lei, Z. Zhong, K. Zheng, J. Chen, and H. Meng, "Challenges on wireless heterogeneous networks for mobile cloud computing," *Wireless Communications, IEEE*, vol. 20, 2013.

[26] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, *et al.*, "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, 2010, pp. 49-62.

[27] S. H. Hung, C. S. Shih, J. P. Shieh, C. P. Lee, and Y. H. Huang, "Executing mobile applications on the cloud: framework and issues," *Computers & Mathematics with Applications*, vol. 63, pp. 573-587, 2012.

[28] P. J. Havinga and G. J. Smit, "Energy-efficient wireless networking for multimedia applications," *Wireless communications and mobile computing*, vol. 1, pp. 165-184, 2001.

[29] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning, "Saving portable computer battery power through remote process execution," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 2, pp. 19-26, 1998.

[30] B. G. Chun and P. Maniatis, "Augmented Smartphone Applications Through Clone Cloud Execution," in *HotOS*, 2009, pp. 8-11.
[31] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, 2011.
[32] S. Chetan, G. Kumar, K. Dinesh, K. Mathew, and M. Abhimanyu, "Cloud computing for mobile world," *National Institute of Technology, Calicut*, 2010.
[33] L. Jordan and P. Greyling, *Practical android projects*: Apress, 2011.
[34] *Android Architecture - The key concepts of Android OS*. Available: http://www.android-app-market.com/android-architecture.html
[35] *Android Architecture*. Available: http://www.tutorialspoint.com/android/android_architecture.htm
[36] *Android developers*. Available: http://developer.android.com/guide/basics/what-is-android.html
[37] *Application fundamentals*. Available: http://developer.android.com/guide/topics/fundamentals.html
[38] C. Haseman, *Android Essentials*: apress, 2008.
[39] I. Ivan and A. Zamfiroiu, "Quality Analysis of Mobile Applications," *Informatica Economica*, vol. 15, pp. 136-152, 2011.
[40] OpenMobster. (2010). *Programming concepts*. Available: https://code.google.com/p/openmobster/wiki/Concepts
[41] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 29-42.
[42] A. Sharma, V. Navda, R. Ramjee, V. N. Padmanabhan, and E. M. Belding, "Cool-Tether: energy efficient on-the-fly wifi hot-spots using mobile phones," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, 2009, pp. 109-120.
[43] G. Ananthanarayanan, V. N. Padmanabhan, L. Ravindranath, and C. A. Thekkath, "Combine: leveraging the power of wireless peers through collaborative downloading," in *Proceedings of the 5th international conference on Mobile systems, applications and services*, 2007, pp. 286-298.
[44] B. G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 301-314.
[45] C. Xian, Y.-H. Lu, and Z. Li, "Adaptive computation offloading for energy conservation on battery-powered systems," in *Parallel and Distributed Systems, 2007 International Conference on*, 2007, pp. 1-8.
[46] A. Bhojan, A. L. Akhihebbal, M. C. Chan, and R. K. Balan, "Arivu: Making networked mobile games green," *Mobile Networks and Applications*, vol. 17, pp. 21-28, 2012.
[47] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, 2009, pp. 280-293.

[48] E. Lagerspetz and S. Tarkoma, "Mobile search and the cloud: The benefits of offloading," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*, 2011, pp. 117-122.
[49] *Dessy at SourceForge.net*. Available: http://dessy.sourceforge.net/
[50] *Android, the world's most popular mobile platform*. Available: http://developer.android.com/about/index.html
[51] (2013). *OsmAnd Maps & Navigation*. Available: https://play.google.com/store/apps/details?id=net.osmand&feature=nav_result#?t=W251bGwsMSwyLDNd
[52] (2013). *Osmand*. Available: https://code.google.com/p/osmand/
[53] *OsmAnd Maps & Navigation*. Available: http://osmand.net/
[54] *OpenStreetMap The Free Wiki World Map*. Available: http://www.openstreetmap.org/#map=5/51.500/-0.100
[55] *OpenStreetMap Foundation*. Available: http://wiki.osmfoundation.org/wiki/Main_Page
[56] "Introduction to A* from Amit's thoughts on pathfinding."
[57] P. Lester. (2005). *A* pathfinding for beginners*. Available: http://www.policyalmanac.org/games/aStarTutorial.htm
[58] *CloudMade*. Available: http://routes.cloudmade.com/f656e49977444ca9b4e4bce21ed79987/api/0.3/
[59] F. G. Sayward, "Experimental design methodologies in software science," *Information Processing & Management*, vol. 20, pp. 223-227, 1984.
[60] Explorable.com. (2011). *Experimental research*. Available: http://explorable.com/experimental-research
[61] California State University. *Experimental Research*. Available: http://psych.csufresno.edu/psy144/Content/Design/Types/experimental.html
[62] The National Advanced Driving Simulator. (2010). *Why simulation?* Available: http://www.nads-sc.uiowa.edu/simulators.php
[63] G. S. Fishman, *Concepts and methods in discrete event digital simulation*: Wiley New York, 1973.
[64] W. D. Kelton and A. M. Law, *Simulation modeling and analysis*: McGraw Hill Boston, MA, 2000.
[65] G. A. Mihram, "Simulation methodology," *Theory and Decision*, vol. 7, pp. 67-94, 1976/02/01 1976.
[66] E. H. Page Jr, "Simulation modeling methodology: principles and etiology of decision support," Virginia Polytechnic Institute and State University, 1994.
[67] R. E. Shannon, *Systems simulation: the art and science* vol. 975: Prentice-Hall Englewood Cliffs, NJ, 1975.
[68] *Logcat*. Available: http://developer.android.com/tools/help/logcat.html
[69] *Reading and writing logs*. Available: http://developer.android.com/tools/debugging/debugging-log.html#startingLogcat
[70] T. E. Foundation. *About the Eclipse Foundation*. Available: http://www.eclipse.org/org/
[71] M. Erickson. (2001). *What is Eclipse, and how do I use it?* Available: http://www.ibm.com/developerworks/opensource/library/os-eclipse/index.html

[72] *Get the Android SDK*. Available: http://developer.android.com/sdk/index.html#download
[73] *Android Developer Tools*. Available: http://developer.android.com/tools/help/adt.html
[74] *Using DDMS*. Available: http://developer.android.com/tools/debugging/ddms.html
[75] Monsoon Solutions Inc. (2008). *Power Monitor*. Available: http://www.msoon.com/LabEquipment/PowerMonitor/
[76] Monsoon Solutions Inc. *Mobile Device Power Monitor Manual*. Available: http://msoon.github.io/powermonitor/PowerTool/doc/Power%20Monitor%20Manual.pdf
[77] *Traceview*. Available: http://developer.android.com/tools/help/traceview.html
[78] *Profiling with Traceview and dmtracedump*. Available: http://developer.android.com/tools/debugging/debugging-tracing.html
[79] GSMARENA. *Samsung galaxy nexus I9250*. Available: http://www.gsmarena.com/samsung_galaxy_nexus_i9250-4219.php
[80] *Processes and threads*. Available: http://developer.android.com/guide/components/processes-and-threads.html
[81] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, 1982, pp. 120-126.
[82] *Using the emulator*. Available: http://developer.android.com/tools/devices/emulator.html
[83] *Tools help*. Available: http://developer.android.com/tools/help/index.html
[84] *Android Emulator*. Available: http://developer.android.com/tools/help/emulator.html
[85] *Install Development Environment*. Available: https://code.google.com/p/osmand/wiki/InstallDevelopmentEnvironment
[86] *Managing Virtual Devices*. Available: http://developer.android.com/tools/devices/index.html
[87] *Managing AVDs with AVD Manager*. Available: http://developer.android.com/tools/devices/managing-avds.html
[88] *The Ubuntu Story*. Available: http://www.ubuntu.com/about/about-ubuntu
[89] *What are Repositories?* Available: https://help.ubuntu.com/community/Repositories/Ubuntu
[90] *PowerTutor - 1.4*. Available: http://kerplunks.store.aptoide.com/app/market/edu.umich.PowerTutor/13/753313/PowerTutor

Glossary

ADT: Android Developer Tools

API: Application Programming Interface

AVD Manager: Android Virtual Devices Manager

DCH: the Dedicated Channel

DDMS: Dalvik Debug Monitor Server

DVM: Dalvik Virtual Machine

EDGE: Enhanced Data Rates for GSM Evolution

EESManager: Energy Efficient Scheduling Manager

FACH: the Forward Access Common Channels

GRPS: General Packet Radio Service

HetNet: Heterogeneous Network

IaaS: Infrastructure as a Service

IDE: Intefrated Development Environment

MAUI: the Mobile Assistance Using Infrastructure

MVC: Mobile Model-View-Controller

NDK: Native Development Kit

Osmand: Open Street Map Automated Navigation Directions

PaaS: Platform as a Service

QoS: Quality of Service

RC: Resource Controller

RDC: Resource Data Collector

REST: Representational State Transfer

RPC: Remote Procedure Call

SaaS: Software as a Service

SDK: Software Development Kit

Appendix A: Sample codes for EESManager's algorithms

Some Java scripts which were used in this thesis are shown below as examples of the scripts for the EESManager in this thesis.

The sample Java codes for the device side

Below are the codes for checking a version of Osmand which is used to find a route.

```
Public RouteCalculationResultcalculateRouteImpl(RouteCalculationParamsparams){
		long time;
		if (params.start != null&&params.end != null) {
			if(log.isInfoEnabled()){
				log.info("Start finding route from " + params.start +
" to " + params.end +" using " + params.type.getName()); //Finding a route
starts working from here
			}
			try {
				RouteCalculationResult res;
				if(params.gpxRoute != null&&
!params.gpxRoute.points.isEmpty()){
					res = calculateGpxRoute(params);
				} else if (params.type == RouteService.YOURS) {
					res = findYOURSRoute(params);
				} else if (params.type == RouteService.ORS) {
					res = findORSRoute(params);
				} else if (params.type == RouteService.OSMAND) {
					// non-cloud-based version is used
					res = findVectorMapsRoute(params);
				} else {
					// cloud-based version is used
					time = System.currentTimeMillis(); // get time
to use to calculate tail time
					int nodes = getNodes(params); //get numbers of
nodes from the map
					startEESManager(nodes,params,time);
				}
				if(log.isInfoEnabled() ){
					//the route is found
					log.info("Finding route contained " +
res.getImmutableLocations().size() + " points for " +
(System.currentTimeMillis() - time) + " ms");
				}
				return res;
```

```
                } catch (IOException e) {
                        log.error("Failed to find route ", e);
                } catch (ParserConfigurationException e) {
                        log.error("Failed to find route ", e);
                } catch (SAXException e) {
                        log.error("Failed to find route ", e);
                }
        }
        return new RouteCalculationResult(null);
    }
```

If cloud-based version is chosen to find the route, EESManager will start working. Below are the scripts of how EESManager works. If time of completing finding the route is over the tail time and the map is simple, EESManager will stop using CloudMade and switch into non-cloud-based version. If time of completing finding the route is over the tail time but the map is complex, EESManager will let CloudMade finish its work. However, if time of completing finding the route is before the tail time, EESManager will do nothing.

```
Public void startEESManager(intnodes,RouteCalculationParamsparams, long time) {
    RouteCalculationResult res;
    if (nodes<25){//check the complexity of the map
        //the map is simple
        while (!stopRequested) {
            res = findCloudMadeRoute(params,time);//send a request file
to CloudMade
    try {
    wait(4000);
          } catch (InterruptedException e) {
        System.out.println(e);//stop using cloud-based version
        res = findVectorMapsRoute(params);//switch to non-cloud-based
version
          }
       }
    }else{
        //the map is complex
        res = findCloudMadeRoute(params);
    }

}

Synchronized void requestStop() {
stopRequested = true;
if (thisThread != null)
thisThread.interrupt();
    }
```

The sample Java codes for the cloud side

Below are the codes for how EESManager works in CloudMade.

```
protected GPX findMapsRoute(LatLon start, LatLon end, long time){
	GPX result;
	if(time = null){
		result = findVectorMapsRoute(start,end);
	}else{
		while (!stopRequested) {
			result = findVectorMapsRoute(start,end);
	try {
	wait(4000);
		  } catch (InterruptedException e) {
		System.out.println(e);
		  }
	    }
	}
}

Synchronized void requestStop() {
stopRequested = true;
if (thisThread != null)
thisThread.interrupt();
}
```