ENERGY DISTRIBUTION THROUGH LIFETIME ESTIMATION AND SMARTPHONE USAGE PATTERNS

by

Karthik Palanivelu

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December 2014

To

My supervisor, mentor and well-wisher Prof. Srinivas Sampalli
who helped me learn by experience and build my career

and

My beloved parents Mr. Palanivelu Duraiswamy and Mrs.
Saralakumari Gajendran
for encouraging me all my life.

Table of Contents

List of Tables

List of Figures

Abstract

Today, smartphones have become an important part of our day to day life. Every day, practically every one of us wakes up in the morning hearing the sound of an alarm coming from our smartphone. First thing in the morning, we see the LED flashing on the phone and we have the urge to check all the notifications we got during the night. We unplug our phone from the charger. We read some notifications, we ignore some, we reply to some. Finally, we get out of the bed. We use it all day for everything: replying to emails, text messages and social networking. Finally, we go to bed, plug it back into the charger cable. We touch it until our eyelids close for the night's sleep. We have this routine usage of our smartphone almost every day. Every user uses his/her phone in his/her own way. Not all the people are the same. Not all the people have the same routine of life. But our smartphone is not really smart enough to learn our routine and adapt to us.

Our smartphone does not understand our needs. Either it just performs on its own or it performs only after we ask. There is a lot of areas that need automatic adaptation. Smartphones are the devices that perform tasks equal to a laptop or a desktop machine but with limited resources like battery, memory, screen size, etc. Unlike laptop users, smartphone users do not carry chargers with them all the time. Considering smartphone's limited battery as a serious concern, the battery would be the main area that needs automatic adaptation. Smartphones should know the user's recharge cycle and use the available energy efficiently by spending when it is in excess and reserving when it is in shortage.

Computation complexity has been doubling every couple of years. But, the battery capacity has been doubling every 10 years [6]. So, it is our responsibility to use the energy efficiently without a compromise in user experience. We propose ENDLESS (Energy Distribution Through Lifetime Estimation and Smartphone Usage Patterns) to determine if energy is to be saved for future use or if it can be consumed for present use according to the estimations of the next recharge time, applications and services that might be used in the near future and how much battery would be needed.

List of Abbrieviations Used

AC	Alternating Current
API	Application Programming Interface
APPNAME	Application Name
DT	Date Type (Weekend or Weekday)
EC	Energy Consumption
ELET	Energy Level at Recharge End Time
ELST	Energy Level at Recharge Start Time
GPS	Global Positioning System
IDE	Integrated Development Environment
JVM	Java Virtual Machine
Li-ion	Lithium Ion
LTE	Long-Term Evolution (4th Generation Mobile Communication System)
NiCd	Nickel Cadmium
NiMH	Nickel Metal Hydride
NoH	Number of Hits
NRE	Next Recharge Estimate
OS	Operating System
PACKNAME	Application Package Name
PAL	Potential Application Launch
PER	Probable Energy Required
RET	Recharge End Time
RM	Recharge Mode
RST	Recharge Start Time
SDK	Software Development Kit
SERVNNAME	Service Name
SET	Service End Time
SST	Service Start Time
TAPR	Time After Previous Recharge

Chapter 1

Introduction

Smartphones have become an essential part of our life in today's world. Most of us, especially college students, teenagers and young adults use our phone as if it is our sixth finger. Smartphones are mobile devices that are capable of performing both telephony and computing services. Telephony services include basic voice calls over GSM carriers and sending/receiving text messages. Computing services include Emailing, social networking, keeping up with appointments through calendar, browsing websites and much more.

1.1 Evolution of Smartphone

Smartphones have a long history dating back to 1973 when Martin Cooper, head of Motorola Research and Development, invented the first mobile device [6]. The device was huge and was not called as smartphone. However, it was able to serve the purpose of voice calling. The phone named DynaTAC was released to the public in 1983 by Motorola for USD 4000.

Since then, many software corporations started investing in smartphone research and made an enormous progress. In 1992, IBM started developing its first smartphone

Figure 1.1: IBM Simon Personal Communicator

Figure 1.2: BlackBerry Smartphones

"Simon Personal Communicator". They launched their device into the market on August 16, 1994. Simon combined the functionalities of both cellphone and PDA. In 1996, Nokia released its first smartphone called "Nokia Communicator". In the evolution of smartphone, BlackBerry phones have a remarkable role. BlackBerry Limited (formerly Research In Motion Limited) started making mobile devices in 1996. Their first device was Interactive Pagers with monochrome display, a thumb wheel for scrolling, and a thumb keyboard. The device was capable of emailing and minimal HTML browsing. Then they developed Java based monochrome display phones in 2002, color phones in 2003, consumer models in 2008. Throughout their journey in the smartphone industry, they concentrated wholly on enterprise businesses until the introduction of Apple's iPhone.

When Apple introduced its first smartphone in 2007, the definition of smartphone was changed. They designed it in such a way that the whole phone could be the same size as its display. They introduced their patented feature multi touch display in it. Due to the extra ordinary design and marketing of the iPhone, a lot of other competitors increased their speed and tried to make equivalent phones. One of the main players, Samsung, a South Korean giant, has now captured a huge share in the world smartphone market. When BlackBerry began running their own proprietary operating system BB 5.x, 6.x, 7.x and now BB10 and iPhone began running Apple's iOS, Samsung leaned over to Google's open source mobile operating system called Android.

Almost at the same time when Apple introduced its iPhone, Google announced their open source operating system Android. As they made it an open source, it

Figure 1.3: Smartphones in 2014

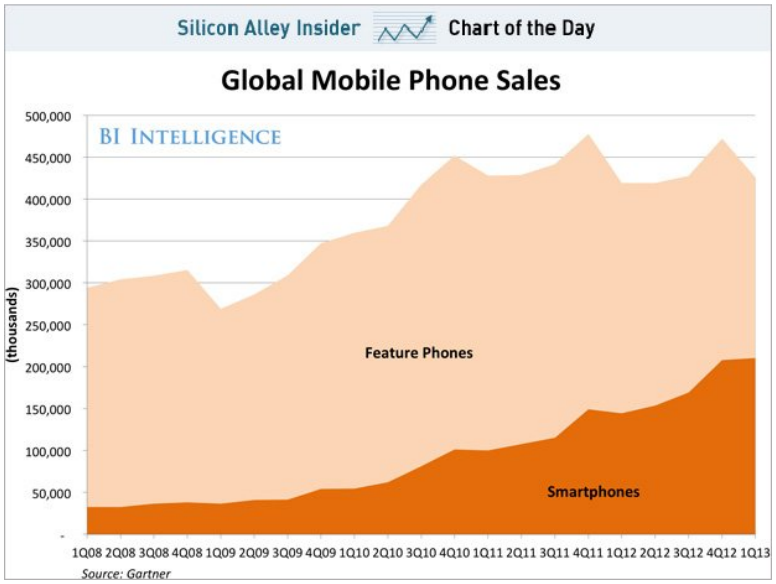


Figure 1.4: Global Sales Feature Phones vs Smartphones

encouraged numerous hardware manufacturers and desktop/laptop manufacturers to get into the smartphone business. Some of the companies that made use of the Android operating system were Samsung, LG, HTC and Sony.

When we look at the recent surveys on smartphone and feature phone sales, smartphone penetration into global market is growing year by year. A survey conducted by Gartner [23] shows that the percentage of smartphones has outnumbered the percentage of feature phones in 2013. Figure 1.4 clearly shows that the growth in the smartphone market share has been gradual since 2008 to 2010. But then it has taken a steep up since then. In 2013, out of 1.8 billion mobile devices sold, 968 million were smartphones.

1.2 Mobile Operating Systems

Smartphones being capable of performing most tasks that a desktop or laptop can do, it needs a well versed operating system to take care of various services like networking, application management, resource management, storage management, security, etc. Few mobile manufacturers have their own proprietary operating system. Apple iPhone runs their own iOS, BlackBerry runs their own BBOS and Microsoft phone runs their Windows Mobile OS. But most of other manufacturers like Samsung, LG, Sony and HTC opt for Google's open source operating system Android. These manufactures make devices with other platforms as well. For example, Samsung Galaxy ATIV runs Windows and HTC sells a device called HTC Windows Phone 8x. Earlier to Microsoft acquisition of Nokia mobile division, Nokia phones were running on the Symbian operating system.

1.2.1 Smartphone Market Share by Operating Systems

Since the introduction of Google's Android and Apple's iPhone in 2007, the evolution of the smartphone has taken a huge leap. Numerous statistical studies show that in recent years, the leadership of the smartphone market was taken over by Android from Symbian which held 50.3% of the world smartphone market share in 2009 [27]. As per the study by Canalys on Global Smarpthone sales in 2009, Symbian was at the top of the table with 50.3%, BlackBerry in second place with 20.9% followed by Apple's iPhone with 13.7%. Microsoft Windows Mobile with 9.0% and Android with 2.8% were the minor players.

Operating System	Global Market Share
Nokia Symbian	50.3%
BlackBerry	20.9%
Apple iPhone	13.7%
Microsoft Windows Mobile	9.0%
Google Android	2.8%
Other (E.g. Palm OS, Linux)	3.3%

Table 1.1: Canalys Study - Global Smartphone Sales in 2009

Within a year, in 2010, according to Canalys' study on Global Smartphone Sales [9], Google Android has taken the lead with a total market share of 32.9%, BlackBerry

Figure 1.5: Canalys Study - Global Smartphone Sales in 2009

with 14.4% market share, went to fourth position in 2010 from second position in 2009. Fortunately, Apple was able to keep its third position with 16.0%. All other platforms slipped a position. Nokia Symbian had dropped to second position with 30.6% and Microsoft was holding just 3.1% market share.

Operating System	Global Market Share
Google Android	32.9%
Nokia Symbian	30.6%
Apple iPhone	16.0%
BlackBerry	14.4%
Microsoft Windows Mobile	3.1%
Other (E.g. Palm OS, Linux)	3.0%

Table 1.2: Canalys Study - Global Smartphone Sales in 2010

If we compare Figure 1.5 and Figure 1.7, it is obvious that Android's growth was enormous. To date, Android is successful in dominating the world smartphone market. According to the study by Canalys on Global smartphone sales in 2013 [1], Google Android has captured 79% market share. Apple's iPhone has moved up from third to second position. But when we compare it with 2012 sales, it has lost 5% share in 2013. Since the acquisition of Nokia mobile division by Microsoft, Symbian platform is out of the table. Even with the Nokia hardware manufacturing unit, Microsoft is able to barely survive in this smartphone market. Throughout these five years, Microsoft Windows Mobile is consistent in its 3% share. In this whole market

Figure 1.6: Canalys Study - Global Smartphone Sales in 2010

shift, unfortunately BlackBerry has taken a deep decline from 20.9% in 2009 to 2% in 2013. BlackBerry is almost bottom of the table now.

Operating System	Global Market Share
Google Android	79%
Apple iPhone	15%
Microsoft Windows Mobile	3%
BlackBerry	2%
Others	1%

Table 1.3: Canalys Study - Global Smartphone Sales in 2013

As we look at the global market share by various platforms, it is clearly visible that Google Android has emerged as the most dominant platform. We shall mention a few reasons for Android being top of the table for the past four years. The main reason is its open source model that attracted a large number of mobile device manufacturers and a huge number of software application developers which in turn resulted in hundreds of thousands of mobile apps on its application store (Google Play Store). Because of the large number of Android mobile applications available, sales of Android mobile devices are growing. And due to the large user base, application developers are interested in making as many Android applications as possible in order to make higher profits. This has become a continuous cycle that keeps growing day by day.

Due to the various factors which include market share dominance, open source

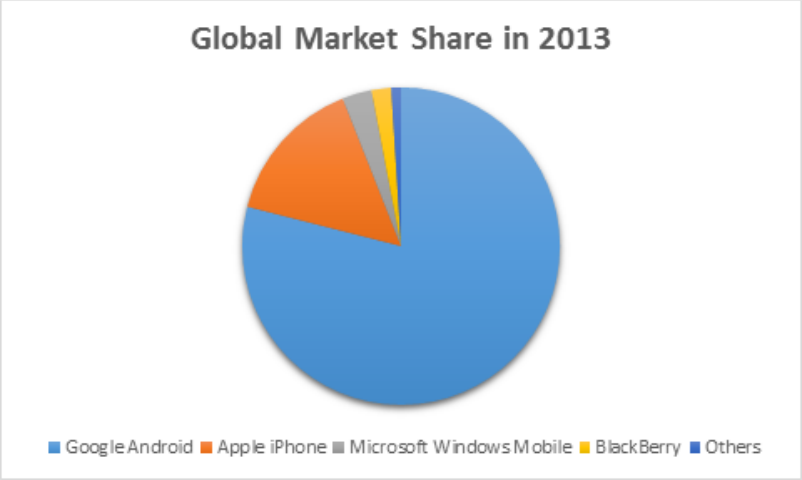


Figure 1.7: Canalys Study - Global Smartphone Sales in 2013

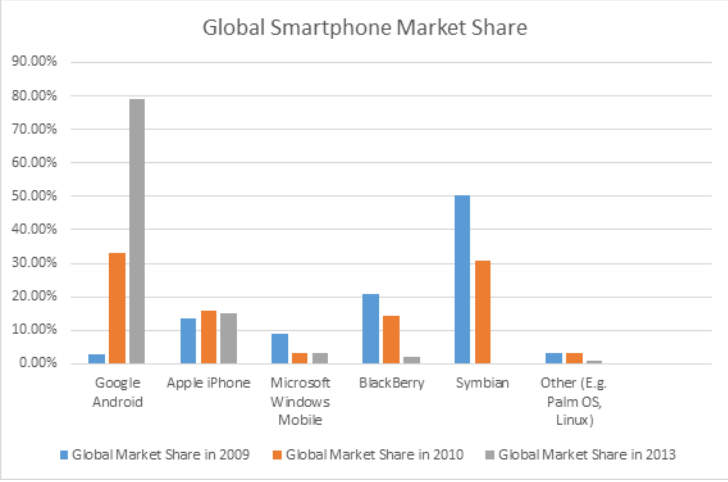


Figure 1.8: Canalys Study - Global Smartphone Sales

model, various supporting platforms (Linux, Mac, Windows) for application development, simple and powerful SDK, flexibility in experimenting various functionalities of an operating system, we chose Android platform to implement our methodology and tested on our Android powered mobile device, Google LG Nexus 4.

1.3 Batteries in Mobile Devices

We know that smartphones are limited in resources such as memory, processor speed, screen size and battery. If we consider the desktop computers, they are always connected to AC power and no need of any battery. But when we think about the capability of performing tasks by these smartphone devices, they almost perform as a mini computer. At the same time, smartphones offer convenience and portability. We can easily carry our smartphones in our pockets. We can use it wherever we want and whenever we want. Unlike laptop users, most smartphone users do not carry their mobile chargers with them [28]. Therefore, the smartphone users expect their mobile device to run at least for a day until they recharge their battery. But the real problem is that whether these batteries used in today's mobile devices evolved to be in line with the smartphone's functionality and features evolution. The answer is no. The smartphone features and functionalities are increasing rapidly. But we use the same Lithium-ion batteries which we used a few years back. According to Perucci et al. [25], Computation complexity doubles every two years whereas battery capacity doubles only every decade. We have not yet reached the technology with which we can access our mobile devices without any battery or the technology with which the battery recharges itself from the atmospheric energy. Until then, it becomes the responsibilities of software engineers to use the battery as efficiently and effectively as possible.

1.3.1 Common types of batteries

The battery is the blood of our mobile devices. As we know that one size does not fit all, the batteries as well are of different types according to the device needs and specifications. The common types of batteries used today in mobile devices are [21]:

1. Nickel Cadmium (NiCd)

2. Nickel Metal Hydride (NiMH)

3. Lithium Ion (Li-Ion)

4. Lithium Polymer (Li-Poly)

NiCd Batteries: These are of very old technology. They are cheap in cost and so it helps to bring down the overall selling price of the device. The disposal of Cadmium waste is actually an environmental issue. The other problem with these types of batteries is that recharging should only start after the complete drain. Otherwise, battery life would be an issue.

NiMH Batteries: These types of batteries are of slightly newer technology compared to NiCd. This beats NiCd batteries by addressing Cadmium's environmental issue. That is, no cadmium used in these batteries. Also, they offer higher capacity in consideration of its size and weight. These batteries should also be drained before the recharge. But the consequence of not doing so is lesser compared to NiCd.

Li-Ion Batteries: This is the most common type of batteries used today. Most smartphones carry this battery because of it longer battery life and lighter weight. But these batteries are expensive which affects the overall mobile device cost. Li-Ion batteries are prone to damage if they are plugged into the charger for a long period.

Li-Poly Batteries: This is a very new technology for mobile device batteries. So they are not very popular. These batteries deliver a longer battery life compared to all other and are very thin and light in weight. Only very few smartphones carry Li-Poly batteries. E.g. Lenovo K900.

1.3.2 Smartphone needs and Battery types

Mobile device manufacturers carefully decide which type of battery is used in their smartphone considering various factors. The important factors one should consider are

1. Battery Size and Weight

2. Battery Cost

3. Battery Capacity

Battery Size and Weight: Manufacturers design their mobile devices according to the market needs. Some devices may be thin and small, some may be thin but with bigger screens, some may have slide form factor and so on. Manufacturers go for certain type of batteries according to the form factor and size of their device.

Battery Capacity: Manufacturers make mobile devices for all types of users ranging from low end to high end. Low end phones feature only basic voice calls and text messaging whereas mid-level phones access the internet through browser applications and limited multimedia features. On the other hand, high level phones run heavy operating systems that are capable of high speed data connection with large number of applications and services running at the same time. So, according to the capability of the mobile device, battery capacity needs will vary.

Battery Cost: Mobile devices are sold in a wide variety of prices. The cheap mobile phones have lower specification like no Camera, no Wifi, etc. and costly devices have higher specification like 4G high speed data connection, bigger touch screen, etc. According to the selling price of devices, manufacturers decide the battery type.

The rest of this thesis is organised as follows.

The second chapter discusses the background information that is required to understand the rest of this thesis. It also gives an introduction of smartphone battery technology, battery consumption by various services and battery saving techniques.

In the third chapter, we explain our methodology with our Android application.

The following chapter gives the detailed information on implementation of our proposed methodology. We also discuss the Android application's features with its screenshots.

The fifth chapter reports our evaluation scheme and the test results.

We end the report with a conclusion and future work.

Chapter 2

Background and Literature Survey

2.1 User's Adaptation to Smartphone

Smartphone users use their smartphone for almost everything in their day-to-day life. Today's smartphone offers a lot more features including a high definition camera, high speed data connection, a huge number of applications, social networking, video conferencing, etc. But not all users use all the features their smartphones possess. The users who are addicted to social networking, have the apps like BBM, Facebook, Instagram, Twitter, etc. on all the time. The users who talk a lot, use their phone application more often. The users who travel frequently, have their data connection on. So, various users use their smartphone features and applications at various levels.

Smartphone user interface varies according to the platform (operating system) and the manufacturer. For example, Apple's iPhone has only one hardware button called "Home". So to switch between the applications, the user has to press the "Home" button to come out of the current application and tap on a new application to open it. In BlackBerry 10, the users can minimize their apps and all the minimized apps will be shown on the first page. So users can swipe up from any application to access this opened application list. But in Android, most recent smartphones have a soft button "Recent application". When users tap this button, a list of recent applications are shown and the users can switch easily. Another example is the "Back" button that is used to navigate within an application. BlackBerry and Android phones carry "Back" button at the bottom left corner which is available almost all the time. But in iPhone, the application developers should implement their back buttons for navigation and the button should be on top left corner. If we consider BlackBerry and iPhone, the phones are made by only themselves BlackBerry Limited and Apple Inc. respectively. But the Android phones are made by a lot of different manufacturers like Samsung, LG, Sony, Lenovo, etc. Each manufacturer alter the user interface to a small extent according to their design and their targeted users. So, the user interface

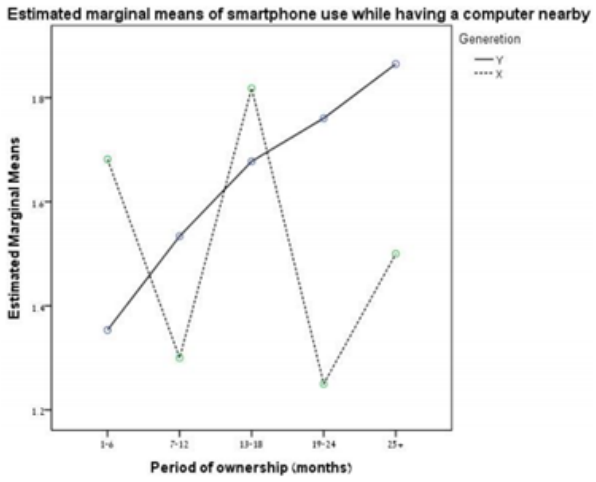


Figure 2.1: User's adaption of smartphone

on a smartphone varies according to the platforms and manufacturers.

Smartphones also vary by features and functionalities. Some phones have cameras and some do not. Some phones use 4G LTE data connection, some use HSPA+ and some still use 3G. Some phones are capable of video recording and some are not. So, the users adapt themselves to their smartphone features as much as possible to their capability.

The users who use the same phone for a long time, he/she gets used to its interface and tends to navigate through the screens and applications much faster. As the user knows what is what and what is where, he/she gets things done quicker and effectively. So, as the users use their phones more, they adapt to their devices.

The users moving from one phone to another find it difficult to use the phone at first. It takes time for them to become familiar with the user interface and features. According to Gafni et al. in "Generation Y versus Generation X: Differences in Smartphone Adaptation" [24], this adaptation differs also by age. The paper investigates the generation gap in this adaptation to various smartphones through the usage of mobile data services by various users. They find that the young generation users use the service less at first but with time, they start using more. On the other hand, with the older generation, there is not much difference in usage of services.

Figure 2.1 shows the tendency of generation Y and generation X to use a smartphone for Internet application when there is a personal computer available nearby, as a function of the smartphone's period of ownership. The authors name the younger

generation (aged 20-30) as generation Y and the older generation (aged 31-59) as generation X.

Overall, we can understand that there is a certain portion of users who adapt to their phones very easily as the period of ownership increases.

2.2 Smartphone's Adaptation to User

Smartphone users use their smartphone for anything and everything in their day-to-day life. But, it is very important to deeply investigate how well the smartphone has learned from its user. When a user buys any smartphone available in the market (Android, Blackberry, iPhone, etc.), it is very new without any information of the user. The phone comes with the factory operating system and preloaded apps like Text Messaging, Email, Phone, Calendar, etc. Once he/she starts using it, the device gets a few information from the user's input. For example, the user's name, email address, location, time zone, etc. As the time passes by, the user has become well adapted to his/her phone but the phone has not adapted to the user. The smartphone is not really smart enough to learn from its user about his/her needs and daily routine. For example, if a person walks from his/her home to work daily in the morning using the same route, he/she walks exactly in the same way taking the same turns even if he/she is thinking about something else. That is how the human brains work. But our smartphones do not work in this way. If the user goes to a library daily and turns his/her cell phone to silent mode as soon as he or she walks in, the device should actually remember this routine and do this automatically after a certain period. But even after a year or two or three, it will remain the same and the user has to continue to do it himself/herself.

As most of the users use their smartphones frequently for everything in their day-to-day life, it has become a vital concern if the device is learning from the user and makes his/her life easier and smoother. The smartphone industry is showing keen interest into this personalisation area to attract more and more customers. Due to various factors, although the corporate giants and academic researchers have put in a lot of effort into this, they have only achieved to a certain extent. The important factors are considered to be limited resources in smartphones which prohibits the heavy processing of data mining algorithms, user's privacy concerns which do not

(a) BlackBerry 10.2 (b) Android 4.2

Figure 2.2: User Adaptive Keyboard on Smartphones

let user data be processed remotely and so on. Considering these limitations, we identified a few smartphone functionalities that needs personalisation:

1. Keyboard
2. Location aware Profile change
3. Time based Profile change
4. Battery Consumption

2.2.1 Keyboard

The smartphone users are of different age groups and different professional levels. It is a well-known fact that most teenagers and students often chat a lot on messaging applications like BBM, WhatsApp, Viber, WeChat, etc. and most of the working professionals especially mid-level and senior-level managers use Email a great deal on their smartphones. No matter what language the users use and what characters they use to write in their language, they do tend to write the same sentences more often.

Today, smartphones remember the words users type and adds them to the dictionary and displays them as suggestions or to do auto correction. But most phones fail

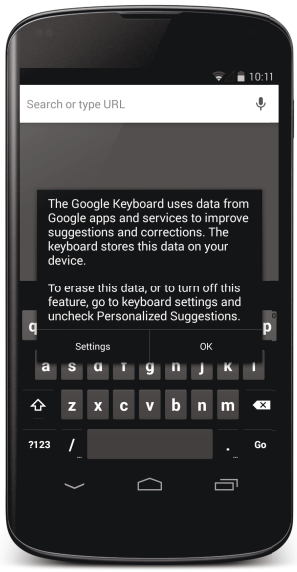
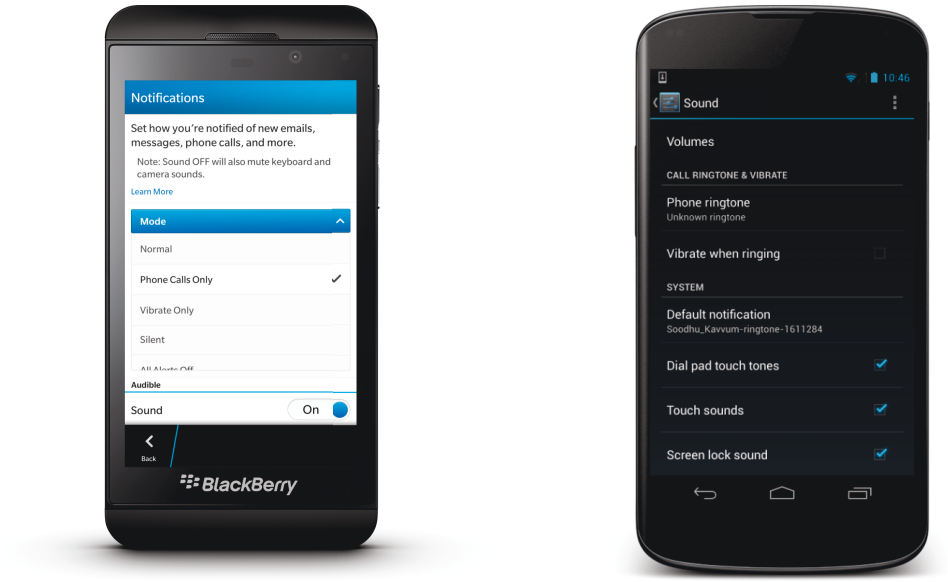


Figure 2.3: Android's Personalized Suggestions

to predict the next word because they did not learn the entire sentence. BlackBerry 10 platform has achieved the prediction of words in their keyboard to a great extent and are still working on the improvements in upcoming versions. Figure 2.2(a) shows a screen shot of Foursquare application on BlackBerry 10. When I entered 1065 (street number), based on my previous typing, the keyboard showed "BLAND" (street name) on top of the key "B". But the same application on Android did not show anything after I typed "1065". So as I continued typing "1065 B", the key board showed some suggestive words that starts with "B" on top of the keyboard: "By" and "But". Google also provides personalized suggestions based on the users input on google apps and services. Figure 2.3 shows the information dialog regarding Google's personalized suggestion. We hope for the benefit of smartphone users, BlackBerry improves their prediction mechanism and other platforms like Android and iOS start providing predictions instead of suggestions.

2.2.2 Location aware Profile change

It is a common rule in all libraries that the people should put their cell phone in silent mode or turn off the devices when they are inside their building. Consider



(a) BlackBerry 10.2 (b) Android 4.2

Figure 2.4: Profile Management on Smartphones

a situation where a smartphone user goes to the library every weekend and turns his/her cellphone to silent or vibration mode. When the smartphone user changes the profile continously, it makes sense for the smartphone to do it automatically. The smartphone may prompt the user after a certain period that if he/she wants the device turn to silent or vibration modes automatically when the GPS detects that the user is inside this library. On user's permission, the smartphone may do this automatically until the user changes his/her settings. Same logic goes for hospitals as well. But none of the smartphone platform provides this functionality. We hope that in time any of the leading smartphone manufacturers concentrate on this area and introduce the same or similar functionality.

2.2.3 Time based Profile change

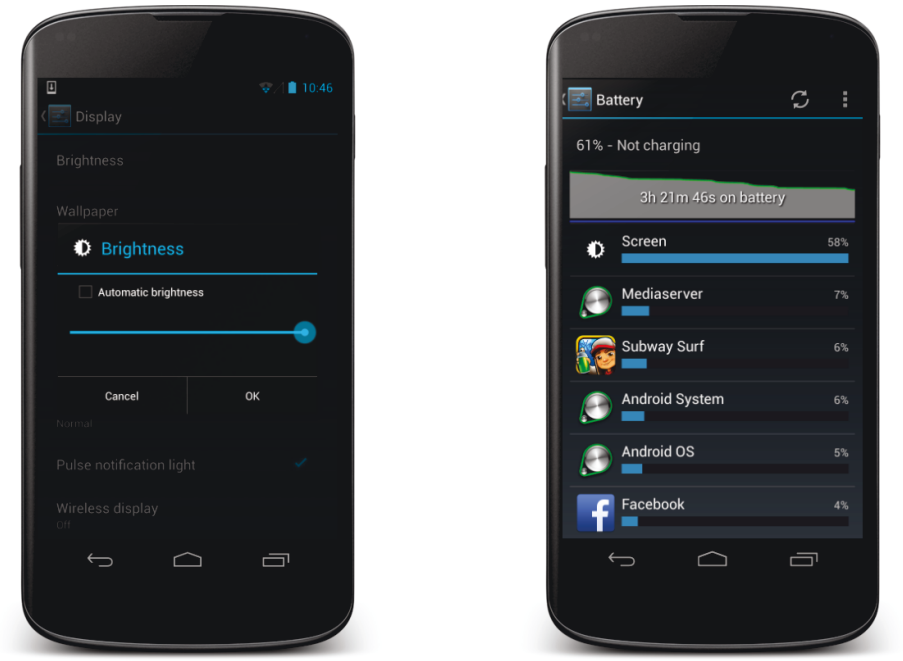
Most of the people in this world follow the same routine: wake up in the morning, go to work, come back from work, do some hobbies and go to the bed. The routine may vary person to person and even for the same person, the routine may vary as he/she grows old. But for a significant period of time, the routine for a student or a working professional would remain the same. For example, if a smartphone user goes to bed at around 11 pm every night and he or she turns his/her phone to "Phone

Calls Only" mode, the device should keep track of this for a certain period of time it should alert the user asking if he/she wants this profile change to be automatic. If the user permits this action, the smartphone should start doing it immediately. This functionality is currently not available in any of the platforms: Android, BlackBerry, iOS, Windows, etc. Figure 2.4(a) shows the manual profile change on BB10 device (BlackBerry Z10) and Figure 2.4(b) shows the manual sound settings screen on an Android device (Google Nexus 4). If we compare these two screenshots, we understand that BlackBerry is ahead of Android in profile management. BlackBerry users can have different notification settings for different apps and combine them into a single profile that shall be activated on a single tap. But Android users should change the sound settings individually. There is no profile management in Android. We hope that smartphone industry will look into this and provide a suitable solution.

2.2.4 Battery Consumption

In a smartphone or any other mobile devices, battery consumption by individual apps and OS services are always a big concern. As the mobile devices are of limited resources and the users expect a longer battery lifetime for a single recharging cycle, it becomes very important for the platform designers and independent application developers to consume the battery as least as possible. Each platform (Android, BlackBerry, iOS, etc.) and each smartphone manufacturer (Samsung, LG, Sony, BlackBerry, Nokia, etc.) deploy their own mechanism and threshold values to switch off various services (E.g. Mobile Data Connection, Wifi, etc.) and minimize some services (E.g. Screen Brightness) at various battery levels. Even though a single manufacturer (e.g. Samsung) makes smartphones with the same platform (Android), their devices are of wide variety ranging across different specification.

Each smartphone model's threshold values and mechanisms vary according to their device battery type, battery capacity, needs, features and design. But the problem with this strategy is that the user is not taken into account in fixing these threshold values. Banerjee et al. [28] states that the battery consumption and recharging patterns varies greatly across users and platforms. But the phone does not try to learn what the apps are the user is frequently using, at what battery level he/she is using them or how much battery those sessions consume, etc. In result of that, even if



(a) Brightness Setting (b) Battery Monitor

Figure 2.5: Android Energy Consumption at Maximum Brightness

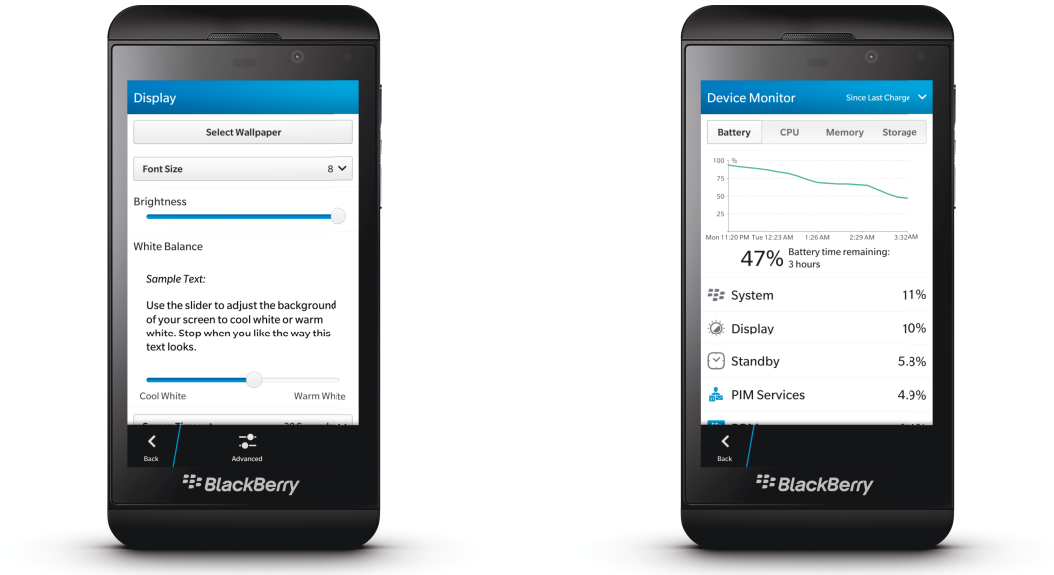
the user wanted to use some services in the background, the devices switches them off automatically without the user's knowledge. So, in this thesis, we try to address the smartphone adaptation to user on battery consumption by tracking their usage and battery recharging patterns. The following sections will explain the existing strategies and our proposed methodology in detail.

2.3 Battery Saving Techniques

2.3.1 Default

Most smartphones have built-in power saving mode that may or may not be configurable by the users. Power saving mode may decrease the screen brightness, shut down services like mobile data, WiFi, etc. depending on the manufacturer's design and platform. Among all the services and apps running on a smartphone, the LCD screen always consumes higher energy.

We set one of our devices Google's LG Nexus 4 (OS: Android 4.2 JellyBean) to maximum brightness and tested the energy consumption by various entities of the



(a) Brightness Setting (b) Battery Monitor

Figure 2.6: BlackBerry Energy Consumption at Maximum Brightness

smartphone. Figure 2.5(a) shows the screenshot of the brightness setting and Figure 2.5(b) shows the battery consumption by various services and applications. In 3 hours 21 minutes and 46 seconds, the phone had reached 61% battery level from its original 100%. The device had lost 39% of battery out of which 22.62% of the total battery capacity, that is, 58% of the current energy loss, is actually consumed by the screen. According to this test run, media server is the second highest energy consuming service. It has consumed 7% of the current energy loss, that is, 2.73% of the total battery capacity. This clearly confirms that no service or application is even close to the screen consumption.

Figure 2.7 shows a snippet from the open source Android code: Power.java [5] that defines the values for power management variables. Full brightness is set to 255 and no brightness is set to 0. From the line 72, we infer that 10% battery level is considered as low battery. And the line 66 says at low battery, the screen brightness will be set to 10.

To understand the effect of brightness on battery consumption, we set our Nexus 4 device to minimum brightness and monitored the battery consumption. To be consistent with our test results we recharged the phone to 100% battery level and started using it for the same time as in Figure 2.8. In 3 hours 21 minutes and 46

```
48.     /**
49.      * Brightness value for fully off
50.      */
51.     public static final int BRIGHTNESS_OFF = 0;
52.
53.     /**
54.      * Brightness value for dim backlight
55.      */
56.     public static final int BRIGHTNESS_DIM = 20;
57.
58.     /**
59.      * Brightness value for fully on
60.      */
61.     public static final int BRIGHTNESS_ON = 255;
62.
63.     /**
64.      * Brightness value to use when battery is low
65.      */
66.     public static final int BRIGHTNESS_LOW_BATTERY = 10;
67.
68.     /**
69.      * Threshold for BRIGHTNESS_LOW_BATTERY (percentage)
70.      * Screen will stay dim if battery level is <= LOW_BATTERY_THRESHOLD
71.      */
72.     public static final int LOW_BATTERY_THRESHOLD = 10;
73.
```

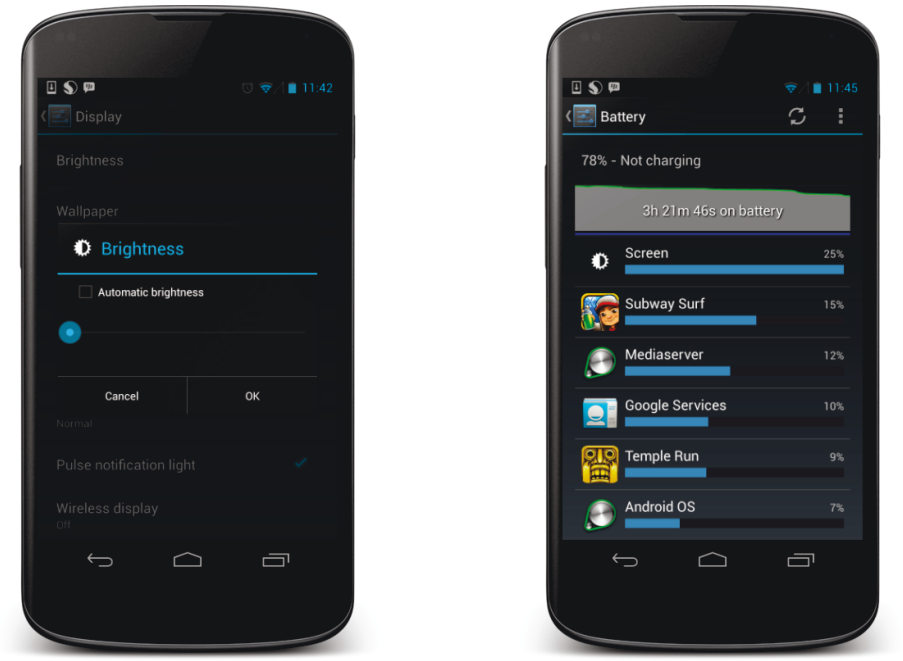
Figure 2.7: Threshold value for Low Battery Level on Android

seconds, the device has lost 22% of its total energy out of which only 5.5% (25% of the current energy loss) was consumed by the display screen. This is much lesser when compared to Figure 2.5 that shows 58% battery consumption by the screen.

Android also provides an option for users to set their device screen to "Automatic Brightness" which adjusts the brightness of the screen according to the room light. That is, when the room light is bright like a sunny day on the streets, the brightness on the device is set to a higher value so the users will be able to read from the screen easily and if the room light is dark, the brightness is set to a lower value to save battery consumption. In Figure 2.8(a) and Figure 2.5(a) we see there is a check box "Automatic Brightness". When the user enables this option, Android will immediately adjust the screen brightness according to the room light and will continue doing so until the user disables "Automatic Brightness".

2.3.2 Static Solutions

Apart from platform's default battery saving techniques, there is a lot of static solutions proposed by academic and industry researchers from various parts of the globe. Some of the those solution are available as commercial applications on app stores like



(a) Brightness Setting (b) Battery Monitor

Figure 2.8: Android Energy Consumption at Minimum Brightness

Google Play Store, Amazon Appstore for Android, BlackBerry World, etc. By the term "static", we mean that the solutions remain the same irrespective of the device and user. They consider the usual battery consuming services and try to provide a longer battery life time. We have identified a few important solutions from research articles, journals and also a few applications that are available on app stores.

1. Battery Life Time Extension Method Using Selective Data Reception on Smartphone [26]

2. Battery Life Time Extension Method By Using Signalling Interval Control [30]

3. Battery Doctor [2]

4. Juice Defender [12]

5. Go Power Master [8]

Battery Life Time Extension Method Using Selective Data Reception on Smartphone

This paper presents a solution to our battery consumption problem by analysing different types of applications that drain the smartphone's battery. The authors of this paper [26] identify social networking applications like MyPeople, Google Talk, Never-Talk and Yahoo Messenger as a significant influence on battery consumption. So, they propose a model to address this problem by building a SNS (Social Networking Service) application server that would receive battery state from the smartphones and decide if the smartphone has enough energy to receive huge data. The data from various social networks are actually routed to the smartphones through the SNS application server.

Figure 2.9 shows the complete work flow of the SNS application server that illustrates how data received from various social networking services are transmitted to smartphones. The SNS application server has two threshold values: Threshold I is packet size and Threshold II is battery state. The packet size threshold value helps to determine the type of data that has been received: text or picture or video. The battery state threshold is to determine if the smartphone has enough energy to receive this data type. So, when a data packet is received by SNS application server, it verifies if the packet size is greater than Threshold I. If not, the server transmits the packet immediately. Otherwise, it verifies if the battery state of the smartphone is less than Threshold II. If no, again the server transmits the data immediately. Otherwise, the packet transmission is rejected and ready to listen to a new packet.

The authors of this paper [26] evaluated their method in both 3G (UMTS) and WiFi (IEEE 802.11) environment. They compared their selective mode reception with normal mode and found that their selective mode has a longer battery life. Figure 2.10(a) and Figure 2.10(b) show their evaluation results in 3G and WiFi modes respectively.

Battery Life Time Extension Method By Using Signalling Interval Control

This paper, from the same authors of "Battery Life Time Extension Method Using Selective Data Reception on smartphone" [26], presents another solution for the

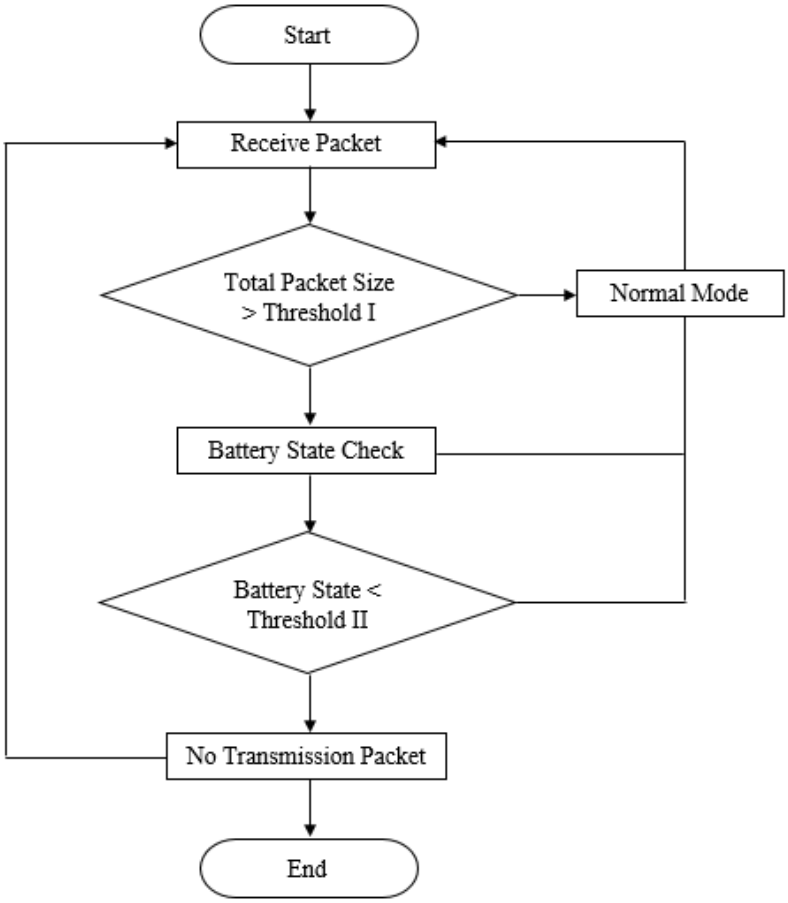
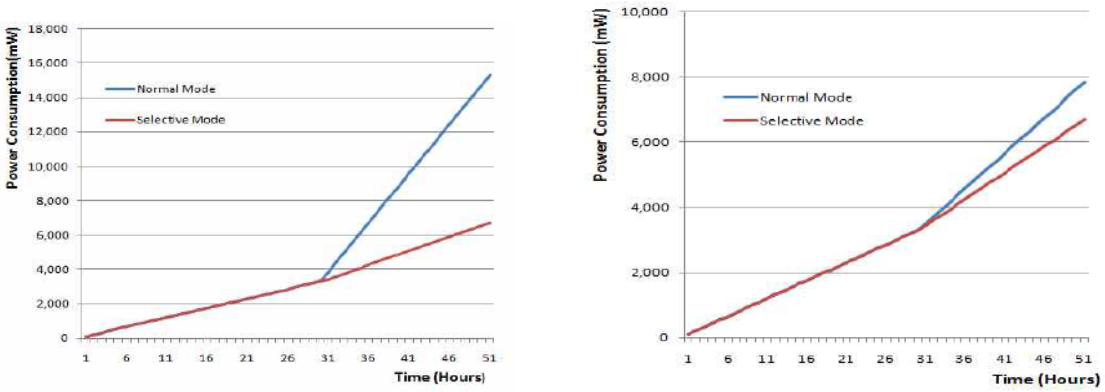


Figure 2.9: Flowchart of Selective Data Transmission in SNS Application Server



(a) 3G Mode

(b) WiFi Mode

Figure 2.10: SDR - Power Consumption Comparison

same problem of battery consumption by SNS applications. Here, the authors have taken different approach of developin the solution within the smartphone instead of an intermediate application server. In this particular paper [30], they discuss the signal exchange between smartphone and the SNS agent server and also its effect on battery consumption. When any SNS application is installed on a smartphone and logged in, the application sends "Keep-Alive" messages to its SNS agent server to maintain the user information. These signalling messages are exchanged on a constant interval irrespective of the smartphone's battery level. As this traffic is not very important to the user, the signalling can be delayed to save some battery and extend its life time.

Figure 2.11 shows the complete workflow of the signalling interval control (SIC) method. The SIC application is constantly looking at outgoing packets and verifies if it is a SNS Keep-Alive message. If not, the message is transmitted. Otherwise, as a next step, the SIC application checks the battery level if it is less than the threshold value. If not, SIC application assumes that the smartphone has enough battery and so it operates in the normal mode by allowing the transmission immediately. However, if the battery level is less than the threshold, the SIC application switches itself to battery saving mode and calculates the time to be delayed based on exponential back off equation in [22]. After the delay time, SIC application transmits the signal message to SNS agent server and starts verifying the next packets.

The signalling interval control method was evaluated by comparing the battery consumption by Keep Alive Messages (KAM) in Battery Saving mode against Normal Mode. The authors tested their solution on both 3G (UMTS) and WiFi environments. As the battery level goes down, their signal interval becomes higher and so the battery consumption by the KAM in battery save mode is very low whereas in normal mode, irrespective of the battery level, KAM consumes same energy. Figure 2.12 shows their performance analysis by comparing power consumption by KAM in normal mode and battery saving mode.

Battery Doctor

One of the most popular battery saving application available on Google Play Store (https://play.google.com/store) is Battery Doctor. The application is downloaded by more than 50 million users and has been rated on an average of 4.5 out of 5.0 by

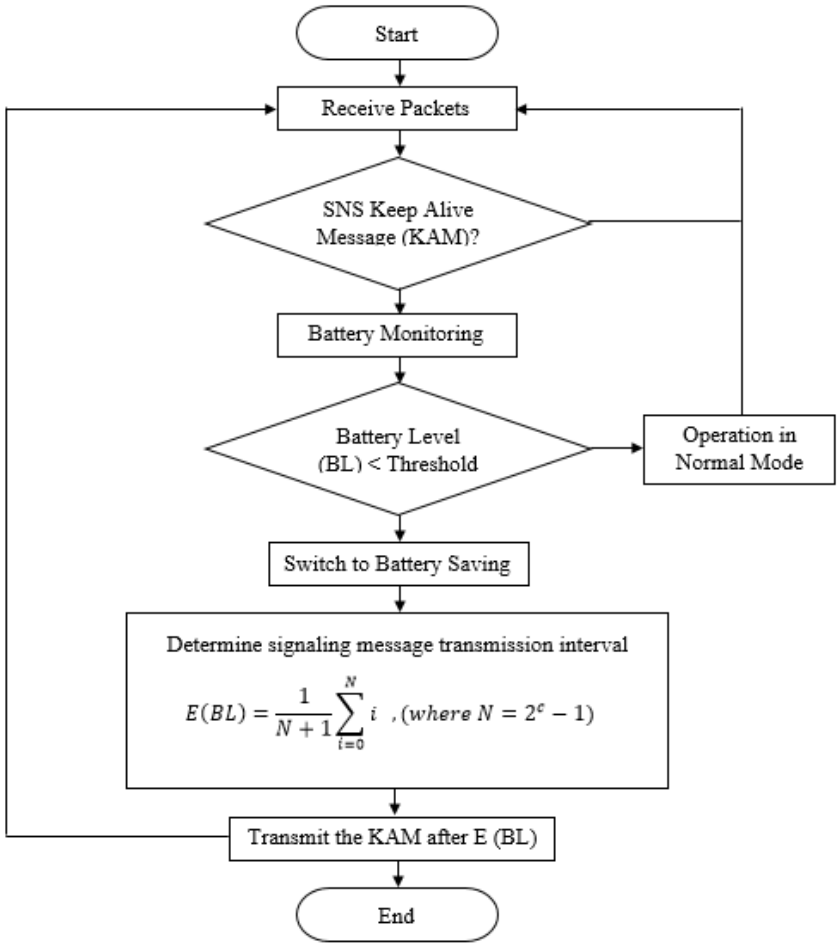
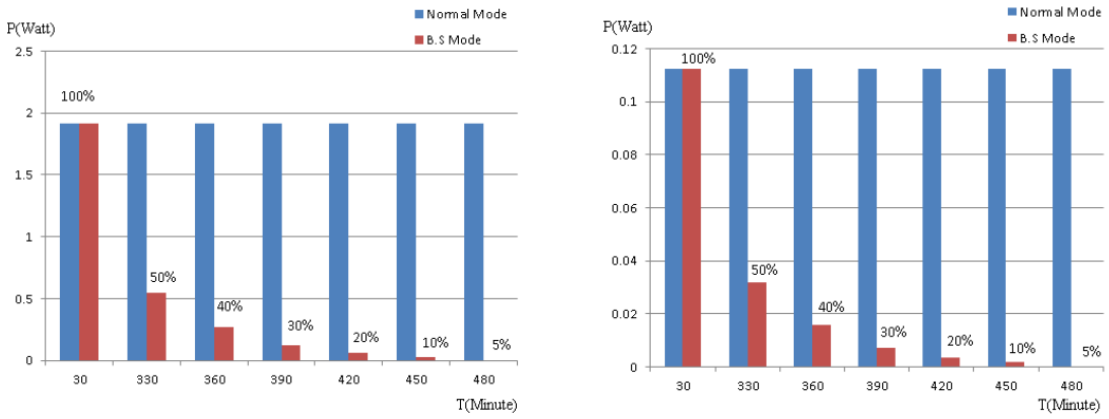


Figure 2.11: Flowchart of Signalling Interval Control Method



(a) 3G Mode

(b) WiFi Mode

Figure 2.12: SIC - Power Consumption Comparison

(a) Charging (b) Discharging

Figure 2.13: Battery Doctor

over 2 million users [12]. The application helps to improve the battery health by addressing memory effect. That is, Battery Doctor reminds the user when the battery is getting over charged and also it monitors and manages the energy consumption. In addition, it helps the battery to get charged in 3 stages - speed when low, continuous when moderate and trickle when high. The application offers great features like fixing schedules for work, sleep, etc., brightness, WiFi, Data connection settings for each mode or schedule, individual task killer, etc. Primarily, it kills the battery draining applications as an energy saving mechanism. When the battery is in the charging state, it notifies the users of the approximate time left to fully charge and when the battery is in the discharging state, it notifies the users how long the battery will last until it completely drains out. Figure 2.13(a) and Figure 2.13(b) displays screenshots of the Battery Doctor application in charging and discharging states respectively.

Battery Doctor claims to extend the battery life up to 50%. We evaluated the application's performance by measuring the smartphone's total time to reach zero battery level from 100%. We compared it against the condition of the device without Battery Doctor application installed.

Battery Doctor ON	Battery Doctor OFF
17 hrs 53 mins	16 hrs 40 mins

Table 2.1: Battery Doctor Performance - Total Battery Lifetime

Juice Defender

Another important battery saving application available on Google Play Store (https://play.google.co is Juice Defender. The application is downloaded by more than 10 million users and rated on an average of 4.4 out of 5.0 by more than 250 thousand users [2]. Unlike other battery saving applications, Juice Defender is not a task killer. It just optimizes the energy consumption by letting the smartphone run at its full capacity when needed and save the battery otherwise. It manages the power consumption by monitoring battery draining services like WiFi, 3G/4G high speed data connection, etc. Juice Defender comes with a few preset modes - balanced, aggressive and extreme. Each mode has a predefined setting. For example, screen brightness value, 3G/4G data connectivity switch to turn on/off, etc. In the balanced mode, when the screen is off, the application delays the background synchronization by 15 minute intervals and when the screen is on, the application does not touch anything. In the aggressive mode, the delay is extended to 30 minutes for background connections and when the battery becomes really low, the application disables the background processes completely even if the screen is on. The extreme profile turns off the connectivity by default but offers users to enable it for particular apps of their interest.

Juice Defender also offers users flexibility to customize the modes according to their individual needs. Notable customizations are Mobile data switch, WiFi switch, Location switch and Background synchronization interval. Figure 2.14 shows a screenshot of the application's home screen. Juice Defender is designed primarily for the user's convenience and offers great interface that helps users manage their settings effectively.

We evaluated the application's performance by measuring the smartphone's total time to reach zero battery level from 100%. We compared it against the normal condition of the device without Juice Defender application installed.

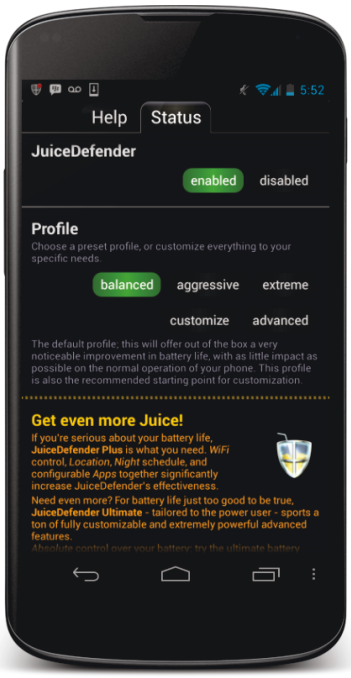


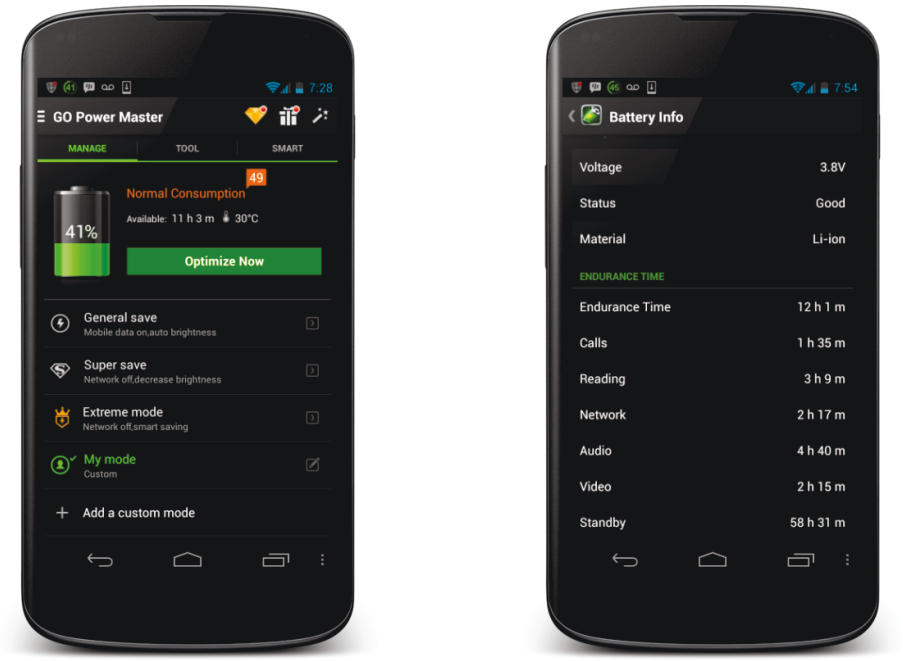
Figure 2.14: Juice Defender

Juice Defender ON	Juice Defender OFF
18 hrs 18 mins	16 hrs 40 mins

Table 2.2: Juice Defender Performance - Total Battery Lifetime

Go Power Master

Another notable battery saving application available in Google Play Store is Go Power Master which has been downloaded by over 10 million users and rated 4.4 (average) out of 5.0 by over 180 thousand users [8]. Go Power Master is almost similar to the two application we previously discussed except that it offers battery life estimation in case of switching down any service like WiFi, 3G/4G data connection, etc. Therefore, the users will be able to decide if switching down any service will be helpful to them. Figure 2.15(a) shows the screenshot of Go Power Master application's home screen. The application also tells the user how long the battery will last if any particular service is used throughout. Figure 2.15(b) shows the endurance time of different features. Like the other two applications, Go Power Master has three modes of operation - General, Super and Extreme. The user interface is not as easy as the previous ones but it perfectly delivers the necessary features of shutting



(a) Home Screen (b) Endurance Time

Figure 2.15: Go Power Master

down unnecessary services and the customization of modes according to user needs. Go Power Master also provides options to manage different components within the application using which users will be able to turn on/off WiFi, Data Connection, Bluetooth, etc. very easily.

We evaluated the application's performance by measuring the smartphone's total time to reach zero battery level from 100%. We compared it against the normal condition of the device without Go Power Master application installed.

Go Power Master ON	Go Power Master OFF
16 hrs 58 mins	16 hrs 40 mins

Table 2.3: Go Power Master Performance - Total Battery Lifetime

2.3.3 Dynamic Solutions

Academic and industry researchers have also been working on user adaptive energy management. These techniques are considered as dynamic solutions as they consider users and the usage patterns. That is, the same solution yields different results based

on various factors - devices, platforms, users and usage patterns. Unlike default and static solutions, this dynamic approach concentrates on improving the quality of experience instead of just extending the battery lifetime. According to Banerjee et al. [28], their tool Llama improves the user experience by providing a brighter display and continuous connectivity without significant compromise in battery lifetime. We also identified a few popular applications on app stores that take a dynamic approach in solving energy consumption concerns. So far proposed solutions by researchers that are very important for our thesis work are discussed below.

1. Users and Batteries: Interactions and Adaptive Energy Management in Mobile Systems [28]

2. Context-aware Battery Management for Mobile Phones [29]

3. Snapdragon BatteryGuru [18]

4. Battery Drain Analyzer [3]

Users and Batteries: Interactions and Adaptive Energy Management in Mobile Systems

This research work is a very intensive study on user adaptive energy management. This paper was published in the year 2007 when the smartphone evolution became exponential. The authors have done a detailed study on how user's usage pattern and recharge cycle impact the battery lifetime. They also present interesting results from their analysis of collected data, user interviews and in situ survey. Banerjee et all [28] listed three findings from their study:

1. Users recharge their phones mostly when they have significant energy left.

2. Users recharge their phones mostly by context like location, time, etc. and battery levels.

3. There is a significant difference in recharging behaviour and battery usage among various users and devices.

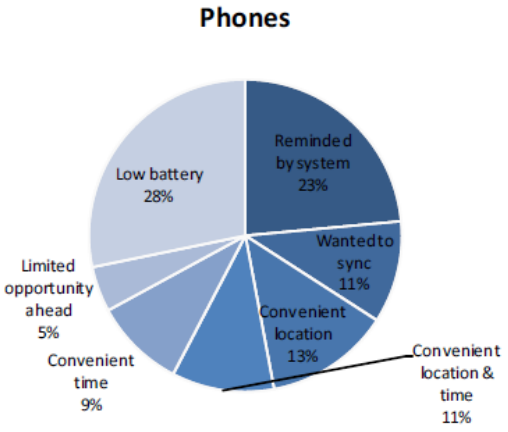


Figure 2.16: Adaptive Energy Management - In situ survey results

These three findings especially the first one motivated the authors to concentrate on adaptive energy management approach in solving battery consumption concerns. If the users recharge their phones at higher battery levels, it is unnecessary to switch down services. That is, if we estimate the battery level the user will recharge his/her phone at, then we can substantially increase the quality of service and thereby the user experience. Figure 2.16 shows their in situ survey results. Only 28% of recharge happened by low battery alarm whereas 49% happened by context like location, time, synchronization need, etc.

But when the system tries to consume the excess battery by offering better user experience, the smartphone may start losing energy faster resulting in a quicker drain. Also, users may recharge their phone sooner as a result of faster battery drain. This becomes a feedback loop. The authors identified three important recharging behaviour - context, battery level and low battery conditions. When users recharge their smartphone by context based like location and time, the adaptive approach may have little or no significant effect noticeable by users. But when they charge based on battery levels, as adaptive approach tries to consume extra energy, the battery level decreased sooner result in users tending to recharge frequently. However, the main concern is when they recharge based on true low level battery. So, as a dynamic user centric approach in solving energy consumption issues, Banerjee et al [28] designed a system named Llama to manage the battery energy addressing all the concerns including feedback loop.

Confidence of not exceeding battery capacity $= p$
Histogram for CDF of recharges given present battery remaining $C_p = H$
Size of Histogram bin $= \Delta H$
Find x such that $H(x) \leq (1-p) \leq H(x + \Delta H)$
Excess energy for Llama tasks $= x$
Energy for foreground tasks $E_f = 100 - x$

Figure 2.17: Llama: Energy Adaptive Algorithm

Llama learns usage patterns from its users rather than some preset knowledge of expected battery lifetime. The basic concept behind Llama is very simple. When the system detects excess energy until the expected next recharging, it increases the user experience by allotting energy to noncritical tasks and delivers brighter screen, shorter synchronization cycle, website caching, etc.

Figure 2.17 shows the energy adaptive algorithm used in Llama. The authors use probabilistic algorithm to estimate excess energy and for allotment to non-critical services so that the impact of decreased battery time is only minimal. Llama also takes into consideration of both short term and long term behaviour changes in usage patterns. The system efficiently predicts extra energy by using histogram of previous energy usages and current battery level. The algorithm is iterative and so Llama predicts periodically and adapts to new results so that battery will not be drained anytime sooner than expected.

Llama uses mean of past recharge times to calculate the time before next recharge may happen. The above algorithm calculates probability distribution of previous recharges to estimate excess energy that can be allotted to Llama applications. For evaluation of this system, the authors implemented this algorithm and deployed it on 10 Windows based mobile phones. They also built a remote server to collect data from the phones. They also took the energy consumption for this data transfer into calculation so that their evaluation will be accurate. Figure 2.18 and Table 2.4 show their evaluation results with respect to recharging behaviour change. From the table we clearly understand that the post Llama, users have attempted less number of recharges per week compared to pre Llama. The authors also presented their user study conducted before and after Llama was installed.

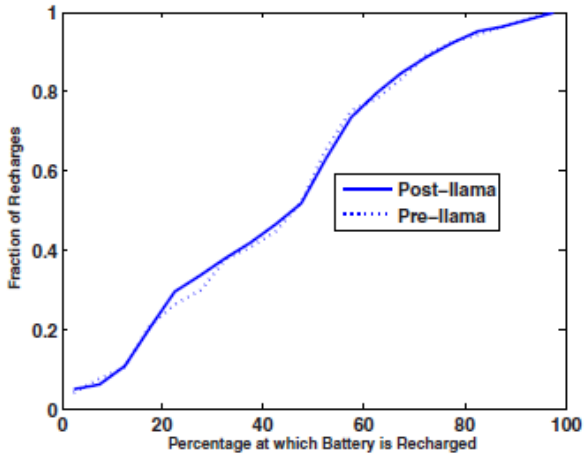


Figure 2.18: Cumulative Distribution of All Recharges before and after Llama Installation

Before Llama	After Llama
10.1	8.9

Table 2.4: Llama - Average charge attempts per week

Context-aware Battery Management for Mobile Phones

Ravi et al. in this paper concentrates on user centric approach in solving energy consumption issues. The authors designed a system called CABMAN (Context Aware Battery MANagement) to warn the user if it predicts that the battery may drain before his/her usual recharge time. Their algorithms finds the next recharge time by using the past data, how much calls he/she may make during the current battery discharging period, how much energy is needed for those calls and how long the battery will be available if the current applications and services continue to run at same energy rate.

Figure 2.19 shows the system architecture of CABMAN. The authors designed the whole system in a modular approach designating each module to a single specific task. If we look at the level just above operating system, we have four monitors with specific assigned tasks. Context monitor helps the system to predict next recharge time by using the cell ID of the mobile network the phone is connected to. Alternatively, WiFi access points and GPS shall also be used to find the users location and the recharging opportunity. The algorithm keeps track of the location and time when

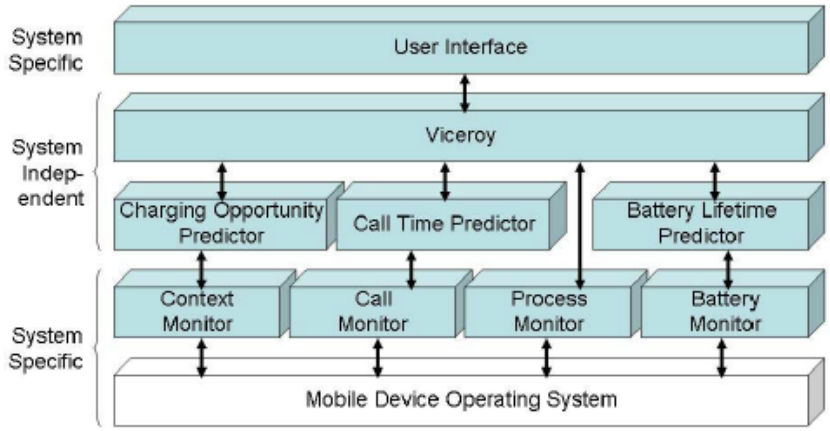


Figure 2.19: CABMAN System Architecture

user charges his/her phone and uses this data for prediction. The probable time until next recharge is found by averaging the time difference between the time of entry into current cell and the time of entry into the probable charging cell.

The call monitor assists the system to predict probable call time needed by the user until the next recharge opportunity. The authors discussed three different options to perform this task. First, prompting the user to tell us how much call time he/she expects to take which is not very efficient and it will also irritate the user. Second tracing the call history by time of day and call duration time at that particular hour. Third approach is to combine call trace with users feedback which is more accurate than the previous two. The battery monitor periodically provides battery state information to the system that is useful in predicting its battery lifetime taking application usage into consideration.

The authors evaluated their individual module in the CABMAN system by calculating the prediction errors. First, as shown in Figure 2.20, in evaluation of recharging opportunity prediction, as the sample size increases, the prediction error decreases. That clearly says, the system will perform better day by day as it collects more data. Ravi et al. also evaluated the other predictions by different parameters that can be found in [29].

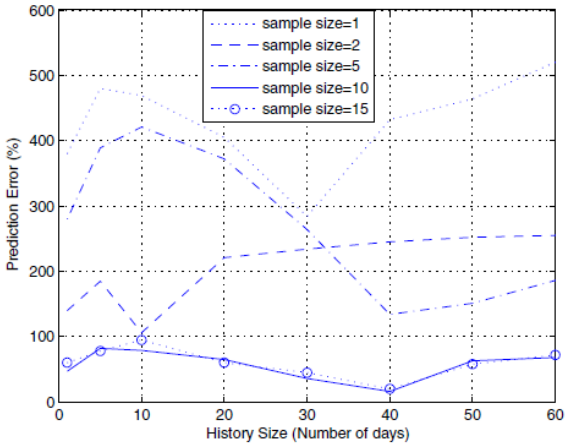


Figure 2.20: CABMAN - Recharging Opportunity Prediction Error

Snapdragon BatteryGuru

One of the recent and excellent battery saving applications that takes user centric approach is Snapdragon BatteryGuru [18] published by Qualcomm Connected Experiences Inc., a Qualcomm company. Qualcomm is the leading mobile device chip manufacturer and plays a vital role in this smartphone industry. A subsidiary company Qualcomm Connected Experience Inc. is working on software and hardware optimization technologies that improve overall user experience. Snapdragon BatteryGuru application is available on Google Play Store for free. It has been downloaded by over one million users and rated on an average of 4.1 on a one-to-five scale by over 30 thousand users. The reason behind the smaller number of ratings and reviews by users compared to the applications we discussed in previous sections, is that the application is very new to the market and will catch up down the line.

The goal of BatteryGuru is to extend the battery life and improve the user experience by optimizing the devices functionality features. This application works only on mobile devices that carry Snapdragon mobile processors. As this application has direct access to the processor chip, they claim that their calculation of energy consumption by various services and applications and estimation of battery life time are almost more accurate than any other third party applications that are available on app stores in plenty. The basic concept behind BatteryGuru is that it learns how the user uses his/her mobile device (with Snapdragon mobile processor) and optimizes

(a) App Refresh Setting (b) Activity Screen

Figure 2.21: Snapdragon BatteryGuru

various energy consuming services without disabling its functionalities. The app does not require any user input and is completely automatic. However, on user's preference, it offers flexible customization. For example, if the user desired BatteryGuru not to manage any of his/her application, he/she can opt out. The application has three main screens - Status, Apps and Actvitiy. In status tab, the app shows the estimated battery life time until next probable recharge and few tips. Apps tab shows a list of apps that are managed by BatteryGuru and allows the user to change the way they are managed. Figure 2.21(a) shows a screenshot of the options that are available to users. The application also shows a detailed data of various activities like WiFi, App refresh, Battery Level etc. Figure 2.21(b) shows the activity tab of BatteryGuru application.

Unlike our previous straight forward evaluations, we cannot calculate the total battery life time a smartphone has got after the app installation. As this application provides a dynamic and user centric solution, the performance will vary over time. So Table 2.5 shows average discharge time of BatteryGuru.

BatteryGuru ON	BatteryGuru OFF
18 hrs 36 mins	16 hrs 40 mins

Table 2.5: BatterGuru Performance - Total Battery Lifetime

Battery Drain Analyzer

One another important battery saving application that tracks users' usage pattern and application centric battery life estimation is Battery Drain Analyzer. The application has been downloaded by over hundred thousand users from Google Play Store and has been rated on an average of 4.1 out of 5.0 by more than 1000 users. Battery Drain Analyzer is a simple and powerful application that offers efficient battery management through users' recharging behaviour. That is, the application tracks the time when the smartphone user usually recharges his/her phone and uses that information in predicting current battery lifetime until next recharge. It also provides a screen with a list of applications and their battery consumptions as shown in Figure 2.22(b). However, the application does not try to control or manage those battery draining applications. It just provides the information to the user. One of the greatest features is that it shows the energy consumption by various applications and services in a day (free app) and in the past one month (premium app) as a graph that is easy to understand and interpret by the user. Battery Drain analyzer also provides buttons to easily enable and disable the services like WiFi, Airplane Mode, Bluetooth, Mobile Data, etc. The toggle buttons, estimated battery life time and the current battery status are all available on the application's home screen as shown in Figure 2.22(a).

Although Battery Drain Analyzer uses application centric dynamic approach to predict current battery life time, its solution for extension is static by having three simple profiles - Day, Night and Save. These profiles can be activated either automatically on time basis or manually by the users. As we evaluated the other applications, we evaluated Battery Drain Analyzer. We recharged our Nexus 4 device to 100% and allowed the battery to drain completely on a normal day-to-day usage. We evaluated the app with 3 profiles activated on time basis (8hrs Day, 8hrs Night and 8hrs Save). The results are shown in Table 2.6.

(a) Home Screen (b) Analyzer Screen

Figure 2.22: Battery Drain Analyzer

Battery Drain Analyzer ON	Battery Drain Analyzer OFF
18 hrs 26 mins	16 hrs 40 mins

Table 2.6: Battery Drain Analyzer - Total Battery Lifetime

2.4 Energy Consumption on iOS

All the mobile platforms, like Android, suffer from energy consumption issues. As iOS being the second most popular mobile operating system, next to Android, it becomes necessity to research on the available battery saving solutions on iOS platform and point out the differences between Android and iOS energy consumption issues. The two entirely different mobile operating systems cannot be compared to each other directly for various reasons such as the difference in displays, difference in hardware capabilities, difference in sensors, etc. Moreover, there are thousands of devices manufactured on Android platform and iOS devices are made in various form factor with different funtionalities. As per an article in BGR [11], eight flagship phones were compared for battery lifetime including iPhone 5S and iPhone 5C. With respect to voice call, Samsung Galaxy S4 aced the competition with 1051 minutes

and HTC One took second place with 771 minutes. With respect to internet use, again Samsung Galaxy S4 aced it with 405 minutes and its sibling Samsung Galaxy S4 mini captured second slot with 394 minutes. In both the aspects, iPhone 5S and 5C's performance seems bad. However, this can be justified by the high defeinition displays, fine hardware and rich user experience offered by iOS devices.

A lot of battery saving applications are available on Apple's Appstore (iTunes) ranging from simple service toggle applications to complex mechanisms. Of all the five different applications discussed earlier on Android platform, only Battery Doctor available on iTunes. The application page on iTunes claims the application is used by over 50 million iOS users. The iOS version of Battery Doctor offers same features as the Android version such as discharge rate and remaining time estimation. More information on iOS and other mobile platform's ennergy consumption is out of the scope of this thesis As mentioned earlier in the section 1.2.1.

2.5 Motivation and Research Problem

The primary motivation of our thesis is the findings from Banerjee et al. [28] - users mostly recharge their devices when there's a significant energy left, most of the recharges are context driven and there is a great difference in users' recharge behaviour and usage pattern. And also as per Perrucci et al. [25], the processing power of mobile devices doubles every two years whereas battery capacity doubles only in a decade. So, it is necessary that limited energy resource available in mobile devices must be used efficiently and optimized according to the user needs.

We understand that the "One size fits all" solution does not apply here. All the default and static techniques we discussed try to extend the battery life time by shutting down the services at low battery levels. Thereby, users are unable to use their critical tasks and in turn overall user experience declines. From our literature survey, it is very clear that the user centric approach benefits users both by improving overall user experience and by extending their battery lifetime. But the user centric approaches we discussed in our previous sections are mostly dynamic in their learning but static in their solutions. For example, Battery Drain Analyzer shows how much the battery is consumed by various applications. But it does not use this data to optimize the energy distribution. Likewise, Snapdragon BatteryGuru application

provides information about users' activities and manages app refresh intervals. But it fails to get the user's preference on priorities of his/her apps and services. Additionally, the app only works on the Snapdragon powered mobile devices. Although, Banerjee et al. [28] solves the energy consumption issue through a user centric approach, they consider only the user's recharge behaviour but not his/her app and service usage pattern.

Considering pot holes in all the solutions and approaches we discussed, in this thesis, we try to find an optimum line between user's overall experience and battery lifetime extension by determining if energy is to be saved for future use or if energy can be consumed for current need according to the estimations of next probable recharge, applications and services that have high probability of using in near future and how much energy will be consumed by those applications or services.

Chapter 3

Methodology

3.1 Overview

In this chapter, we propose our methodology that solves energy management issue in dynamic approach. The basic concept behind our approach is that user profiling with respect to time and duration of each user's application session varies significantly as found out by [28]. In brief, our whole methodology consists of two phases - learning phase and activated phase.

1. Learning Phase: We monitor user's activities and record the data locally on his/her smartphone for a significant period. We do not apply any energy saving mechanism throughout this phase. As shown in Figure 3.1, Logger records the data in the device local database. The other two modules are inactive during this Learning phase.

2. Activated Phase: We use the data that is collected in estimating future values and apply it to our unique approach to enhance the user experience and try to extend the battery life per recharge cycle. We also keep recording the data to make our algorithm work better. As shown in Figure 3.2, Logger continues to store the data by monitoring user's recharge cycle and usage activities. In addition, the Estimator module and Distributor module become active to save and distribute the energy efficiently. These modules are explained in detail of how they work in corresponding sections.

As shown in our solution architecture Figures 3.1 and 3.2, our approach contains three modules - Logger, Estimator and Distributor. These three modules combined work together with the smartphone operating system and its associated database to achieve our goal in solving energy management in a user centric approach. As shown in Figure 3.1, the Logger module is the entry point. For a significant period

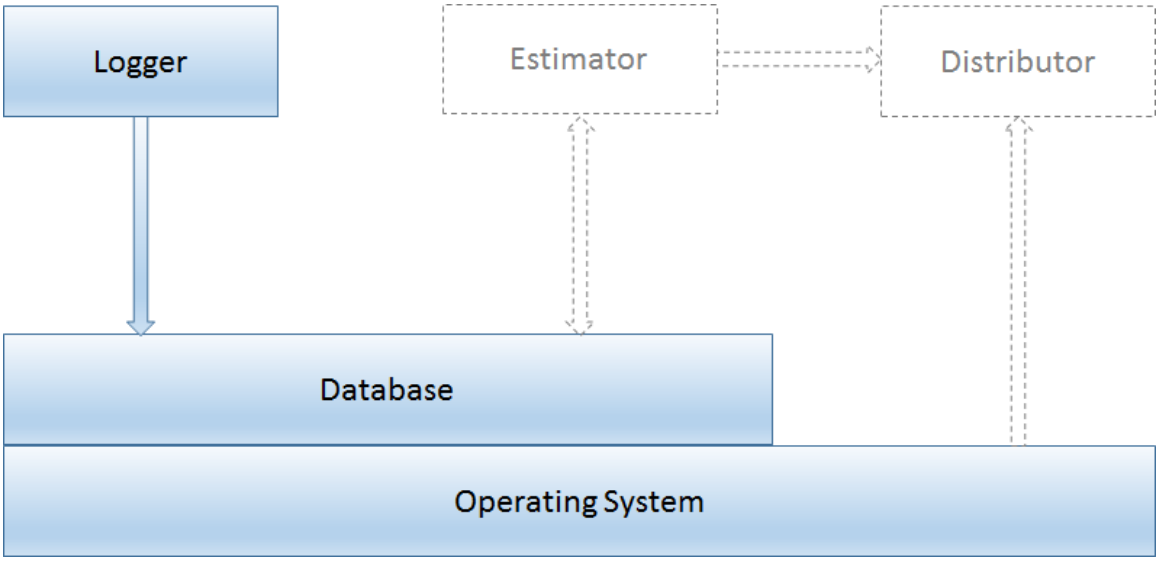


Figure 3.1: Solution Architecture - Learning Phase

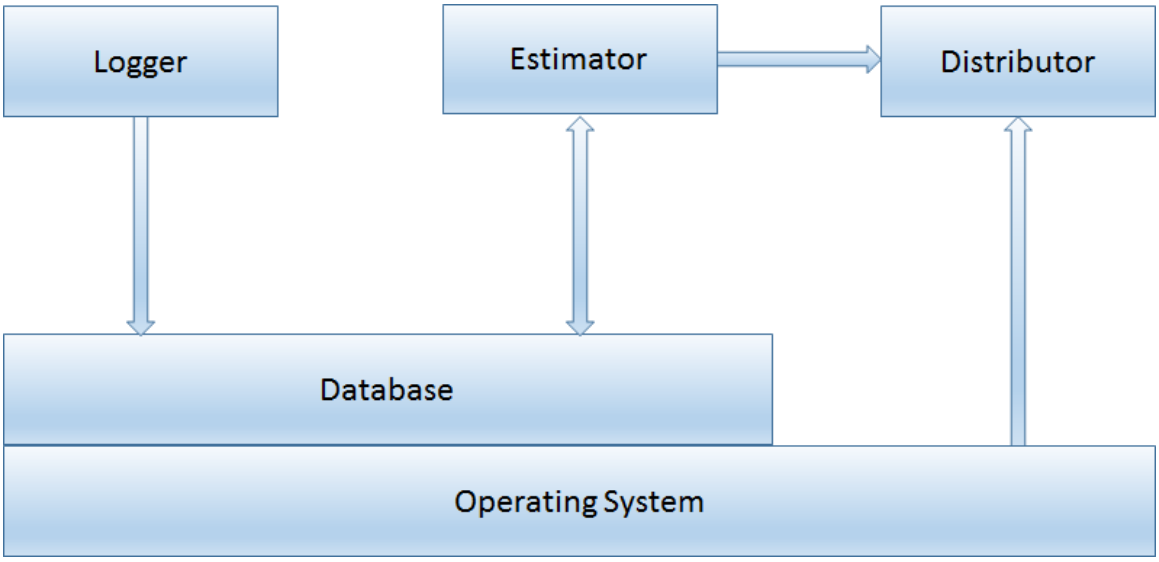


Figure 3.2: Solution Architecture - Activated Phase

of time, Logger collects the necessary data from the user's recharge behaviour and usage activity on his/her device. This significant period of time is called the Learning Phase. Logger stores all the data locally in the smartphone and provides access to the Estimator module. Logger continues collecting data after the Learning phase to make the solution better with the data. As we have seen in our literature survey, Figure 2.20, it is evident that increase in sample size minimizes the prediction error. So, Logger continues to collect data in the Activated Phase. Once the device enters the Activated Phase, the Estimator module fetches the data from local storage and uses them to estimate various items like, Next probable recharge time, the applications or services that may be launched before next recharge and the energy required by those applications or services and the battery life time until the next recharge. Then these estimated values and some definite values from the operating system are used by the Distributor module to make decisions as to what applications or services are allowed. In following sub sections, each module is described in detail of what they do, how they work independently and all together.

3.2 Logger

In our proposed solution, the Logger module collects all the necessary data and stores them in the device's local database that is accessible by the other two modules - Estimator and Distributor. Logger is the entry point of our approach and runs independently for a significant period of time before the other two modules become active. To track a user's recharge cycle and usage behaviour, it is important to know his/her daily routine. As most of the people in most parts of the world study or work five days in a week and take two days off from their routine life, we fixed this significant period as seven days (Monday to Sunday). So, after the solution is deployed on a smartphone, the Logger starts monitoring the recharge cycle and user's application and service activities and stores them in the local database for the first seven days. After seven days, the Estimator and Distributor modules become active and starts their operation. But, Logger continues to monitor the usage activities and stores the data in a local database to improvize the estimation accuracy and thereby resulting in overall higher battery lifetime extension and enhanced user experience.

Logger is triggered by the following events and in turn the module records various

data that are necessary for our approach in solving energy management issue:

1. When smartphone is plugged into a charger (AC or USB)
2. When smartphone is unplugged from the charger
3. On a regular interval of 1 minute
4. On any service switched on/off

Logger helps to track two areas of a user's activities - Recharge cycle and Application usage pattern. To monitor the recharge cycle, we need to know when the user recharges his/her smartphone daily, what is the energy level of its battery before and after recharge happens and how often he/she recharges his/her device. So when the smartphone is plugged into a charger, the following information is stored in the local database.

1. Day Type (DT)
2. Recharge Start Time (RST)
3. Energy Level at RST (ELST)
4. Recharge Mode (RM)
5. Time After Previous Recharge (TAPR)

The recharge cycle may differ in weekdays and weekends. So, the first item in the database record is the day type which is a boolean value of 0 if it is weekday and 1 if it is a weekend. For the readability purpose, we show the user 'D' for Weekday and 'E' for Weekend. The next item is the actual time of the day when recharge starts. We also store the energy level at the begining of recharge. Recharge mode is used to determine if the charging is slow or fast and helps to estimate the time frame to reach 100% energy level. Logger also calculates the time difference between current recharge and previous recharge which helps in estimating the next recharge time. When the smartphone is unplugged from the charger, the current database record is filled with the following information.

1. Recharge End Time (RET)

2. Energy Level at RET (ELET)

When the user plugs his/her smartphone into the charger, leaves it until atleast 30% energy level and unplugs it, the Logger creates one record in the recharge statistics table and stores them in the device's local database. Table 3.1 is a sample that shows how recharge cycle data is stored in the user's smartphone.

DT	RST	RET	ELST	ELET	TAPR	RM	NRE
D	22:53	02:34	26.0	100.0	12h32m42s	AC	21:45
E	13:25	16:42	14.0	53.0	10h28m37s	USB	14:29

Table 3.1: Logger - Recharge Statistics Sample Data

In addition to the recharge cycle, we also need to record the user's daily usage pattern on applications and services. In order to predict the probable application launches and their energy consumption, we need to analyze his/her past usage details. To do this, we designed two tables in the same device's local database namely App Usage and Service Usage. As per [10], on an average a user spends 1 minute 15 seconds on an app per session although it varies widely ranging from few seconds (e.g. Whether application) to several minutes (e.g. social network application). As a conservative method, to capture most of the application sessions, we set the interval as 1 minute. So, for every 1 minute, the Logger wakes up, checks the foreground application and records the following data:

1. Day Type (DT)

2. Date (DATE)

3. Time of the Day (TIME)

4. Application Name (APPNAME)

5. Package Name (PACKNAME)

6. Number of Hits (NoH)

7. Energy Consumption (EC)

The first column, Day Type identifies if the database record deals with a weekday or weekend. Then, the date is noted down for future verification purposes. Time of the day is necessary data in order to calculate the number of hits. For example, if the Logger creates a record for "Facebook" app at 09:20 and if it identifies the same facebook app open at 09:21, the Logger will not create a new record. Instead, it will increase the "Number of Hits" count. To minimize the data storage, we create records per app per hour and thereby calculating the number of hits and the energy consumption by that particular app. This table is used by the Estimator module to predict the probable application launch and estimate energy consumption. Sample App Usage records are shown in Table 3.2.

DT	DATE	TIME	APPNAME	PACKNAME	NoH	EC
D	2014-08-13	08:00	Facebook	com.facebook.katana	20	5
D	2014-08-13	08:00	Gmail	com.google.android.gm	5	1

Table 3.2: Logger - Application Usage Sample Data

Services like WiFi, Bluetooth, GPS, 3G/4G data services, etc consume significant energy. But, smartphone and its applications need one or more services to perform various actions like sending an Email, making a phone call, browsing a website, sending a file to a nearby phone, etc. Some users may use WiFi more than any other service and some users may use GPS more than anything else. As we mentioned earlier, users show a significant variance in their usage pattern. In order to know the priorities of the user for these services, we need to know how long these services are switched on or off throughout a day. For our solution, we have taken all the important services into consideration such as WiFi, 3G/4G Data Services, Location, Accelerometer, Mobile Network, Bluetooth and NFC. Whenever these services turned on We store the following data except End Time (ET) in Service Usage table.

1. Day Type (DT)
2. Date (DATE)
3. Service Start Time (SST)
4. Service End Time (SET)

5. Service Name (SERVNAME)

When the service is switched off, the End Time (ET) is filled in the current record. This switching off and on may be by the user or the operating system itself. We store this data on both circumstances. Sample data is shown in Table 3.3.

DT	DATE	SST	SET	SERVNAME
D	2014-08-16	08:03	18:45	Mobile Data
D	2014-08-16	18:45	07:28	WiFi

Table 3.3: Logger - Service Usage Sample Data

Logger starts recording this data for the first seven days after deployment and after the Estimator and Distributor modules become active, Logger continues to do its task until the user stops or resets.

3.3 Estimator

The Estimator module collects the data that the Logger module has recorded so far and tries to estimate a series of information as efficient as possible that are critical in managing the available energy until the battery dies. The following information are the derived by the Estimator from the log table:

1. The next probable recharge time
2. The applications that may be launched or run in background within the time frame
3. The energy required by high priority applications

The Estimator thread runs in the background all the time and is ready to serve the Distributor when the information is seeked. The Estimator has a bidirectional access to the local database. Thus meaning, the Estimator reads the necessary data and writes back its current evaluation to keep the database updated. The most significant table that the Estimator accesses in write mode is the Priority table. Tables 3.4 and 3.5 show how the data is stored in a local database. Priority numbers are assigned in such a way that the lesser the number is the higher the priority is. That is, the

application or service with priority 1 will have the highest priority than the others. These tables are updated continously by the Estimator on time and date type basis. For example, during the day on a weekday, Email might have a higher priority than BBM. But on a weekend, the application may lose its priority completely.

PACKNAME	PRIORITY
com.bbm	6
com.facebook.katana	25
com.google.android.apps.docs	1
com.google.android.gm	2
com.whatsapp	30

Table 3.4: Estimator - Priority for Applications

SERVNAME	PRIORITY
Accelerometer	05
Bluetooth	04
Data	02
Location	03
WiFi	01

Table 3.5: Estimator - Priority for Services

We also have other tables that maintain user's priority for applications and services. When the user wants certain applications and services to have higher priority than others, they can set it up through the application's user interface and the main thread writes that information in the tables. The tables 3.6 and 3.7 show sample data. Users can also mention some applications or services that they do not care about them. They are recorded into whitelist. We will explain in detail how the users are allowed to set these values in Chapter 4. For now, these tables are in datebase to override the priorities set by our application to any application or service.

As we listed earlier, the Estimator analyses the available information and estimates the items (NRE, PAL and PER) as efficient and as accurate as possible. The following sections explain in detail how the information are derived and estimated.

3.3.1 Next Recharge Estimate (NRE)

Next Recharge Estimate (NRE) is a time quantity that identifies the next probable recharge start time. Its value can be anything in between 00:00 and 23:59. When the

PACKNAME	USERPRIORITY
com.bbm	3
com.facebook.katana	5
com.google.android.apps.docs	1
com.google.android.gm	2
com.whatsapp	4

Table 3.6: Estimator - User Priority for Applications

SERVNAME	USERPRIORITY
Accelerometer	5
Bluetooth	4
Data	02
Location	03
WiFi	01

Table 3.7: Estimator - User Priority for Services

user unplugs his/her smartphone from the charger, Estimator calculates NRE and fills out the open or latest record's NRE column in Table 3.1.

NRE is calculated in two methods and the maximum of those values is stored in the recharge statistics table of our local database. When we have two values for the next recharge time, as a conservative measure, we assume that the next recharge will happen in the later time (i.e. Max of NRE) rather than an earlier one.

In the first method, recharge start time (RST) and date type (DT) are considered. If the current day is a weekend, only the records related to the weekends are considered and if the current day is a weekday, only the records related to the weekdays are considered. Then, K-Means (where $K = 2$) operation is performed on the RST column of those records extracted. The simplest and cost efficient method to find NRE would be mean of all past RST or TAPR. But it will result in a single recharge start time that is inappropriate in most case. Most of the smartphones do not offer 24 hours of a battery lifetime under normal cicumstances. Considering other cluster algorithms will incur higher cost in memory and processing time that will result in higher energy consumption by our own solution. So, the optimum method to find NRE is to use K-Means. We fixed K as 2, because in smartphones battery lasts minimum of 8 hrs and average of more than 12 hours. So, ideally, users do not have a need to recharge their smartphones more than twice a day. Thus, we created 2 clusters of Recharge

Start Time. Forming two clusters results in two centroid values. That is, two usual recharge start time values. The one that comes first is refferred to as NRE_1 and the respective cluster is $Cluster_{NRE1}$. For example, if the current time is 12:30 and the usual recharge start times are 10:35 and 22:43, then our NRE_1 is 22:43.

In the second method, time after previous recharge (TAPR) and date type (DT) are considered. TAPR values of corresponding DT are extracted and then avaerage of all TAPR within $Cluster_{NRE}$ is calculated as $TAPR_{avg}$. Adding TAPRavg to the current time will give us NRE_2. As we mentioned earlier, the maximum of two values NRE_1 and NRE_2 is the Next Recharge Estimate (NRE). Figure 3.3 shows the complete work flow of the Estimator module.

3.3.2 Potential Application Launch

Given the current time and the next recharge estimate, our next goal is to find the applications that have high possibility of launching by the user or any background process. We also estimate the battery lifetime based on the current discharge rate. If battery lifetime is sooner than our NRE, then the time at which the battery dies is called $Time_{Life}$. Otherwise, NRE is called $Time_{Life}$. We extract the application names and NoH from the application usage table by querying application's launch time that lies in between current time and $Time_{Life}$ and keep the list in the main memory for an easy access to the Distributor module. We use the number of hits of the extracted application to update the priority table we mentioned in the previous section. Figure 3.4 shows the complete workflow of how we determine the probable or potential applications that may be launched by user or any background process. We also use the same technique to find potential service use (PSU) from the service usage table.

3.3.3 Probable Energy Required

The final item for the Estimator to find is energy required by those potential applications found in previous sections. From the potential application launch, we know the applications that have high probability of launching within the timed frame until next recharge or until battery lasts. From this applications usage table, we also know

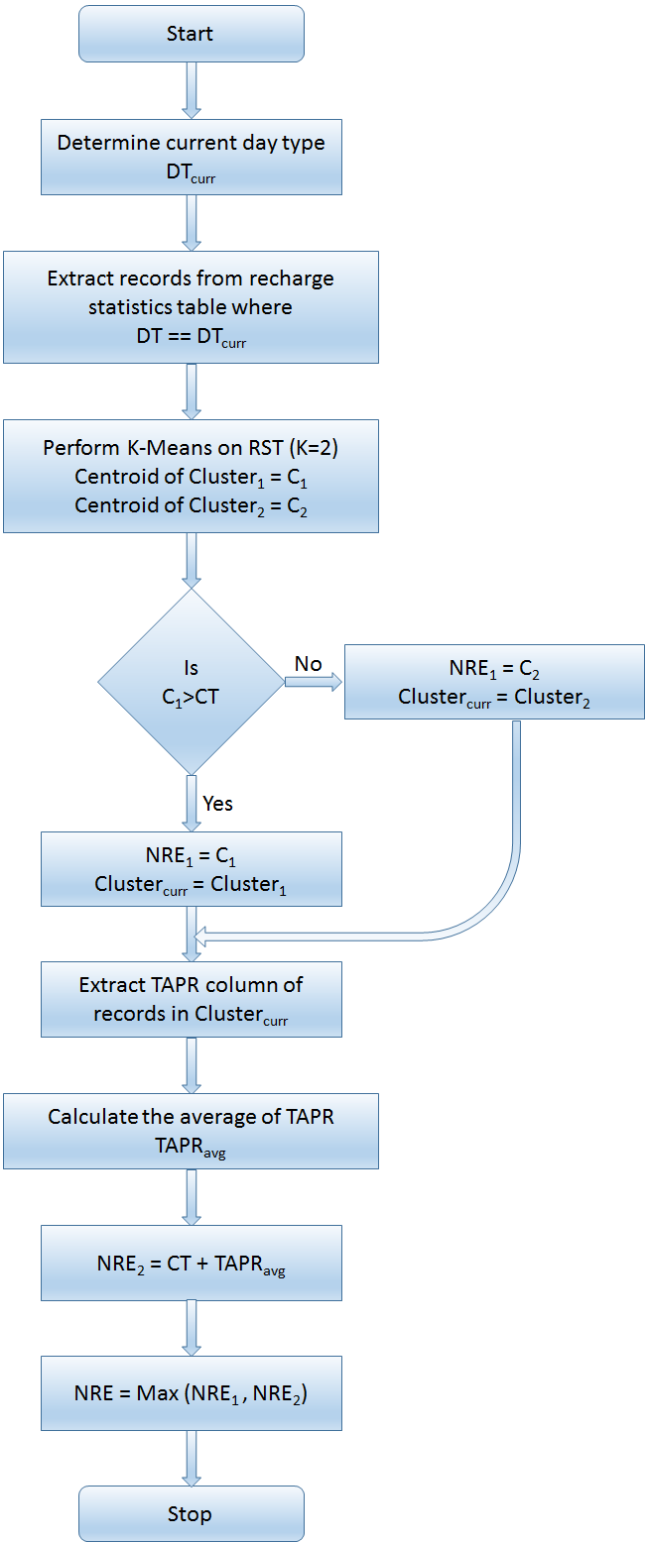


Figure 3.3: Next Recharge Estimate Flowchart

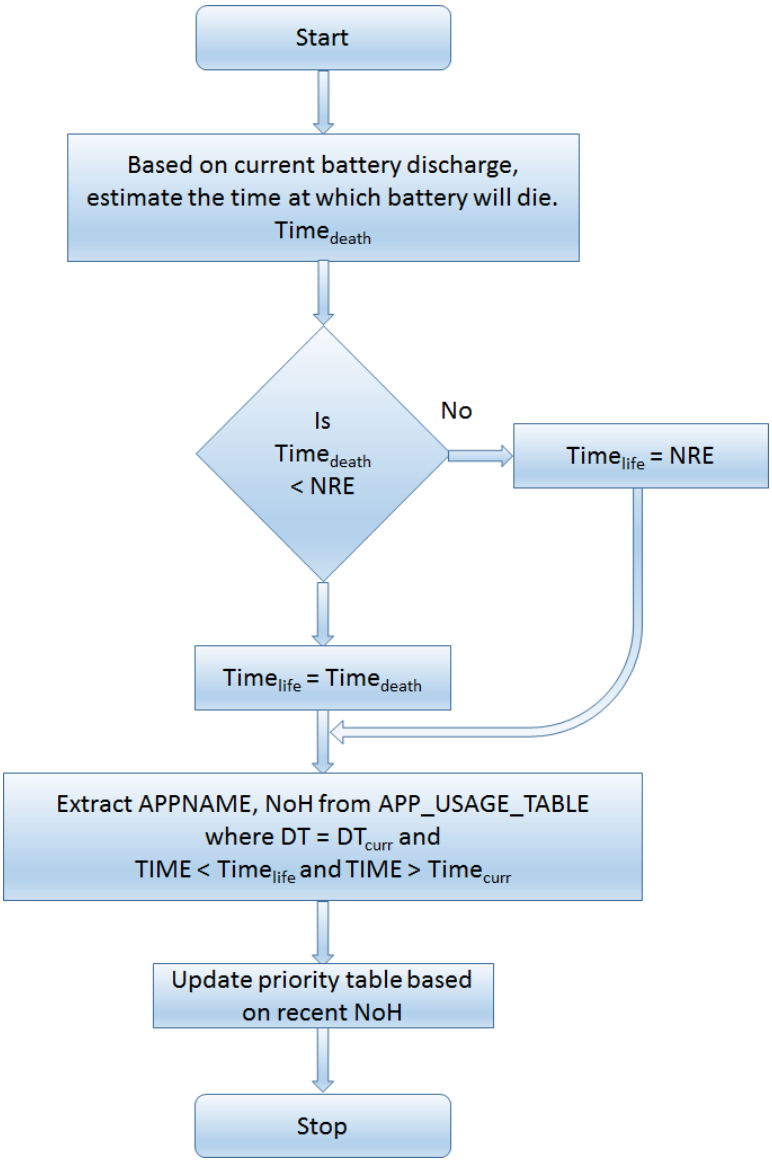


Figure 3.4: Potential Application Launch Flowchart

the energy consumption by those applications during those sessions. We copy the application names and corresponding energy consumption values to the main memory for easy access by the Distributor. We consolidate the records by application names and adding energy consumption by those applications. This value is called probable energy required (PER) We also use the same technique to find probable energy required by different services. Tables 3.8 and 3.9 show how the energy consumption by different potential applications are summed and consolidated for Probable Energy Required (PER). For example, there are two Gmail sessions, one with 2% and the other with 1% energy consumptions. So PER for Gmail is 3%. Like wise, Facebook has two sessions within the timeframe and the total PER for Facebook is 7%.

NO	PACKNAME	EC
1	com.google.android.gm	2
2	com.facebook.katana	5
3	com.bbm	2
4	com.facebook.katana	2
5	com.google.android.gm	1

Table 3.8: Enerygy Consumption by Potential Applications

PACKNAME	PER
com.bbm	2
com.facebook.katana	7
com.google.android.gm	3

Table 3.9: Estimator - Probable Energy Required (PER)

3.4 Distributor

The final module in our architecture is the Distributor which helps in analysing the Estimator's output and makes decisions on where to spend the energy and where it needs to be saved. The primary goal of our research work is to distribute the limited energy available in our smartphone as efficient as possible. Efficient in the sense, the energy should be saved from unnecessary background services and foreground applications and be spent on user's much needed tasks. Logger records each and every user activity on applications, services and recharge behaviour which in turn helps the

Estimator to predict usual recharge time and usual application usage time, duration and energy consumption. Based on the Estimator's prediction, the Distributor is capable of making the decision whether energy is sufficient until the battery lasts or until the next recharge time. As a first step, the Distributor calls the Estimator and initializes the Battery's current lifetime to $\text{Time}_{\text{Life}}$ and calls the operating system to know $\text{Energy}_{\text{available}}$, the total energy available at the moment. The Distributor also identifies the current app on forground launched by the user if the smartphone is in use or background applications that are running if the smartphone is idle. The second step for the Distributor is to calculate the energy required by potential applications and services that might be used in near future within $\text{Time}_{\text{Life}}$. Again, the Distributor calls the Estimator for a list of services and energy required by them. The sum of energy required by all those services is $\text{Energy}_{\text{SERVICES}}$. Likewise, the Distributor also calls the Estimator for a list of potential applications that may be launched in the near future within $\text{Time}_{\text{Life}}$ and that have a higher priority than the current application on foreground or background. As we described in section 3.3, the Estimator knows the energy required by the applications based on the day type and time of day. That is, energy required by the same applications and services may be different for different sessions. So, the total energy needed $\text{Energy}_{\text{Needed}}$ is nothing but the sum of energy required by potental services $\text{Energy}_{\text{SERVICES}}$ and the energy required by potential applications $\text{Energy}_{\text{PAL}}$. Then the Distributor verifies if $\text{Energy}_{\text{available}}$ is less than $\text{Energy}_{\text{Needed}}$. If it is true, the Distributor notifies the user to close the current app to save energy for more necessary applications and services in the future and closes the application if it is on background. If $\text{Energy}_{\text{Needed}}$ is less than $\text{Energy}_{\text{available}}$, the Distibutor goes to sleep doing nothing and wakes up at next interval. This whole process is repeated on an interval of 1 minute. This interval is set to 1 minuted based on the same logic we mentioned in the Logger section as per the article [10]. The Algorithm 1 shows the whole process of the Distibutor's decision making.

The Distributor does not switch on or off any service by itself. Our solution logs and tracks the service usage only for calculation purposes and to notify the user of his/her usual activities.

The next chapter "Implementation" explains in detail how this proposed solution is implemented on Android platform. The chapter also describes different key features

Algorithm 1 Distributor for Applications (On regular interval)

$Time_{\text{Life}} \leftarrow$ Battery's current lifetime (from Estimator)
$Energy_{\text{available}} \leftarrow$ Energy available (from operating system)
$App_{\text{curr}} \leftarrow$ Current foreground or background app

$SERVICES \leftarrow$ The services that may be used within $\text{Time}_{\text{Life}}$
$Energy_{\text{SERVICES}}$ 0
for each service S in $SERVICES$ **do**
  $Energy_{\text{S}} \leftarrow$ Energy required by S
  $Energy_{\text{SERVICES}} = Energy_{\text{SERVICES}} + Energy_{\text{S}}$
end for

$PAL \leftarrow$ The applications that have possibility of launch within $Time_{\text{Life}}$ and that have priorities higher than App_{curr}
$Energy_{\text{PAL}}$ 0
for each app A in PAL **do**
  $Energy_{\text{A}} \leftarrow$ Energy required by A
  $Energy_{\text{PAL}} = Energy_{\text{PAL}} + Energy_{\text{A}}$
end for

$Energy_{\text{Needed}} = Energy_{\text{SERVICES}} + Energy_{\text{PAL}}$

if $Energy_{\text{available}} < Energy_{\text{Needed}}$ **then**
  if App_{curr} is Foreground **then**
    Notify user that he/she needs to close the app to save energy for much needed apps in future
  else
    Close the app immediately.
  end if
end if

that are used in our implementation and deployment.

Chapter 4

Implementation

Our proposed solution has been implemented on the Android platform using Java as the primary programming language and XML for user interface. As we mentioned in the section 1.2.1, due to various factors that include a huge user base and market share by sales, powerful sdk for development, flexibility and ease of access to operating system calls, We inclined towards Android and we successfully implemented, deployed and evaluated our solution. In this chapter, we discuss the solution in two parts - user interface and the key concepts that were used behind the scenes in our implementation. We also explain the limitations in our implementation and the direction to overcome the limitations in future.

The development environment used in our implementation is as follows:

Programming Language(s)	Java
User Interface	XML
Primary SDK	Android 4.2.2 (API 17)
Other Libraries	None
IDE	Eclipse (Juno)
Device(s) used	Nexus 4 and different emulators
Development Platform	Windows 8.1
Deployment Platform	Android 4.2 (Jelly Bean)

Table 4.1: Development Environment

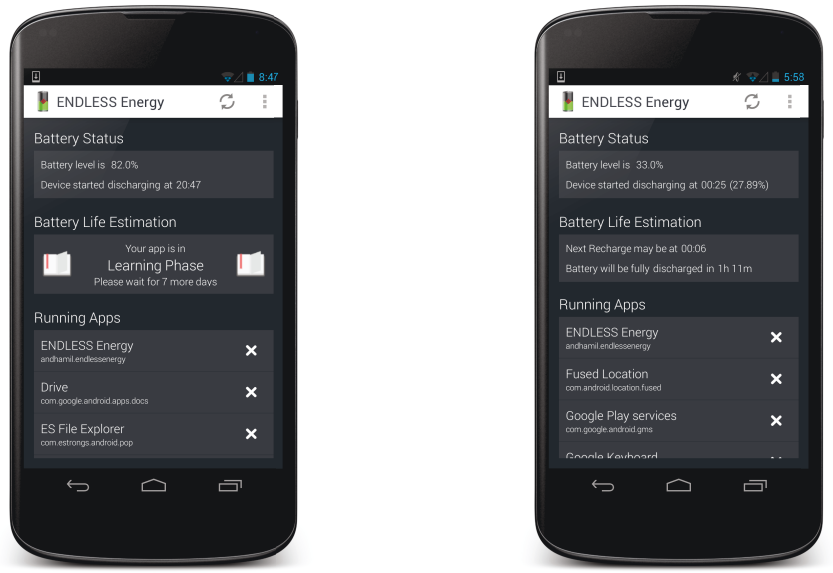
4.1 User Interface

The Logger, Estimator and Distributor modules have been integrated into one single application called ENDLESS which is the title of our thesis - Energy Distribution through Lifetime Estimation and Smartphone usage patterns. We implemented our solution in a more user centric approach that is intuitive and interactive. Users can see every element that the application logs, estimates and controls other applications through our Energy Distributor. The application also offers users the total control of

the data collected and managed. In this section, we describe each and every screen that the users able to view and manage. In the Android perspective, a screen is called "Activity" and a portion of the screen that is reused in multiple circumstances is called "Fragment". An activity is like a container that may or may not embed one or more fragments. Throughout this section, we will use the terms "Activity" for screens and a wide range of Android terminologies for various user interface controls. To understand these terms, we encourage you to go through the Android developer website [7] before continuing with the following sections. The user interface controls like labels, buttons, lists, etc may also be placed directly on the activity.

4.1.1 Home

When the application is installed and launched for the first time, the home activity would look like Figure 4.1(a). All the static information that are not related to our Logger are displayed for the first seven days. As we have discussed earlier, the first seven days are fixed as the learning phase in which the Logger logs the neccessary data in a local database. So, the estimation values like Next Recharge Estimate (NRE), probable battery lifetime, etc are not displayed until the application goes into the activated phase. In the activated phase, the home activity is filled completely with all the basic information and users have access to other activities. Figure 4.1(b) shows the home activity in the Activated phase. The home activity contains three parts of information - Battery Status, Estimation and Running Apps. Battery status contains the current battery level and the time when the user plugged his/her smartphone into the charger or unplugged it from the charger. It also shows the charging rate in percentage per hour if the smartphone is in charge mode and otherwise it shows the discharging rate. All this information is directly pulled from the Android operating system. Battery status information is hown to the user since day one of installation. That is, battery status is available in both the learning and activated phases. Estimation shows Next Recharge Estimate that is calculated by the Estimator module based on the logged data to date. It also shows how long the battery will last in the discharging mode and how long the battery will take to reach 100% if it is in the charging mode. The last section shows currently running apps and with the user's direction, ENDLESS application is capable of placing a process



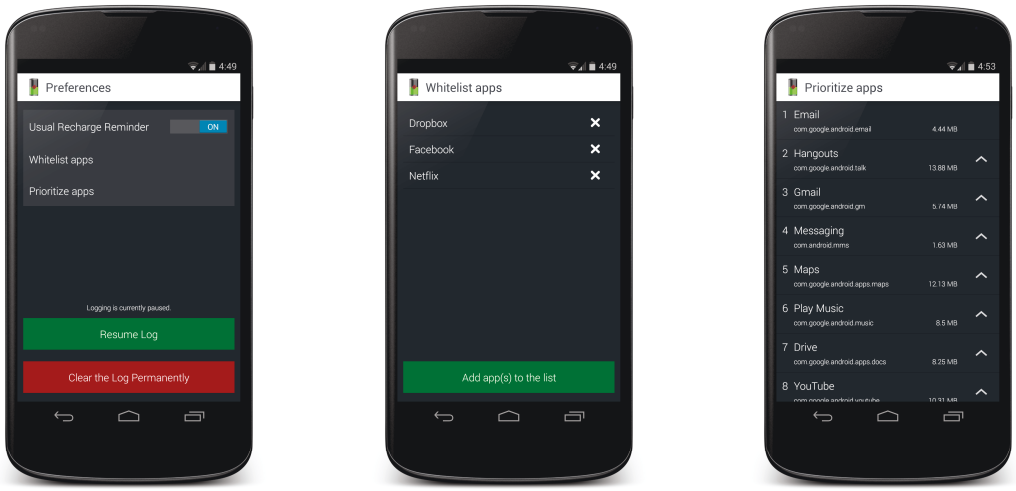
(a) Learning Phase (b) Activated Phase

Figure 4.1: Home Activity

termination request to Android for that particular app. Running apps information is also available in both the learning and activated phases.

4.1.2 Preferences

The Preferences activity lets the user set his/her own application preferences. The activity is accessible from the Home activity's action bar. As shown in Figure 4.2(a), there are five options available for the user within the Preferences activity. First, the usual recharge reminder that reminds the user at Next Recharge Estimate (NRE) time. The reminder is sent to the user from the application using Android Notifications [14]. From the preferences activity, the users can turn on or turn off this feature. By turning it off, the user will no longer receive these notifications. This is a small feature that helps the user maintain his/her recharge cycle consistent. The second option is the Whitelist Apps that lets the user add his/her favourite apps that should not be interrupted during any circumstance by our ENDLESS applications. Any application listed in this category will not be closed regardless of the battery level. Figure 4.2(b) shows how the apps are added to Whitelist. The third one is



(a) Various Options (b) Whitelist Apps (c) Prioritize Apps

Figure 4.2: Preferences Activities

to prioritize apps. Apart from our ENDLESS application's prioritization by tracking the usage, users are allowed to set their own priorities for the apps. The user's priority overrides our calculated priorities. Furthermore, the apps that are added to Whitelist will not be shown here for user prioritization. This is due to the fact that all the apps that are whitelisted will be priority 1 automatically. Figure 4.2(c) shows the prioritization of apps by user. When the user taps on the up arrow nearby each application, the app would move up gaining a higher priority. For example if the app has a priority of 10 and the user taps on its up arrow, it would result in priority 9. The app that was in priority 9 would move down to 10. The fourth option is to pause and resume logging. For example, if the user goes on a vacation, his/her whole usage and recharge cycle is going to be different and he/she might want to discard those behaviours. In that case, the user can pause the logging and resume it when he/she wishes. The last option is to clear the log completely and start over. When the user clears the log, the application erases all the data that has been logged and turns into the learning phase considering it as day 1 after installation.

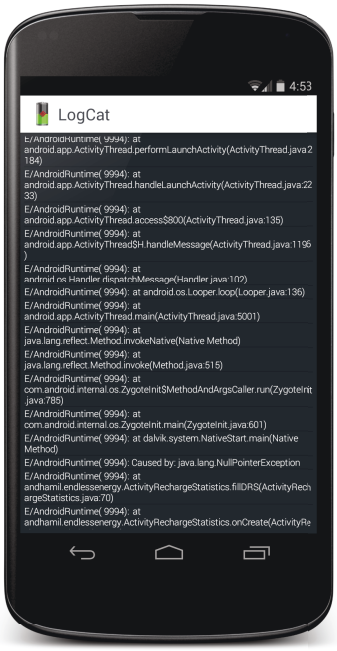


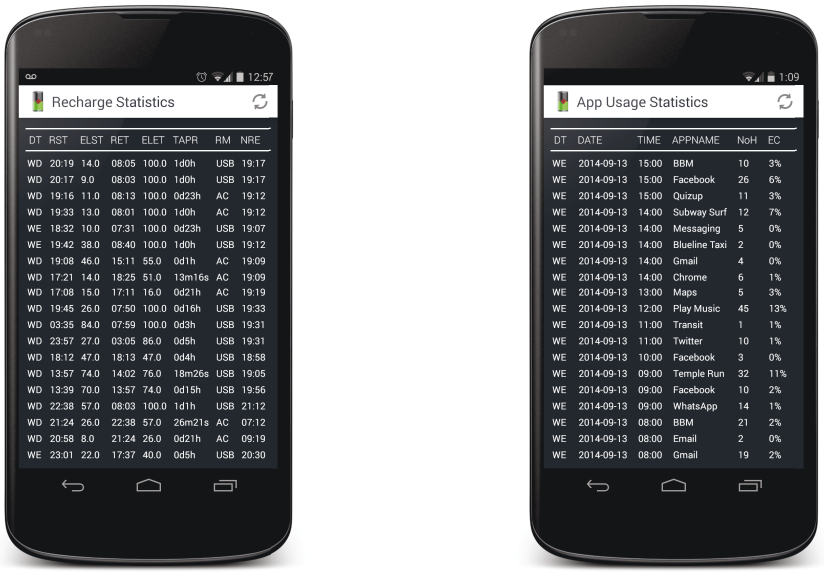
Figure 4.3: LogCat Activity

4.1.3 LogCat

LogCat is a debugging tool that is built into the Android Operating System. It helps to collect and display the data of what is really happening in the system. Our LogCat activity invokes this LogCat tool and displays the output for the user. This is helpful for both the user and us to understand the current state of the smartphone especially running processes and their resource utilization. For further reading on LogCat, please visit the link in [13]. Figure 4.3 shows a screenshot of LogCat activity.

4.1.4 Logger

The Logger Module contains two activities namely Recharge Statistics and App Usage. Both of these activities are display panels for the Logger database. All the data that is stored in database by the Logger module are displayed to the user in these activities. Figures 4.4(a) and 4.4(b) show the complete log of recharge statistics and app usage behaviour. They are read only and the user has no ability to edit or delete any particular record. However, users can export the log to their computer or



(a) Recharge Statistics (b) Application Usage

Figure 4.4: Logger Activities

any other device using Email or any other sharing application. These activities are helpful for the user to know his/her own smartphone usage patterns and helpful for us to debug.

4.1.5 Estimator

The Estimator acts as an interface between the Logger and Distributor. Therefore, the Estimator does not have its own user interface screen. The module pulls the data from the database, then calculates and estimates certain parameters and provide results to the Distributor. However, the Next Recharge Estimate (NRE) is pulled from the database and displayed it on the Home screen as shown in Figure 4.1(b).

4.1.6 Distributor

The Distributor module helps to decide if the energy is sufficient enough to last until the battery drains out or the next recharge happens. It makes a call to the Estimator to find the next recharge time, the potential application launch, their priority and their energy consumption. Based on the estimated values, the Distributor verifies if

the foreground application is of higher priority to consume the energy. When there is enough energy left before the next recharge happens or battery drains out, the Distributor does nothing but sleeps for a minute and wakes up again. There is no user interface in this case. However, if there is not much energy left, the Distributor finds out if the current application is on foreground or background. In the case the application is on background, it is closed immediately and the user is notified through the Android notification system as shown in Figure 4.5(a). This message does not need any input or confirmation from the user. But, if the application is on the foreground,the Distributor sends a notification to the user that it needs his/her attention as shown in Figure 4.5(b). When the user taps on this notification, an activity is popped up with detailed information as shown in the Figure 4.6(a). This information contains the name of the application that must be closed in order to save energy for the future. The activity contains two buttons - "Yes, please!" and "No, thanks!". When the "Yes, please!" button is tapped, ENDLESS energy application sends a request to the Android operating system to terminate the corresponding application. When the application is killed completely, the user is notified again that the termination is successful as shown in Figure 4.6(b). But, if the user decides to keep the app open and taps on "No, thanks!", the Distributor closes the current activity, sleeps for a minute and continues to run in the background.

4.2 Behind the Screen

In this section, we explain the key concepts and techniques that are used in this application. Apart from application's interactions with the user, there is a lot of background processes and activities running to provide various functionalities to the solution and ensure its accuracy and efficiency. Such concepts are discussed in detail in the following subtopics.

4.2.1 Shared Preferences

Shared Preferences is an Android facility provided by Google through public Java-class SharedPreferences [16]. The third party Android developers make use of this class to store and retrieve key value pairs that may be shared across various applications or activities within a single application. The Shared Preferences space

(a) Background Application (b) Foreground Application

Figure 4.5: Notification for Application Kill

(a) User Prompt to Kill (b) App Kill Confirmation

Figure 4.6: Foreground Application Kill

should be created either as public or private. Any space that is created private will be accessible only by the application that created the space. On the other hand, the public space is accessible by any aplication that has access. The "shared preferences" is created and accessed through Android public api public abstract SharedPreferences getSharedPreferences (String name, int mode) in which the first parameter "name" identifies the preferences file that the developer is trying to access and "mode" identifies the the access level of that particular space. Mode can be MODE_PRIVATE, MODE_WORLD_READABLE, MODE_WORLD_WRITABLE or MODE_MULTI_PROCESS. As we do not have a need to share our data with any other process or application, our shared preferences file is always private. We also have a subclass called Editor [17] that helps us get an access to store new value to a key or modify the value of an existing key.

Listing 4.1: Create or get Shared Preferences and a corresponding Editor objects

```
SharedPreferences preferences =
getSharedPreferences("andhamil_endlessenergy_preferences",
Context.MODE_PRIVATE);
SharedPreferences.Editor prefEditor = preferences.edit();
```

In our implementation, we extensively use this Shared Preferences to access and modify various data such as the application installation date, user's preference on usual recharge reminder, etc. The data can be stored to and retireved from Shared preferences in any Java supporting data type. That is it can be either string, integer, boolean, float or long. To achieve this, the Shared Preferences class and its nested member class Editor provide various public methods like putBoolean, getBoolean, putString, getString, etc. The following code snippets show an example of how we store and retrieve the application installation date as a string.

Listing 4.2: Retireve application installation date

```
String strInstalledDate = preferences.getString("INSTALLED_DATE", "");
```

Listing 4.3: Store current date as application installation date

```
prefEditor.putString("INSTALLED_DATE",
dateFormat.format(dateCurrent));
```

```
prefEditor.commit();
```

4.2.2 SQLite Database

Android has an built-in SQLite database management system that can be used by the operating system and various applications to create and access their data in tables of rows and columns. This SQLite database can be used in the same way as any one use on a desktop machine. Android provides a public class SQLiteDatabase [19] and a public helper class SQLiteOpenHelper [20] for developer to use all the database functionilities like creating a database, creating tables, adding records and executeing any sql command. In most cases, we do not need the class SQLiteDatabase. All the functionalities that we need can be achieved by deriving SQLiteOpenHelper class. In our application, we implemented the class DBHelper by deriving SQLiteHelper. This class contains the database version, table names and column names as its member variables. When there's a change in the database or table structure, we need to increase the database version that helps reset the data. The following snippet shows the declaration of class and few of its member variables. We have implemented many tables for various purposes. For the purpose of understanding how database is accessed, we explain through the table called "APPPRIORITY" that helps to store user's prioirty values for various applications. In the subsequent paragraph we will discuss how this table is created and used.

Listing 4.4: DBHelper class to access SQLite database of our application

```
public class DBHelper extends SQLiteOpenHelper
{
        private static DBHelper mInstance = null;

        private static final int DATABASE_VERSION = 1;
        private static final String DATABASE_NAME = "DB_ENDLESS_ENERGY";

        private static final String TABLE_APP_PRIORITY =
            "TABLE_APP_PRIORITY";
        private static final String COLUMN_APP_PRIORITY_ID = "ID";
```

```
        private static final String COLUMN_APP_PRIORITY_PACKAGE_NAME =
            "PACKAGE_NAME";
        private static final String COLUMN_APP_PRIORITY_PRIORITY =
            "PRIORITY";
}
```

SQLite database like any other database works on the lock and use method. That is two threads cannot write into the same database at the same time. It is not a good practice to create a new object everytime we need to access the database. So we implemented a method to get current instance of the class as shown below.

Listing 4.5: getInstance() method to return current instance of DBHelper class

```
public static DBHelper getInstance(Context ctx)
{
        if (mInstance == null)
                mInstance = new DBHelper(ctx.getApplicationContext());
        return mInstance;
}
```

We create all the tables by overriding onCreate method of SQLiteOpenHelper class. This method is executed only once until there is a change in the database version. Creating and accessing the tables is very straight foward by executing respective SQL commands. To create the APPPRIORITY table, we execute CREATE TABLE sql command with the column names and their data types as shown below.

Listing 4.6: onCreate() method to do one time activities like creating tables

```
public void onCreate(SQLiteDatabase db)
{
        String CREATE_TABLE_APP_PRIORITY = "CREATE TABLE " +
            TABLE_APP_PRIORITY
                        + "(" +
                        /* 00 */COLUMN_APP_PRIORITY_ID
                        + " INTEGER PRIMARY KEY AUTOINCREMENT," +
                        /* 01 */COLUMN_APP_PRIORITY_PACKAGE_NAME + " TEXT
                            UNIQUE," +
```

```
                    /* 02 */COLUMN_APP_PRIORITY_PRIORITY + " INTEGER" +
                        ")";
        db.execSQL(CREATE_TABLE_APP_PRIORITY);
}
```

After the creation of tables, we need some public methods to be accessible by various classes of the application to add, update and delete records in any of the tables we created. To add an app to the priority list, we first get SQLiteDatabase object in read mode to find the lowest priority app. We get another SQliteDatabase object in write mode. We increase the prioirty number by 1 and add the record by calling the insert method. We use ContentValues a key value pair for the field names and values of the current record. Likewise, to retrieve the priority of an app, we execute SELECT statement and use the cursor to traverse across the return record. The following code snippets show how the priority is added and retrieved by respective methods.

Listing 4.7: addAppPriority() method to add an app to the priority list at the end

```
public boolean addAppPriority(String strPackageName)
{
        boolean bReturn = false;
        int nPriority = 0;
        String countQuery = "SELECT MAX(" + COLUMN_APP_PRIORITY_PRIORITY
                        + ") FROM " + TABLE_APP_PRIORITY;
        SQLiteDatabase dbRead = this.getReadableDatabase();
        Cursor cursorRead = dbRead.rawQuery(countQuery, null);
        if (cursorRead.moveToFirst())
                nPriority = cursorRead.getInt(0) + 1;
        if (!cursorRead.isClosed())
                cursorRead.close();
        if (dbRead.isOpen())
                dbRead.close();
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(COLUMN_APP_PRIORITY_PACKAGE_NAME, strPackageName);
```

```
        values.put(COLUMN_APP_PRIORITY_PRIORITY, nPriority);
        long lResult = db.insert(TABLE_APP_PRIORITY, null, values);
        if (lResult >= 0)
                bReturn = true;
        if (db.isOpen())
                db.close();
        return bReturn;
}
```

Listing 4.8: getAppPriority() method to retrieve the priority value of an app

```
public int getAppPriority(String strPackageName)
{
        int nReturn = -1;
        String strQuery = "SELECT " + COLUMN_APP_PRIORITY_PRIORITY + " FROM
            "
                        + TABLE_APP_PRIORITY + " WHERE "
                        + COLUMN_APP_PRIORITY_PACKAGE_NAME + "=" + "'" +
                            strPackageName
                        + "';";
        SQLiteDatabase db = this.getReadableDatabase();
        Cursor cursor = db.rawQuery(strQuery, null);
        if (cursor.moveToFirst())
                nReturn = cursor.getInt(0);
        if (!cursor.isClosed())
                cursor.close();
        if (db.isOpen())
                db.close();
        return nReturn;
}
```

As these public methods are defined in DBHElper class, they can be accessed through any class in our application. As in our example App prioirty, the Prioritize Apps activity calls these methods. When a user clicks on the up arrow next to an app, the current app moves up and the above app moves down. That is, the two apps

exchange their priority values. The following code snippet is an implementation of on click event that is triggered by a user's tap on the arrow button. In this method, we get priorities from the database and write back with new values.

Listing 4.9: onClick() to modify the pripority values in APP_PRIORITY table

```
public void onClick(View v)
{
        DBHelper dbHelper = DBHelper.getInstance(mContext);

        int nCurrentAppPriority = dbHelper.getAppPriority(
                Utils.malInstalledApps.get(position).getPackageName());
        int nUpstairsAppPriority = dbHelper.getAppPriority(
                Utils.malInstalledApps.get(position - 1).getPackageName());
        dbHelper.updateAppPriority(strCurrentPackageName,
            nUpstairsAppPriority);
        dbHelper.updateAppPriority(strUpstairsPackageName,
            nCurrentAppPriority);
        Utils.malInstalledApps.get(position)
                .setAppPriority(nUpstairsAppPriority);
        Utils.malInstalledApps.get(position - 1)
                .setAppPriority(nCurrentAppPriority);

        Utils.sortInstalledAppsByPriority();
        notifyDataSetChanged();
}
```

4.2.3 Broadcast Receiver

Broadcast Receivers is a methodology to receive intents sent by either the Android operating system or other applications installed on the device. To receive any intent, the application should register for a corresponding event either dynamically in the Java code or statically in manifest xml file. After the registration, application must derive the class"BroadcastReceiver" and override "onReceive" method to implement the event handler. In our application, we need to receive two broadcast

messages - one is to be received when the user plugs-in or unplugs his/her smartphone from the charger and the other is to be received when there is a change in the battery level. Android sends out broadcast message for every 1 percent change in the battery level. As shown in the following snippet, we registered for these broadcast messages at the end of "AndroidManifest.xml". ACTION_POWER_CONNECTED and ACTION_POWER_DISCONNECTED are intent names for charger plug in and charger plug out respectively. ACTION_BATTERY_CHANGED is the intent name for battery level change. These registrations are intended to receive broadcast message from the Android operating system. However, the other applications installed on the device may also send these broadcast message if they have permission. As per our implementation, the Android sends the message to "EnReceiverOnPlug" derived class' onReceive method for the charger plug in and plug out event whereas the android sends the message to "EnReceiverOnBatterChange" derived class' onReceive method for the battery change event.

Listing 4.10: Broadcast Receiver registration on AndroidManifext.xml

```
<receiver
android:name="andhamil.endlessenergy.custom.EnReceiverOnPlug" >
    <intent-filter>
        <action
            android:name="android.intent.action.ACTION_POWER_CONNECTED"
            />
        <action
            android:name="android.intent.action.ACTION_POWER_DISCONNECTED"
            />
    </intent-filter>
</receiver>
<receiver
    android:name="andhamil.endlessenergy.custom.EnReceiverOnBatterChange"
    >
    <intent-filter>
```

```
        <action
            android:name="android.intent.action.ACTION_BATTERY_CHANGED"
            />
    </intent-filter>
</receiver>
```

Both the derived classes “EnReceiverOnPlug” and “EnReceiverOnBatterChange” override onReceive method to implement the necessary handler. The following code snippets show the implementation of both receivers. When the user plugs in or plugs out his/her smartphone to or from the charger, we record the recharge statistic details like start time, end time, battery level at the start and finish times, etc. in the database and update the user interface of home activity. When the “EnReceiverOnBatteryChange” class receives a battery change broadcast message, the “onReceive method” records the charging or discharging rate and updates the user interface of home activity.

Listing 4.11: BroadcastReceiver for charger plug-in and plug-out events

```
public class EnReceiverOnPlug extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        UtilsDB.recordDeviceRechargeStatistics(context);
        Utils.recordChargeOrDischargeStartTime(context);
        ActivityHome.updateBatteryInfo(context);
    }
}
```

Listing 4.12: BroadcastReceiver for battery level change event

```
public class EnReceiverOnBatteryChange extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
```

```
    {
        Utils.recordChargeOrDischargeRate(context);
        ActivityHome.updateBatteryInfo(context);
    }
}
```

The "Broadcast Receivers" methodology is very helpful in implementing handlers for various events occurring in the device hardware especially the battery. As shown above, we are able to detect and handle the battery status change in a few lines of Java code. For a detailed reading on Broadcast receivers, please visit the link in [4].

4.3 Limitations

During our implementation, we had very few notable limitations regarding technology and access level. In Android, the most sophisticated development platform is Java. The applications developed using Java run on top of JVM (Java Virtual Machine) which is a part of both old Android's Dalvik system and new ART (Android Run Time) system. So, our application itself consumes a noticeable energy. But if our solution is embedded within the operating system and just the user interface runs on JVM, we believe there will be a reduction in energy consumption. However, within our scope it is difficult to implement our solution in the operating system level. The another limitation we had was accessing energy consumption by different applications installed. For calculating individual application energy consumption, we followed the guidelines and documents released by Google [15]. Until the Android 4.2 Jelly Bean, the APIs were public and available for third party developers. But, since the Android 4.3 Kit Kat, Google made them private and so, to deploy our solution on a device, it is necessary that our ENDLESS Energy application has a root access granted by the user. As we evaluated our solution on Nexus 4 with Android 4.1.2 operating system, we did not have the necessity for device rooting.

Chapter 5

Evaluation

This chapter presents the evaluation of our proposed system by comparing smartphone battery lifetime with all the solutions we discussed in Chapter 2. We also compared various features of the solutions. For example, a few solutions turn off the mobile data to save battery and some solutions increase the sync interval. Then, we tested the battery consumption by each application and compared with eachother. We evaluted ENDLESS for the accuracy of its next recharge estimate and also compared the number of recharge attempts against Llama [28]. We present the detailed results in the following sections.

5.1 Battery Lifetime Comparison

We recorded battery life time when the smartphone is managed by our solution - ENDLESS and compare the same with all other solutions we discussed earlier. To be consistent and as accurate as possible, we used the same device (Google LG Nexus 4) running the same operating system (Android 4.1.2). The battery lifetime is calculated as time taken by the battery to reach 1% energy level from 100%. Also we developed a small tool that records the battery lifetime. When the battery reaches 0%, the system will completely shut down and it will become impossible to record the battery drain time accurately. So we use the next nearest integer 1% to calculate our battery life time. For accuracy in our evaluation, we allowed the battery to drain out completely before next recharge. To be fair to all the solutions under our evaluation, the smartphone usage should not vary to a greater extent. So we tracked a single day smartphone usage and automated it to occur daily during our testing period. That is, we automated Email, SMS and Calls and fixed some time period for the game and social network applications that cannot be automated. During both the automated and manual testing, we kept the display on. Table 5.1 shows values and times for our fixed applications. The Email, SMS and Phone calls were automated and the others

were manually executed.

Application	Value
Email	20
SMS	10
Phone Call	5 with duration 5 Minutes and 5 with duration 15 Minutes
Facebook	5 sessions of 10 minutes each
Twitter	3 sessions of 10 minutes each
Temple Run	1 session of 20 minutes
Subway Surf	2 sessions of 15 minutes each
BBM	5 sessions of 5 minutes each
WhatsApp	5 sessions of 10 minutes each
YouTube	1 session of 30 minutes

Table 5.1: Fixed Parameters for Battery Lifetime Comparison

The data from our testing results is shown in Table 5.2. We tested each application individually for two weeks and compared with each other. When one application is managing the smartphone, the others are completely uninstalled and have no influence over the device. That is, at any point of time, only one of these applications is installed. All the values in the table are in minutes. For example on the day 1 (Sunday) the default battery lifetime was 1024 minutes (17 hours and 4 minutes). The first column represents default battery lifetime (without any battery saving application installed). The second column represents the battery lifetime when the smartphone is managed by Battery Doctor application. And the last column represents battery lifetime when the device is managed by our proposed solution ENDLESS. We also calculated the average of all two week values and show in the last two rows. The last but one row shows the average battery lifetime in minutes derived from the applications. The last row shows the same average in hours and minutes for good readability. From the average, we found that the battery lifetime of our smartphone Nexus 4 was increased significantly by our proposed solution ENDLESS. When we compared each other, the ENDLESS battery lifetime is higher than any other solution and the default battery lifetime. And also, the other applications yield battery lifetime higher than the default. Figure 5.1 shows the line chart of Table 5.2. For the whole two weeks period, the battery lifetime by ENDLESS is always higher than any other solution and the default battery lifetime is always lower than any other solution. We also found that the battery lifetime on weekends is higher than the weekdays. So,

having a dynamic mechanism to save energy will yield better results. This chart lets us conclude that these solutions extend battery lifetime significantly and ENDLESS is better than any of the solutions we discussed. Figure 5.2 shows the column chart of the average battery lifetime offered by the solution in two weeks.

	Default	Battery Doctor	Juice Defender	Go Power Master	Battery Guru	Battery Drain Analyzer	ENDLESS
	in mins	in mins	in mins	in mins	in mins	in mins	in mins
Sun	1024	1106	1144	1044	1163	1156	1191
Mon	996	1102	1103	999	1124	1117	1129
Tue	938	1013	1069	967	1086	1065	1105
Wed	969	993	1067	1006	1090	1073	1140
Thu	994	1026	1054	1011	1073	1054	1093
Fri	1040	1124	1108	1039	1131	1129	1134
Sat	1048	1168	1135	1083	1141	1135	1201
Sun	1047	1124	1164	1072	1187	1171	1203
Mon	965	1078	1115	978	1132	1136	1147
Tue	979	986	1035	982	1064	1044	1089
Wed	962	1011	1087	979	1102	1092	1136
Thu	980	1008	1028	992	1061	1046	1082
Fri	1028	1112	1119	1048	1126	1120	1143
Sat	1036	1176	1146	1062	1157	1153	1217
Avg mins	1000.43	1073.36	1098.14	1018.71	1116.93	1106.50	1143.57
Avg hh:mm	16:40	17:53	18:18	16:58	18:36	18:26	19:03

Table 5.2: Battery Lifetime Comparison

5.1.1 Static Solutions Vs. Dynamic Solutions

The applications that we tested shall be listed in 3 broad categories - Default, Static and Dynamic. As we discussed in Chapter 2, the applications - Battery Doctor, Juice Defender and Go Power Master fall under Static Solution category. Snapdragon BatteryGuru, Battery Drain Analyzer and ENDLESS applications are Dynamic Solutions. As our solution ENDLESS yields different results based on the user's smartphone usage pattern and recharge behavior, it is categorized as Dynamic solution. There is no similarity between ENDLESS and the other static solutions we

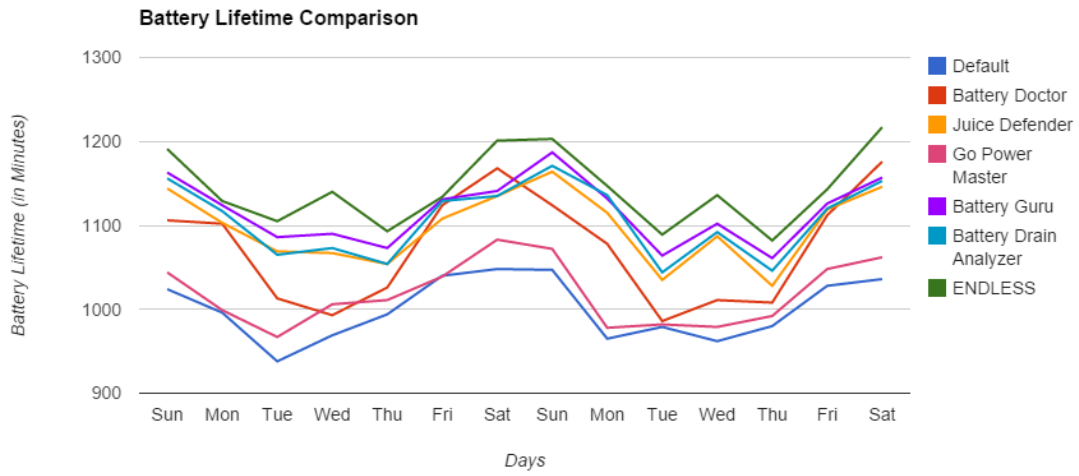


Figure 5.1: Battery Lifetime Comparison

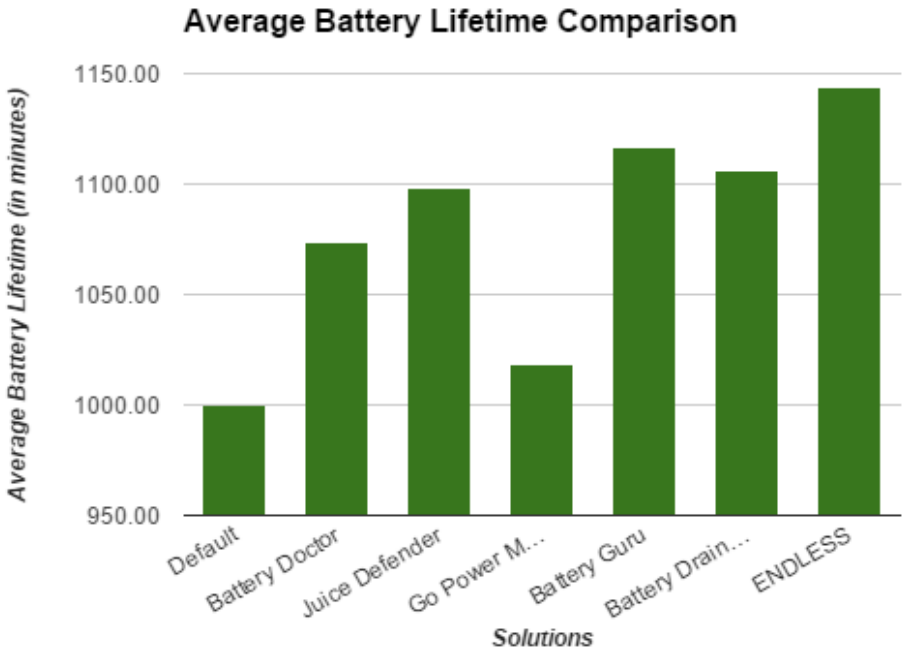


Figure 5.2: Average Battery Lifetime Comparison

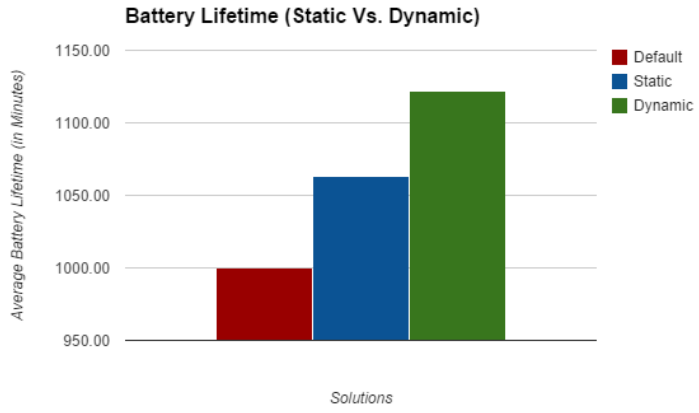


Figure 5.3: Battery Lifetime (Static Solutions Vs. Dynamic Solutions)

discussed except that both try to achieve the same goal - extending battery lifetime. From Table 5.2, we categorized these applications and derieved Table 5.3. The table shows the average battery lifetime of our Nexus 4 device yielded by default, static and dynamic solutions in the respective columns. Figure 5.3 shows the pictorial representation of this derived data. From the chart, it becomes obvious that the battery lifetime offered by static solution are higher than the default battery lifetime but the battery lifetime offered by dynamic solutions are significantly higher than that offered by static solutions. So, we are convinced that extending battery lifetime through tracking user's usage activities and recharge behaviour yields better results. For a clear understanding of ENDLESS' performance over the other solutions within the dynamic category, we created the chart shown in figure 5.4. As we previously mentioned all dynamic solutions offer longer battery life time than the static solutions and also within the dynamic category, ENDLESS offer longer battery than the other solutions.

	Default	Static	Dynamic
Average Battery Lifetime (in mins.)	1000.43	1063.40	1122.33
Average Battery Lifetime (in hh:mm)	16:40	17:43	18:42

Table 5.3: Battery Lifetime (Static Solutions Vs. Dynamic Solutions)

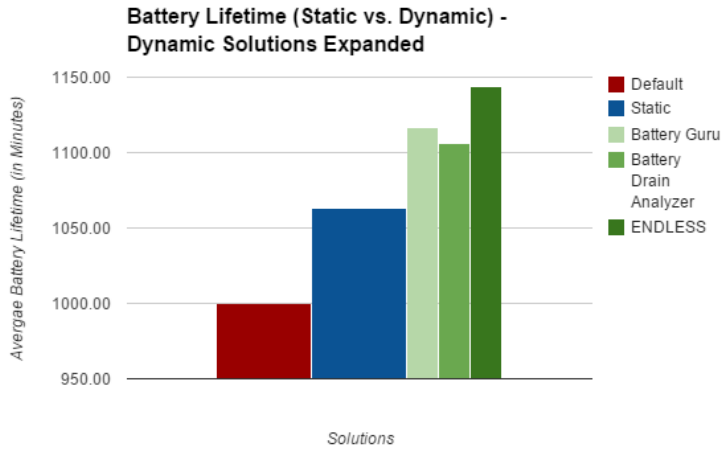


Figure 5.4: Battery Lifetime (Static Vs. Dynamic) - Dynamic Solutions Expanded

5.1.2 Difference within two consecutive Weeks

The battery life time estimation and extension over time can be well understood by analyzing the progress of these applications. Our ENDLESS solution records the data on Weekend and Weekday basis. From Figure 5.1, we clearly see that the battery lifetime on weekends are higher than that on weekdays. So, classifying the records into weekdays and weekends is really helpful to efficiently extend the battery life. We compared week 1 data with week 2 data as shown in Figure 5.5. Each block represents the difference in values of a particular day on week 1 and week 2. For example, The blue blocks shows difference in battery lifetime on sundays. When we measure the whole week difference, the default and static solutions differ by more than 120 minutes (2 hours). But the dynamic solutions differ from 80 to 110 minutes which is significantly lower than the static solutions. Therefore we understand that tracking users usage and recharge behaviour yields consistency in the results. The difference measured by ENDLESS is lesser than the other two dynamic solutions we discussed. This is because, we classify all our logs into weekends and weekdays to better understand and estimate energy consumption on various days and routines. If we classify the data into seven specific days, the results may be even better but it will result in higher processing and delay that would lead to higher energy consumption by the solution itself. So, for the scope of this thesis, we limited ourselves to two categories - weekdays and weekends. In the future, this shall be extended to specific

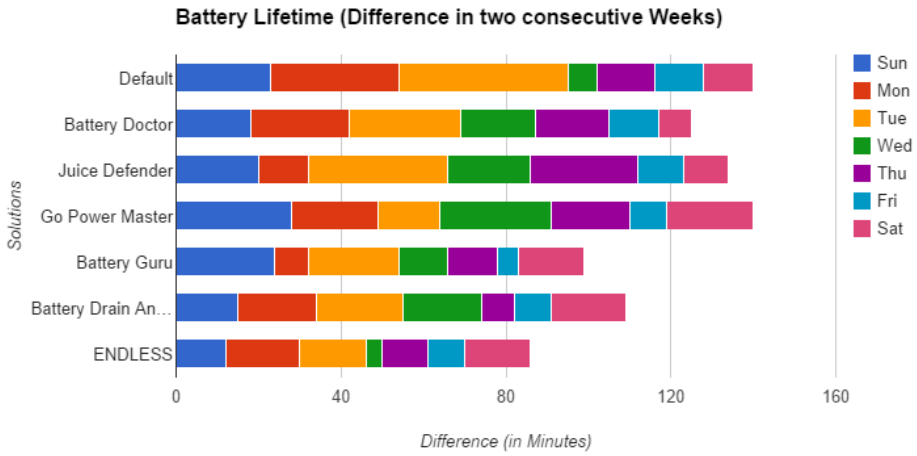


Figure 5.5: Battery Lifetime (Difference within two consecutive Weeks)

day classification at a minimal cost on processing and energy consumption.

5.2 Feature Comparison

Smartphones offer a wide range of features from sending/receiving text messages to automatically synchronizing cloud data that may contain contacts or files of any type. Here, the term "Feature" represents the functionalities offered by the platform that is visible to the users. For example, Bluetooth is a feature used by the users to transfer the files. Features may be a background service started by the operating system or an application like Google Drive, Dropbox (Cloud Sync Feature). The alterations in features may directly impact the user experience. As a battery saving mechanism, the solutions or applications alter the features to minimize processing. The alteration may involve delaying the synchrinoziation interval or completely shutting down a service. In this section, we present the comparison of feature alteration by the applications we discussed earlier.

The following is a list of abbreviations for the terms used in Table 5.4.

1. UT - Untouched. Solution has no influence on the feature.

2. OFF - Solution turns OFF the feature.

3. OFF(L) - Solution turns OFF the feature on LOW Battery.

Feature	Default	Battery Doctor	Juice Defender	Go Power Master	Battery Guru	Battery Drain Analyzer	ENDLESS
Accelero-meter	UT	UT	UT	UT	UT	UT	UT
Blue-tooth	UT	OFF	OFF	OFF(L)	OFF	OFF	UT
WiFi	UT	OFF(L)	OFF(L)	OFF(L)	UT	UT	UT
Mobile Data	UT	OFF(L)	OFF(L)	OFF(W)	OFF(L)	OFF(L)	UT
Email Sync	UT	OFF	OFF	OFF	DOAU	OFF(L)	DOAU
Cloud Sync	UT	OFF	OFF	OFF	DOAU	OFF(L)	DOAU
Bright-ness	LOW(10)	LOW(L)	LOW(L)	LOW(L)	UT	UT	UT

Table 5.4: Feature Comparison

4. OFF(W) - Solution turns OFF the feature when Smartphone is in WiFi.

5. DOAU - Depends on Application Usage.

Table 5.4 clearly shows that the default Android system does not turn on or off any feature except "Brightness". As we already discussed in Chapter 2, Android operating system reduces the brightness to 10 when the battery reaches 10% battery level. We can also infer from the table that the static solutions turn off almost all the features as battery saving mechanism irrespective of user needs. So, they certainly affect user experience. On the other hand, the dynamic solutions especially Battery Guru and our proposed ENDLESS, delay Email and Cloud sync based on user's app usage statistics. Thus, dynamic solutions extend battery lifetime without compromising user experience.

5.3 Energy Consumption by Itself

The energy consumption by the application or solution is a critical concern in designing an energy saving mechanism. When we propose a new methodology in extending battery lifetime, consuming energy significantly more than any other application or service will compromise the primary goal. So, the designers constantly deploy their

solution and test for its battery consumption throughout the implementation phase. We tested our proposed ENDLESS solution and the other applications under our consideration for their energy consumption. We performed the test for a two week period in the same way as we did in Battery Lifetime comparison. When one application is installed on our device, all other battery saving applications are uninstalled for accuracy. We used the same device Google Nexus 4 for this testing. At the end of our testing, our result shows that ENDLESS and all the other battery saving applications under our comparison - Battery Doctor, Juice Defender, Go Power Master, Snapdragon BatteryGuru and Battery Drain Analyzer consume less than 1% battery per discharge cycle. That is, in a single discharge cycle or one complete battery lifetime, the battery saving applications consumed less than 1% of the total energy available. This proves that our ENDLESS is inline with other battery saving mechanisms in self energy consumption. Table 5.5 shows the complete data of battery consumption by each battery saving application. This table clearly conveys the fact ENDLESS is inline with other battery saving applications and infact it consumes lesser energy than few of them. Figures 5.6 and 5.7 show the pictorial representation of Table 5.5.

	Battery Doctor	Juice Defender	Go Power Master	Battery Guru	Battery Drain Analyzer	ENDLESS
Sun	0.9	0.8	0.9	0.7	0.9	0.8
Mon	0.8	0.7	0.8	0.7	0.8	0.8
Tue	0.9	0.9	0.7	0.8	0.8	0.8
Wed	0.7	0.8	0.8	0.7	0.8	0.7
Thu	0.9	0.7	0.9	0.7	0.9	0.7
Fri	0.7	0.8	0.8	0.9	0.9	0.9
Sat	0.9	0.8	0.9	0.8	0.9	0.9
Sun	0.8	0.8	0.9	0.9	0.9	0.9
Mon	0.8	0.9	0.9	0.7	0.7	0.8
Tue	0.8	0.9	0.7	0.8	0.7	0.7
Wed	0.9	0.7	0.8	0.8	0.8	0.7
Thu	0.7	0.8	0.7	0.8	0.9	0.8
Fri	0.8	0.7	0.9	0.9	0.9	0.9
Sat	0.9	0.8	0.9	0.9	0.9	0.9
Avg	0.82	0.79	0.83	0.79	0.84	0.81

Table 5.5: Battery Consumption by Itself

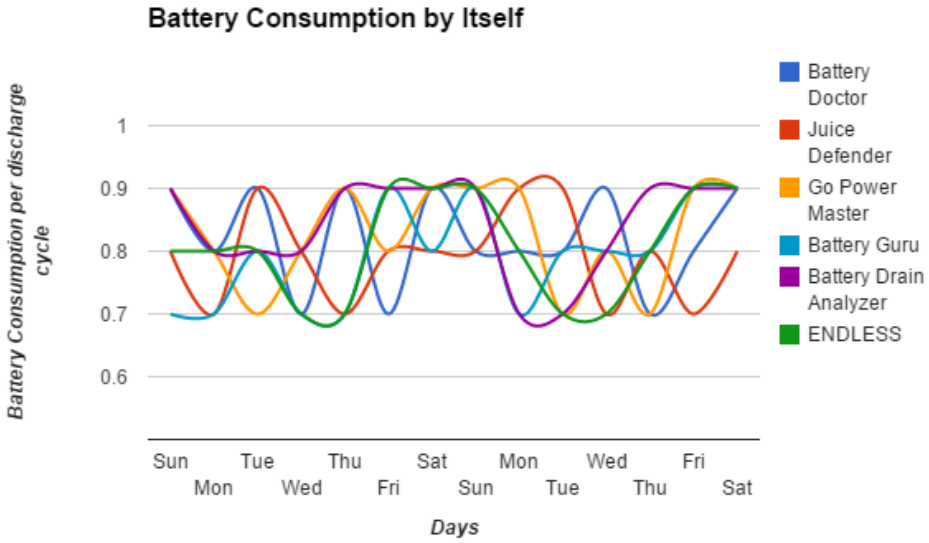


Figure 5.6: Battery Consumption by Itself

5.4 Next Recharge Estimate Accuracy

ENDLESS manages and distributes the energy based on logged data, calculations and estimations. The most important estimation is the battery life time in the current discharge cycle. NRE (Next Recharge Estimate) plays a vital role in determining the approximate battery life time. As we discussed in Chapter 3, NRE is calculated based on past Recharge Start Times (RSTs) and Time After Previous Recharges (TAPRs). The performance of our ENDLESS solution highly depends on the accuracy of our estimated recharge start time (NRE). The purity of NRE clusters can be justified by evaluating the accuracy of individual records. If the accuracy of each record is higher, then the cluster is highly pure. To evaluate the accuracy, we cleared all the past data and deployed ENDLESS freshly on our Nexus 4 device. After the learning phase, we started monitoring estimated NRE and actual RST for a period of one month. Figure 5.8 shows a pictorial representation of the difference between estimated and actual recharge start time over 1 month period. During the first week after the learning phase, the estimations were deviated highly from the actual RST and stumbled in consistency. As it gained more data, during the second and third week, the difference has been reduced. As shown in Figure 5.8, the line is steeply declining but consistent. During the fourth week, we noticed the difference is consistent around 10 minutes.

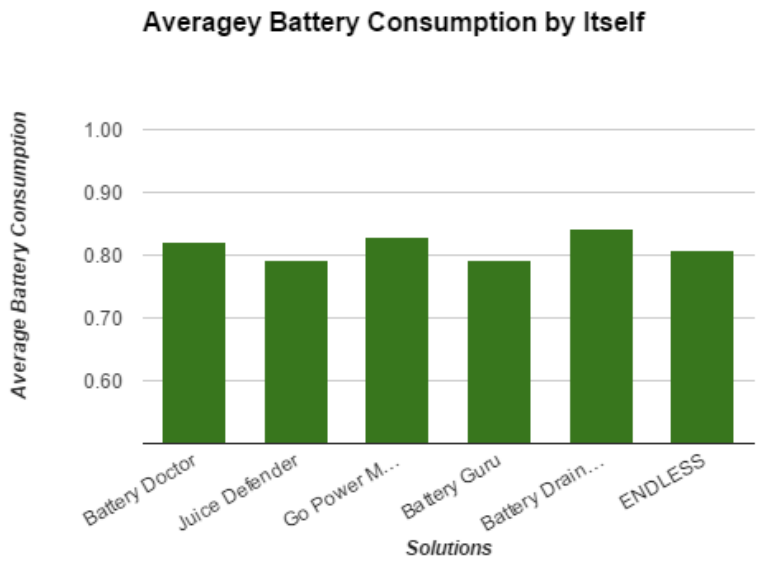


Figure 5.7: Average Battery Consumption by Itself

This might be higher when the user's recharge behaviour changes drastically and this might be lower when the user's recharge behaviour is consistent. But in any case, as the sample data increases the difference decreases and results in the increase of NRE accuracy.

5.5 Against Llama

Llama is user driven energy management algorithm designed and published by Banerjee et al. in [28]. The findings from Llama is the primary motivation for our proposal in the user centric direction. So, it becomes neccessary to evaluate our ENDLESS performance against our base research paper [28]. The authors deployed their algorithm on Windows Mobile phones and evaluated the number of recharges per week. As we implemented our ENDLESS for Android platform, we cannot compare the absolute result against Llama. Therefore, we evaluated the difference in number of recharges between the before solution deployment and after solution deployment and finally compared the differences. Difference is the actual parameter that conveys how better the solution is on its own platform. We calculated the number of recharges that happened in a 30 day period before and after ENDLESS deployment and found them to be 43 and 38 respectively. As shown in Table 5.6, in Windows Mobile,

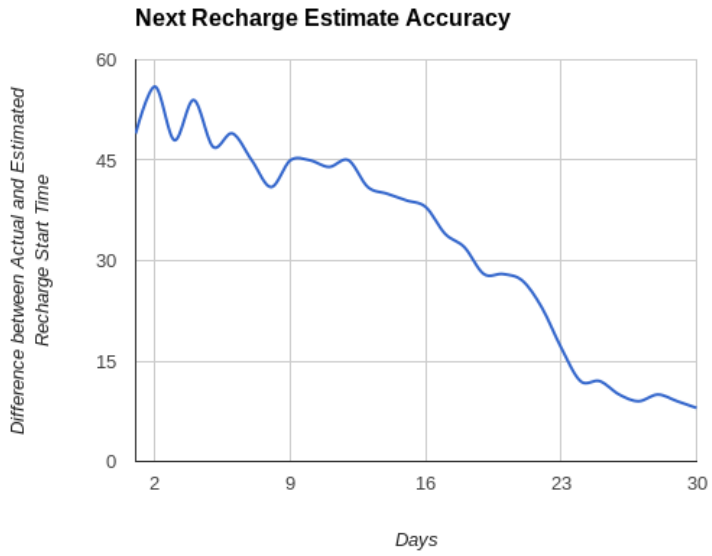


Figure 5.8: Next Recharge Estimate Accuracy)

the number of recharges per week were 10.1 before Llama deployment and 8.9 after Llama. On the other hand, in Android, the number of recharges were 10.03 before ENDLESS deployment and 8.87 after ENDLESS. In other words, Llama reduced the number of recharges by 11.88% Windows Mobile and ENDLESS reduced the number of recharges by 11.57% in an Android smartphone. Thus the performance of ENDLESS is almost equal to Llama and varies only by 0.31%. From this evaluation, we are able to prove that it is possible to extend the battery lifetime without compromising much feature alteration. Llama concentrates only on user's recharge behaviour to extend the battery lifetime whereas ENDLESS concentrates on both recharge behaviour and the app and service usage statistics to extend the battery lifetime as well as to improve the user experience by saving or releasing energy for necessary applications.

	Before	After
Llama	10.1	8.9
ENDLESS	10.03	8.87

Table 5.6: ENDLESS Vs. Llama (Recharge attempts per week)

Chapter 6

Conclusion

Smartphones have become and integral part of our life and help us get connected with the world all the time. Although smartphones possess high end hardware and offer amazing functionalities, the limited battery energy limits their capability. Battery saving and extension has always been challenging for researchers and smartphone manufacturers. This thesis discussed the evolution of smartphones, various smartphone operating systems and various types of batteries that are used in smartphones. We explained the best existing solutions and how they differ from each other. There is always more than one solution to any problem in the world. A solution is considered best when it achieves maximum performance with minimal cost. The battery saving mechanisms try to extend the battery lifetime without much compromise in the user experience. We categorised the solutions into static and dynamic depending on their consideration of users' usage patterns. In our evaluation, it is clear that the solutions that consider user needs and behaviour yield better results than the static solutions. Among the dynamic solutions we discussed, ENDLESS out performed in battery lifetime with negligible prediction errors. By rigorously testing ENDLESS for a significant period, we proved the performance of ENDLESS is better and consistent. The following summarizes our findings through this thesis.

1. User centric approach in extending the battery yields significantly better results compared to the static solutions.

2. Smartphone users use their devices different based on daily routine.

3. Tracking users' usage patterns by weekend/weekday basis yields better results than the other dynamic solutions.

4. As the log data increases over time, accuracy of the estimation increases.

6.1 Future Work

The good solutions to a problem is always good at present and have ways to improve for future. In the real world, all problems change over a period and so does the solution. In section 4.3, we have listed the limitations in our ENDLESS solution and the interested researchers can extend this thesis by addressing them. In this thesis, we assumed that the users may not recharge their devices more than twice a day and so we fixed K = 2 in K-Means estimation of NRE (Next Recharge Estimate). In the future, we shall make the K dynamic so that the solution will adapt to the user's recharge behaviour if it happens more than two times in a day. In our solution's design and implementation, we categorized the data into weekday/weekend and the evaluation showed the difference in smartphone usage pattern which in turn helped distribute the available energy efficiently. In future, if it is categorized further into each day or in a multi dimensional attribute, for example in addition to every day categorization, if we add holiday/workday categorization or something similar to that, it may have potential to find a pattern and distribute the energy more efficiantly. Also, as we implemented and deployed our ENDLESS only on Android platform, there is an opportunity to extend the solution to other platforms. We believe that after optimizing the implementation, deploying the solution on other platforms and testing it on other devices it will provide interesting facts and results.

Bibliography

[1] Android on 80 http://www.canalys.com/newsroom/android-80-smart-phones-shipped-2013. Accessed: 2014-07-14.

[2] Battery doctor (battery saver). https://play.google.com/store/apps/details?id=com.ijinshan.kbatterydoctor_en. Accessed: 2014-07-14.

[3] Battery saver with widget. https://play.google.com/store/apps/details?id=com.msd.battery.indicator. Accessed: 2014-07-14.

[4] Broadcastreceiver — android developers. http://developer.android.com/reference/android/content/BroadcastReceiver.html. Accessed: 2014-07-14.

[5] Class that provides access to some of the power management functions. https://android.googlesource.com/platform/frameworks/base/+/40eec4c0f1392665dbfcd9ca9ea4a9519a71c34a/core/java/android/os/Power.java. Accessed: 2014-07-14.

[6] Father of the cell phone. http://www.economist.com/node/13725793?story_id=13725793. Accessed: 2014-07-14.

[7] Getting started — android developers. http://developer.android.com/training/index.html. Accessed: 2014-07-14.

[8] Go battery saver power widget. https://play.google.com/store/apps/details?id=com.gau.go.launcherex.gowidget.gopowermaster. Accessed: 2014-07-14.

[9] Googles android becomes the worlds leading smart phone platform. http://www.canalys.com/newsroom/google%E2%80%99s-android-becomes-world%E2%80%99s-leading-smart-phone-platform. Accessed: 2014-07-14.

[10] How much time do we really spend on our smartphones every day? http://www.businessinsider.com.au/how-much-time-do-we-spend-on-smartphones-2013-6. Accessed: 2014-07-14.

[11] iphone 5s, 5c battery life doesnt stack up to android counterparts. http://bgr.com/2013/11/05/iphone-5s-android-battery-life-comparison/. Accessed: 2014-11-05.

[12] Juicedefender - battery saver. https://play.google.com/store/apps/details?id=com.latedroid.juicedefender. Accessed: 2014-07-14.

[13] logcat — android developers. http://developer.android.com/tools/help/logcat.html. Accessed: 2014-07-14.

[14] Notifications — android developers. http://developer.android.com/guide/topics/ui/notifiers/notifications.html. Accessed: 2014-07-14.

[15] Power profiles for android — android developers. https://source.android.com/devices/tech/power.html. Accessed: 2014-07-14.

[16] Sharedpreferences — android developers. http://developer.android.com/reference/android/content/SharedPreferences.html. Accessed: 2014-07-14.

[17] Sharedpreferences.editor — android developers. http://developer.android.com/reference/android/content/SharedPreferences.Editor.html. Accessed: 2014-07-14.

[18] Snapdragon batteryguru. https://play.google.com/store/apps/details?id=com.xiam.snapdragon.app. Accessed: 2014-07-14.

[19] Sqlitedatabase — android developers. http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html. Accessed: 2014-07-14.

[20] Sqliteopenhelper — android developers. http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html. Accessed: 2014-07-14.

[21] Types of cell phone batteries. http://www.puremobile.com/cell-phone-batteries. Accessed: 2014-07-14.

[22] N. Song B. Kwak and L. Miller. Performance analysis of exponential backoff. In *Networking, IEEE/ACM Transactions*, volume 13, pages 343 – 355. IEEE, 2005.

[23] Alex Cocotas. Smartphone sales are on the verge of overtaking feature phone sales. http://www.businessinsider.com/chart-of-the-day-smartphones-to-beat-feature-phone-sales-2013-6. Accessed: 2014-07-14.

[24] R. Gafni and N. Geri. Generation y versus generation x: Differences in smartphone adaptation. In *Learning in the technological era: Proceedings of the Chais conference on instructional technologies research*, pages 18 – 23. Raanana: The Open University of Israel, 2013.

[25] F.H.P. Fitzek G.P. Perrucci and J. Widmer. Survey on energy consumption entities on the smartphone platform. In *IEEE Vehicular Technology Conference (VTC Spring)*. IEEE, 2011.

[26] J. Lee M. Kim, D. Yun and S. Choi. Battery life time extension method using selective data reception on smartphone. In *IEEE International Conference On Information Networking (ICOIN)*. IEEE, 2012.

[27] Prince McLean. Canalys: iphone outsold all windows mobile phones in q2 2009. `http://appleinsider.com/articles/09/08/21/canalys_iphone_outsold_all_windows_mobile_phones_in_q2_2009.html`. Accessed: 2014-07-14.

[28] M. Corner S. Rollins N. Banerjee, A. Rahmati and L. Zhong. Users and batteries: Interactions and adaptive energy management in mobile systems. In *Ubiquitous Computing*. UbiComp Proceedings of 9th International Conference, 2007.

[29] L. Han N. Ravi, J. Scott and L. Iftode. Context-aware battery management for mobile phones. In *IEEE International Conference On Pervasive Computing and Communications (PerCom)*. IEEE, 2008.

[30] J. Lee Y. Hyeon, M. Kim and S. Choi. Battery life time extension method by using signalling interval control. In *IEEE International Conference On Advanced Communication Technology (ICACT)*. IEEE, 2012.