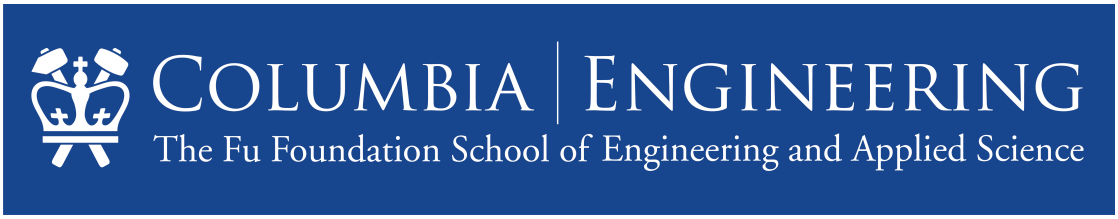

ELEN E4901 Project Two

Study of relation between energy consumption and processes

Chenxi Rao (ID cr2832)

Yihan Dai (ID yd2349)

Zikai Lin (ID zl2442)

Supervised by Prof. Qiu

December 8, 2015

Declaration

I confirm that I have read and understood the University's definitions of plagiarism, collusion and the fabrication of data from the Code of Practice on Assessment. I confirm that I have not committed plagiarism or fabrication of data when completing the attached piece of work, nor have I colluded with any other student in the preparation and production of this work.

Signed Chenxi Rao Date Dec 10,2015

Signed Yihan Dai Date Dec 10,2015

Signed Ziaki Lin Date Dec 10,2015

Abstract

This project claims the methodology about developing one kind of app used to monitor the system information including the memory size, process name or id, battery information and CPU usage. Then the relationship between the energy consumption of processes and the system information is analyzed, and then the appropriate methods are used to reduce the energy consumption are presented in the discussion part.

Contents

Chapter 1

Introduction

1.1 Objectives

This report is to show the analytical relations between the energy consumption and tasks. By considering different scenarios where tasks simulated consisting of configuration of the emulators or real devices, the detailed tasks' information, the information about the operating system and the statistical and algorithm methods used, the relations can be figured out.

1.2 Requirements

This project can be divided into three main phases. The first phase is to require the students to develop an app which can monitor the CPU usage, energy consumption which can also refer to the battery information of a specialised task and the memory size of this task.
The second part needs the analysis of the relations between energy expenditure and processes in the app, through implemented different internal functions of the app like the operating system where the app established.
The third part needs the achievement of the configuration of energy usage of the tasks. Due to lecture notes, it gives some kind of methods including algorithms methods (First Job First Serve or Shortest Job First Serve), dynamic programming algorithm to allocate the tasks and some function to stop some useless services. Here are some hints from the lecture notes:

• **A**

There are a number of tasks running on a mobile device, and it is unnecessary for some of them to keep running, such as data synchronization. Students can design some algorithm to reduce the times of data transmission to reduce the communication cost. Furthermore, some useless services or previous apps can be terminated to reduce the usage of CPU.

• **B**

From the analysis in phase 2, students can obtain the detailed information about energy consumption of running tasks. The energy consumption can be considered as the **cost**, while the performance constraint can be used to quantify the **time constraint**. With these two constraints, the goal is to find the solution with lowest cost while satisfying the time constraint requirement.

The project satisfies most of the requirements:

1. Develop an Android app to monitor resource usage for running process.
2. Analyze the relations between energy consumption and tasks.
3. Design methods to reduce the energy consumption.

1.3 Theory

1.3.1 Basic Knowledge

As shown below, the way of getting energy consumption through hidden hidden API and Internal is claimed by the blogger [1].

a) Major Problem

In order to fetch the information of the ranking of energy consumption of each app in a phone, a very class PowerFile should be imported into the project. The problem comes out that this class is not available to the developer and unfortunately, it can not be used in a normal way.

b) Knowledge Of The Class

To solve the problem, we should first focus on some principles of this class. As it is in the com.android.internal.os.PowerProfile. This class is responsible for parsing log files, we can create one of its objects , to get specific information

through the object. Specific consumption calculation is more complicated. The principle of getting the ranking of mobile power consumption is by reading the log files to achieve battery through the hidden class android.os.BatteryStats and com.android.internal, the special package. Internal package can not be found in android.jar. Even if the package can be found from somewhere else and put into android.jar, it can not be used since it is restricted by ADT.

c) Two Probable Solutions
As impossible as it seems to be, it could be solved through two solutions, through what we found out on the internet. Following two solutions are as listed below:

1. modify ADT to remove restriction (but it is hard to achieve)
2. not put it into internal android.jar. Instead, put it alone as a package into the project , so it will not be limited.

In second way, we can use hidden API and Internal package. We should be careful put it into every table of content except libs because in this case it would be detected by ADT. We will focus on the second solution in our following discussion.

d) Reasons why hidden API and Internal package can not be used
When we use the android SDK for development, a very important jar filesandroid.jar. This package removes all marked as the end of hide classes, methods , enumerations, fields, and Internal package . When our program running on the device, a framework.jar file will be loaded on the device, which contains the removal part. So our solution is to find ways to get those missing API framework.jar and put them into android.jar. Of course these can be solved by compiling the source code , but that too much trouble. Here I introduce a simple way to get the content.

1.3.2 Procedure of getting missing API

a) Obtain framework.jar
By adb pull command (you can also use DDMS): adb pull /system/framework/frame-

work.jar We need classes.dex file in this package. It can only be fetch from simulator whose version is under 2.3.3 since .odex file on a real machine generally is been optimized , and simulator with version above 2.3.3 does not have classes.dex file . Note : we can not get classes.dex file in a real machine,so to get it in the simulator.

b) Change framework.jar into framework.zip, unpack, get classes.dex file.

c) Convert .dex files into .jar. Here we can use tool dex2jar to decompile. You will get a classes.dex.dex2jar.jar file after the conversion.

d) Rename classes.dex.dex2jar.jar to classes.dex.dex2jar.zip, then unzip it. Copy android.jar from the Android SDK directory / platforms / platform-X /, then use WinRAR software to open it. Afterwards, we add classes-dex2.jar files on the base of android.jar. Finally, we can safely get android.jar we need.

Chapter 2

Methodology

As shown in Figure 2.1, to get the system process information, we can build an activity manager object to implement it.
Using method **RunningAppProcessInfo()**, we can get process information as following:

1. .pid > process id
2. .uid > user id where the process is
3. .processName > process name
4. .pkgList > app package name that running with the process

The most useful information is process id, it can helps us to get battery, memory and CPU information. By using **getProcessMemoryInfo(pid)**, we can get the process memory size.

```
double totalTime = 0.0;
for (ActivityManager.RunningAppProcessInfo appProcessInfo : appProcessList) {
    int pid = appProcessInfo.pid;
    double time = getAppProcessTime(pid);
    totalTime = totalTime + time;
}
DecimalFormat decimalFormat = new DecimalFormat("0.000");
DecimalFormat decimalFormat1 = new DecimalFormat("0.0");

for (ActivityManager.RunningAppProcessInfo appProcessInfo : appProcessList) {
    // process id
    int pid = appProcessInfo.pid;
    // user id
    int uid = appProcessInfo.uid;
    // process battery usage
    double ratio = 100 * getAppProcessTime(pid) / totalTime;
    String num = decimalFormat.format(ratio);
    powerUsage = num;
    //process CPU Usage
    String num1 = decimalFormat1.format(ratio);
    cpuUsage = num1;
    // process name
    String processName = appProcessInfo.processName;
    // process memory size
    int[] myMempid = new int[] { pid };

    Debug.MemoryInfo[] memoryInfo = mActivityManager
            .getProcessMemoryInfo(myMempid);
    int memSize = memoryInfo[0].dalvikPrivateDirty;
```

Figure 2.1: The code of getting Process id, Memory size and Process name

Then, the next step is to gain Power Usage and CPU utilization. Using power profile, we can calculate the power usage in this way. Consider formula $W = U \times I \times t$, voltage U in the mobile phone is always the same. So we can use $Q = I \times t$ to represent the power usage percent. In the Android system, class **BatteryStatsImpl** provides the app process running time and class **PowerProfile** provides the current value.
First, the following steps have shown how to use the **BatteryStatsImpl**:

1. Build **ActivityManagerService** and initialize the **BatteryStatsService** and write the data into **batterystats.bin**;
2. Build **BatteryStatsImql** in **BatteryStatsService** and write data into **batterystats.bin**;
3. Execute **mBatteryStatsService.getActiveStatistics().readLocked()**, let the **BatteryStatsImpl** to read the data in **batterystats.bin**;
4. During calculating the app power usage, **BatteryStatsImpl** methods can be called and then information from it can be output.

Secondly, to use the **PowerProfile**, we tried use public double **getAveragePower(String type)** to read the current value from the **powerprofile.xml**.
Finally, we can get the following power consumptions: process consumption, wake lock consumption, data traffic consumption, WIFI consumption, and others sen-

sors consumption, then sum them up and we will get the total consumption of an app process.

Since we can not use power profile for Android 4.0 or above directly, we use another method to roughly calculate the battery usage. We assume that the power consumption is in proportion to the CPU process time. We firstly read the process time by its **pid** and sum them up as whole. The percentage of process's running time from the whole can roughly represent the battery usage of the process. Also, the CPU usage can be calculated in the same way so these two values should be the same inside the app.

```
// process running time for power usage and cpu usage
private long getAppProcessTime(int pid) {
    FileInputStream in = null;
    String ret = null;
    try {
        in = new FileInputStream("/proc/" + pid + "/stat");
        byte[] buffer = new byte[1024];
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        int len = 0;
        while ((len = in.read(buffer)) != -1) {
            os.write(buffer, 0, len);
        }
        ret = os.toString();
        os.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (in != null) {
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    if (ret == null) { return 0; }
    String[] s = ret.split(" ");
    if (s == null || s.length < 17) { return 0; }

    long utime = string2Long(s[13]);
    long stime = string2Long(s[14]);
    long cutime = string2Long(s[15]);
    long cstime = string2Long(s[16]);

    return utime + stime + cutime + cstime;
}
```

Figure 2.2: The code of getting Battery Usage and CPU Utilization

Chapter 3

Results & Analysis

Here are the outputs of the app. It consists of the CPU utilisation, battery information and memory size of all the running tasks. Besides, the output contains the Process ID and Process name.

3.1 Results

Figure 3.1 shows the initial picture of this app. One button called "System Process Info" is used to display the main useful information including Memory size, Battery usage, CPU utilization in Figure 3.3. Another button named "Running App Info" is to show the processes information as the processes ID and Processes names in Figure 3.2. It also shows the whole available memory size is 251MB.

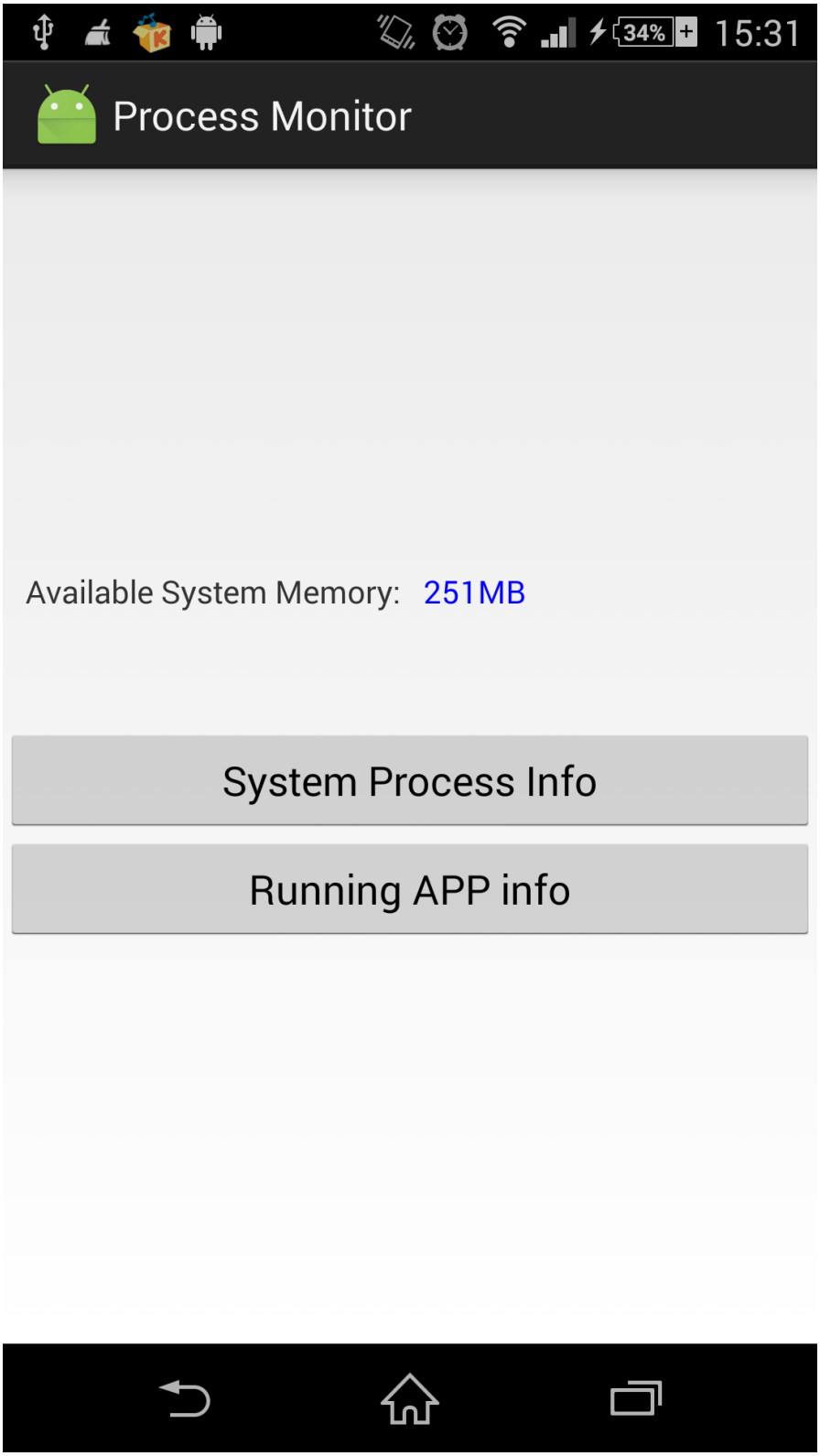

Figure 3.1: Main page of app

As shown in Figure 3.2, there are totally four characters respectively named

Applabel, Packet name, Process id and Process name. For instance, the Applabel of apk named QQ is **QQ** itself. The packet name is **com.tencet.mobileqq**. The running process is called **com.tencet.mobileqq:web** and the **id** is 23354.

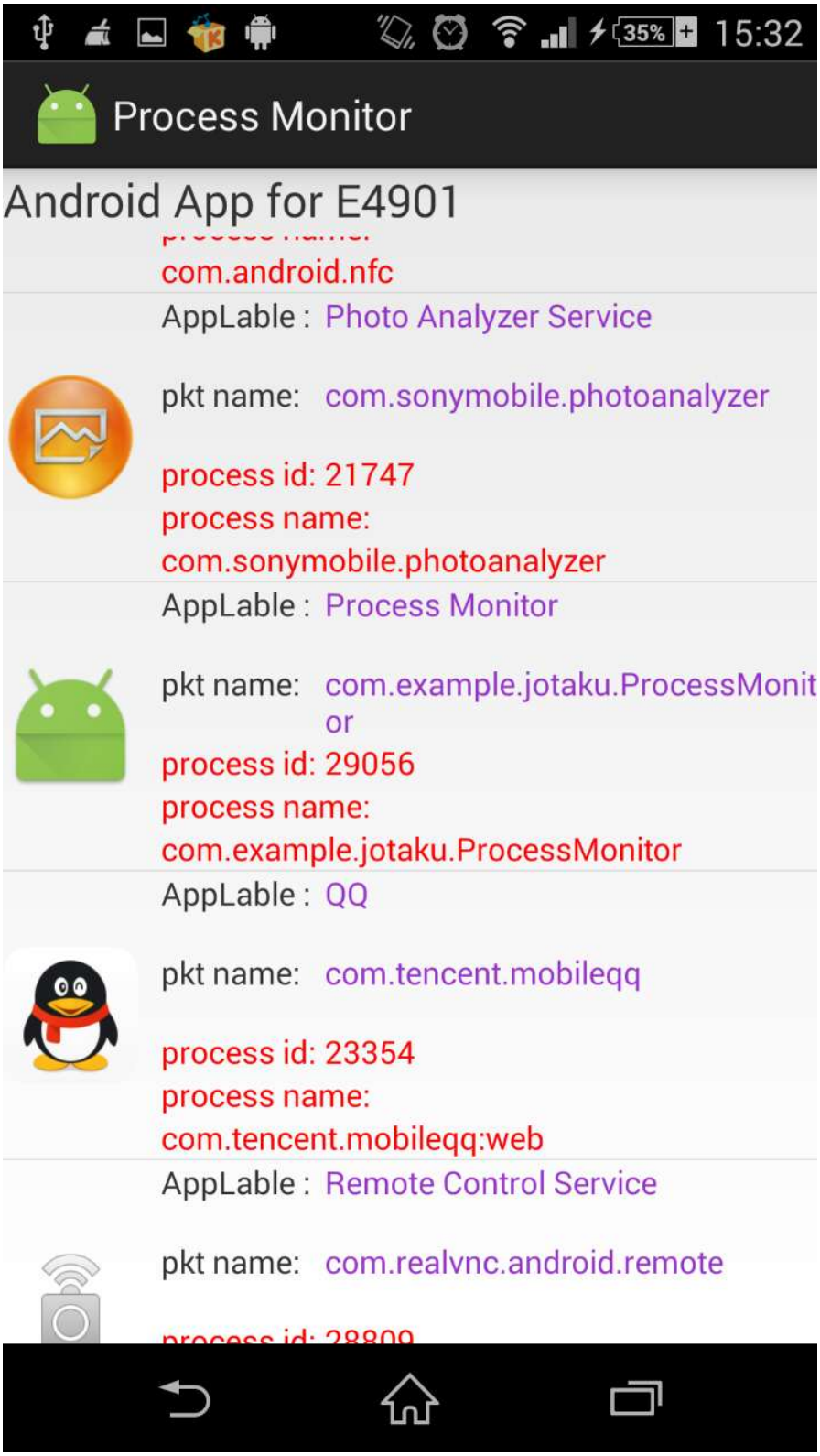

Figure 3.2: Process information of app

According to Figure 3.3, there are 49 processes that are running in the apk

totally. If the process id named 18731 is chosen as an instance, meantime the other characters can be displayed to the users. The process is called **com.sonyericesson.album**, and its CPU usage is about 0.3%. The memory size of this process occupies with is about 36812KB and its battery usage is about 0.321%. It can be indicated that the battery usage is close to the CPU usage.

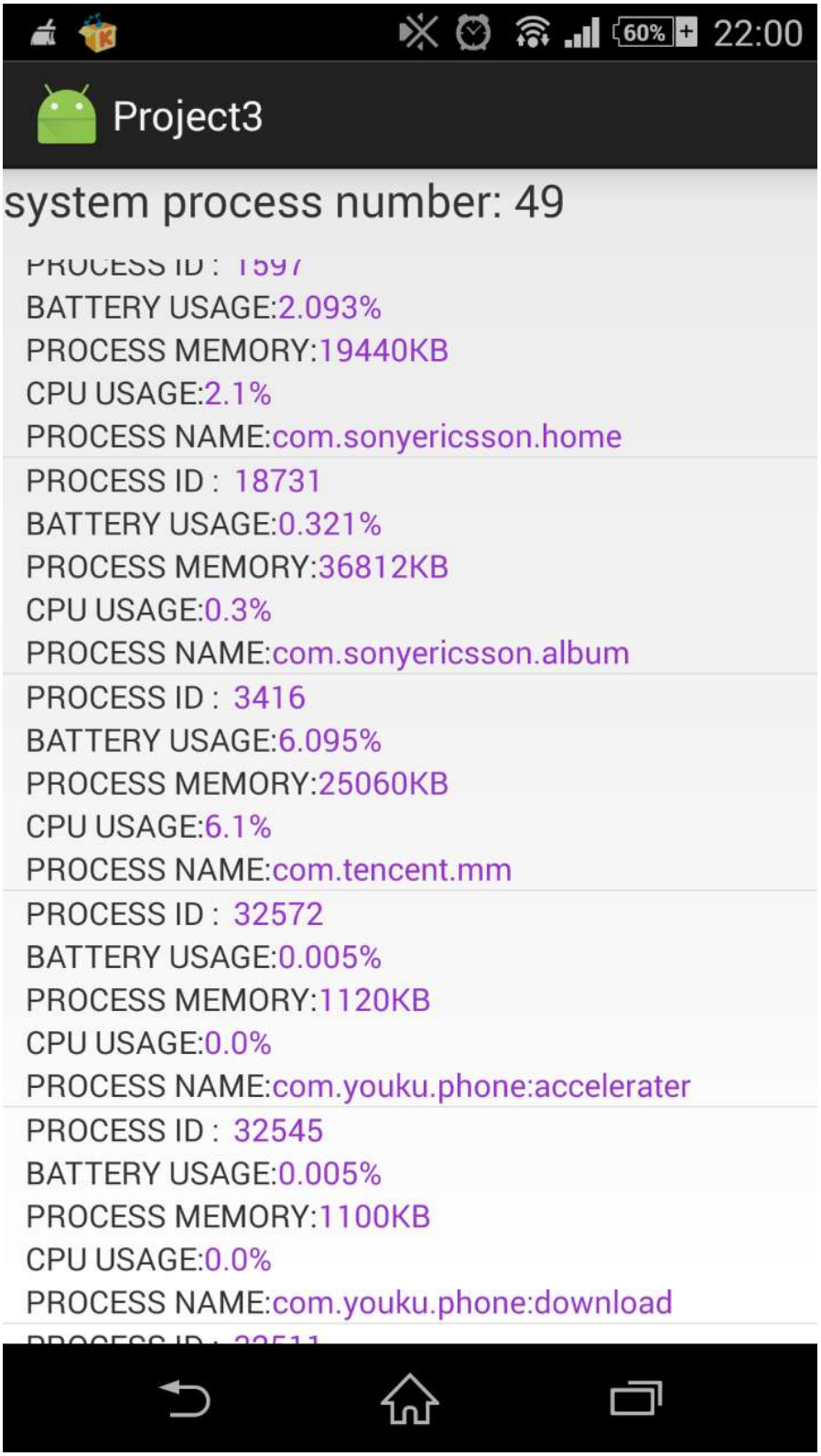

Figure 3.3: System Information of app

3.2 Analysis

3.2.1 The relations between energy consumption and the system information

From the process info part, we can see that the power usage has no relation to the memory usage. Though some processes have larger memories, it has less power usage because it may in the sleep mode. Under different modes, the process power usage is different. For example, process of Twitter will has less power consumption if there is no data transmission and network connection while in its sleep mode. And once it is awake, the process will begin sending and receiving data. In that case, the power consumption will increase. However, under this two conditions, the memory size remains the same.

The CPU usage is the same with the power usage. In the app, we develop the power usage from the CPU usage. We assume that the power usage is in proportion to the CPU processing time. Also, the CPU usage is in proportion to the CPU processing time, too. We record all the process running time in a period and sum it up as a whole. The percentage of process running time from the whole can roughly represent the battery usage of the process and also the CPU usage.

Chapter 4

Discussion

4.1 How to access the battery information

From the above background theory, there's an alternative method to achieve the ranking of the applications' battery information since the **PowerProfile** class cannot be accessed by the developers.
According to the notes, the internal class can be used by the class reflection. The developer can decode the internal class and make it as the module for their separate work. However, it's difficult to decode the internal class and it's no illegal to use such kind of method.

4.2 Advanced Phase Achievement

4.2.1 Basic Knowledge

Before we move to improve the battery performance, there are some principles from Kai Hu [2] we need to know.
a) Understanding Battery Drain
Mobile power consumption in different hardware module are not the same , among which some of them are very energy consuming , while power consumption in other modules are relatively small.
Calculating and recording power consumption are troublesome and contradictory. Moreover recording power consumption itself is a waste of energy. The only viable

solution is to use a third-party equipment to monitor power so as to get to the real power consumption.

Power consumption is small when the device is in standby state. Set N5 as an example, in flight mode it can stand closer to a month. But when the screen lights up, the hardware modules need to start to work. It would need to consume a lot of power.

After wake up timing task in the device processing by using WakeLock or JobScheduler, the device should be back to its initial state timely. Every time when you wake cellular signals to transfer data consumes a lot of power. It is even more power consuming than WiFi.

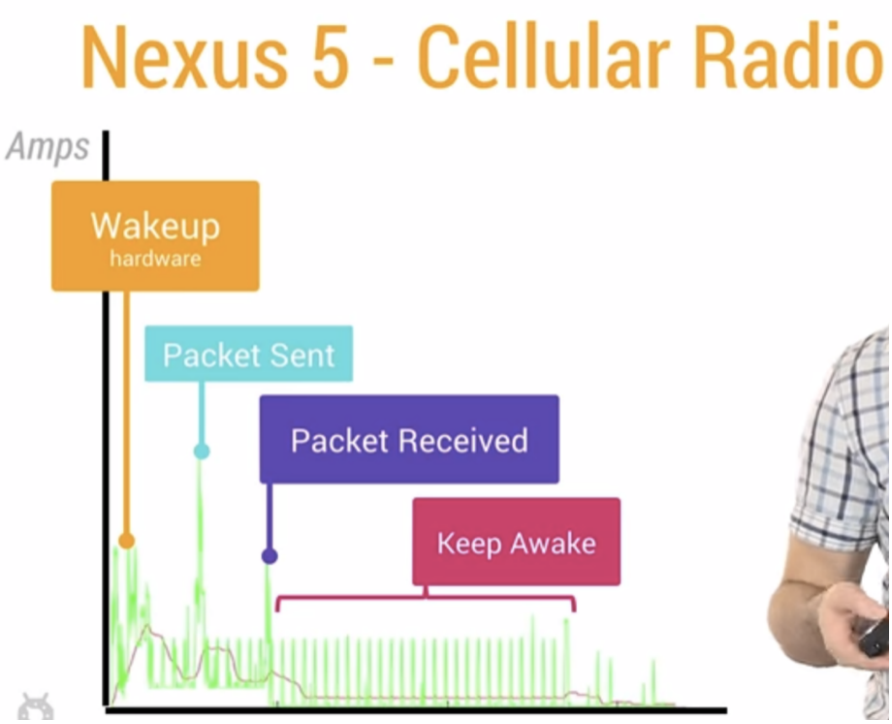

Figure 4.1: Energy consumption in different states in N5

b) Battery Historian

Battery Historian is a new API when Android 5.0 is introduced . By the below instructions , you can get information of power consumption on the device:

adb shell dumpsys batterystats > xxx.txt // battery consumption in the whole device

adb shell dumpsys batterystats > com.package.name > xxx.txt // battery consumption in a single app

After we get the original data of power consumption, we need to write a python script through Google to convert data into a more readable html file :

python historian.py xxx. txt > xxx.html

After open the html file, you can see a list of the data seemed as like generated by TraceView.

Figure 4.2: HTML file of original power consumption

c) Track Battery Status & Battery Manager

We can get the current charging state of phones by programming. After obtaining the state of charge information, we can especially focus on some part of the code to optimize it. For example, the phone will perform some very power-consumption operations only in the case when the mobile is in AC charging state.

d) Wakelock and Battery Drain

To maintain power and continue to urge the user to use your App could caught you in dilemma. Still, we can come out with some solution to address the problem. Suppose your phone has installed a lot of social networking applications. Even if the phone is in standby mode, it will often be waked up to check the new data to synchronize information on these applications. Android will continue to shut down a variety of hardwares to extend the standby time: Firstly, the screen is gradually darkened until it is closed. Then the CPU goes to sleep. But even in this sleep state, most applications still try to work. They will continue to wake up the phone. One of the easiest way to wake up the phone is to wake PowerManager.WakeLock API to keep the CPU working and to prevent the screen from darken. This makes the mobile phone can be woken up, implemented, and then it will go back to sleep. It is important to know how to get WakeLock. At the same time, it is also of importance to release WakeLock timely. This is why using parameter() method with timeout parameter is very critical.

But only set the timeout is not enough to solve the problem. We need to consider

how long time out should be, when to re-access the value and etc. Non- precision timers would be a good solution to the problem. Generally, we will set fixed time for an operation, but the dynamic modification might be better in this case. For example, if another program need to be waken up five minutes later, it is better to wait and bundle the two tasks together and operate them simultaneously, which is the core principle of non- precision timer. We can customize the scheduled task. But if it detects a better time schedule, it can delay your mission in order to save power consumption.

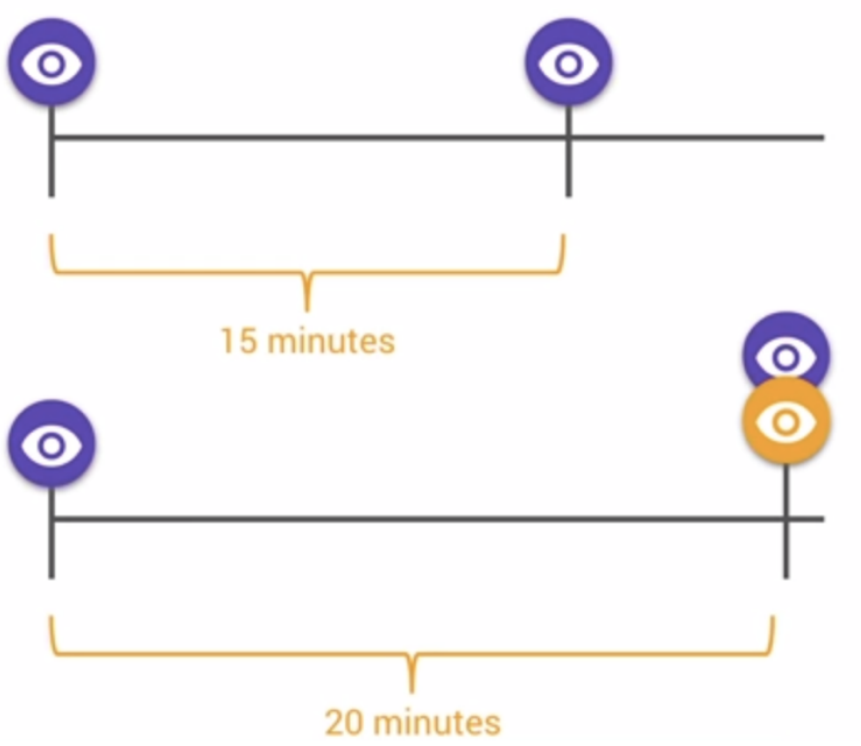

Figure 4.3: The mechanism of JobScheduler API

e) **Network and Battery Drain** Typically , transmitting data using 3G mobile networks, there are three power consumption states :

Full power: the highest-energy state. The mobile network connection is activated, allowing the device to operate the maximum transfer rate .
Low power: an intermediate state. The power consumption is almost 50% as that in full power state.
Standby: the lowest state. No data need to be transmitted, in which case cost the least power.

In short , in order to reduce power consumption in a cellular mobile network, it is best to batch execution network requests and avoid frequent intervals network requests .

Through what we learned in Battery Historian before, we can get the device power consumption data. If the data of power consumption in a mobile cellular network (Mobile Radio) is as presented below, in which the spacing is very small and frequently intermittent, suggesting that the performance of power consumption is very unsatisfying.

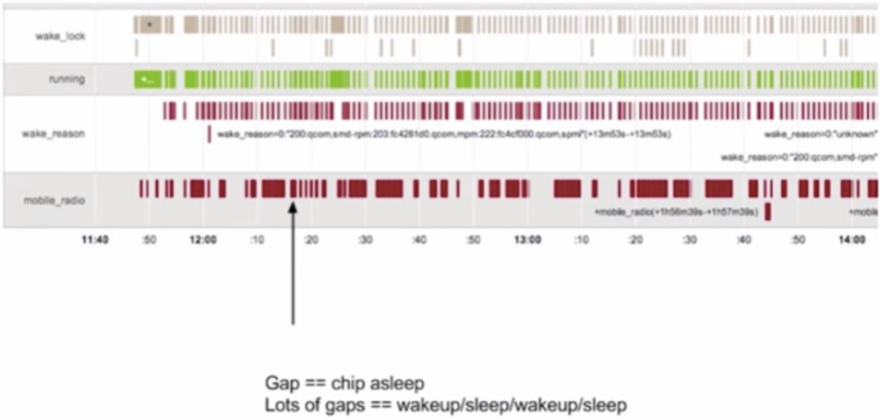

Figure 4.4: Power consumption before optimization

After optimization, the figure below shows the power consumption performance is good:

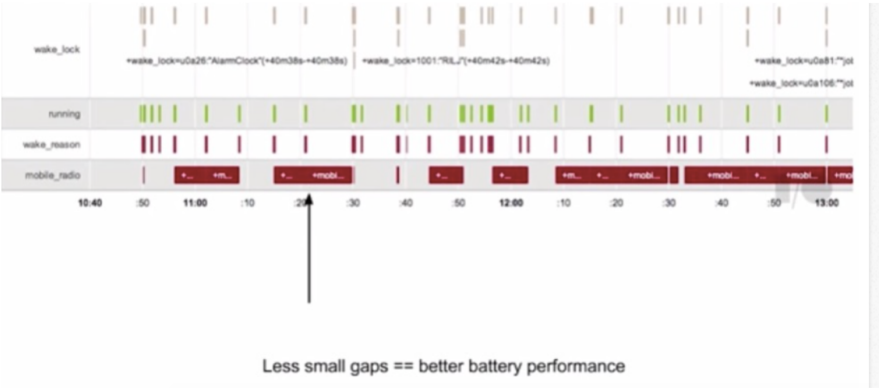

Figure 4.5: Power consumption after optimization

Energy consumption when data transmission through network in WiFi cost less than that in the mobile network indicating that we should try to avoid data transfer

under mobile network, but instead WiFi environment would be much more power-friendly.

4.2.2 What we have accomplished

In our project, we used a method to bind apps with the action in which if we click on the bar of an app which we dont want to use it for a relative long time, the screen would showcase kill the process n APP running with the process . By clicking the second bar we can go to the very apps detail information and decide if we want to remain the process. By clicking on the second bar, we can safely close it and release some CPU memory in order to speed up other activities and save energy at the same time.

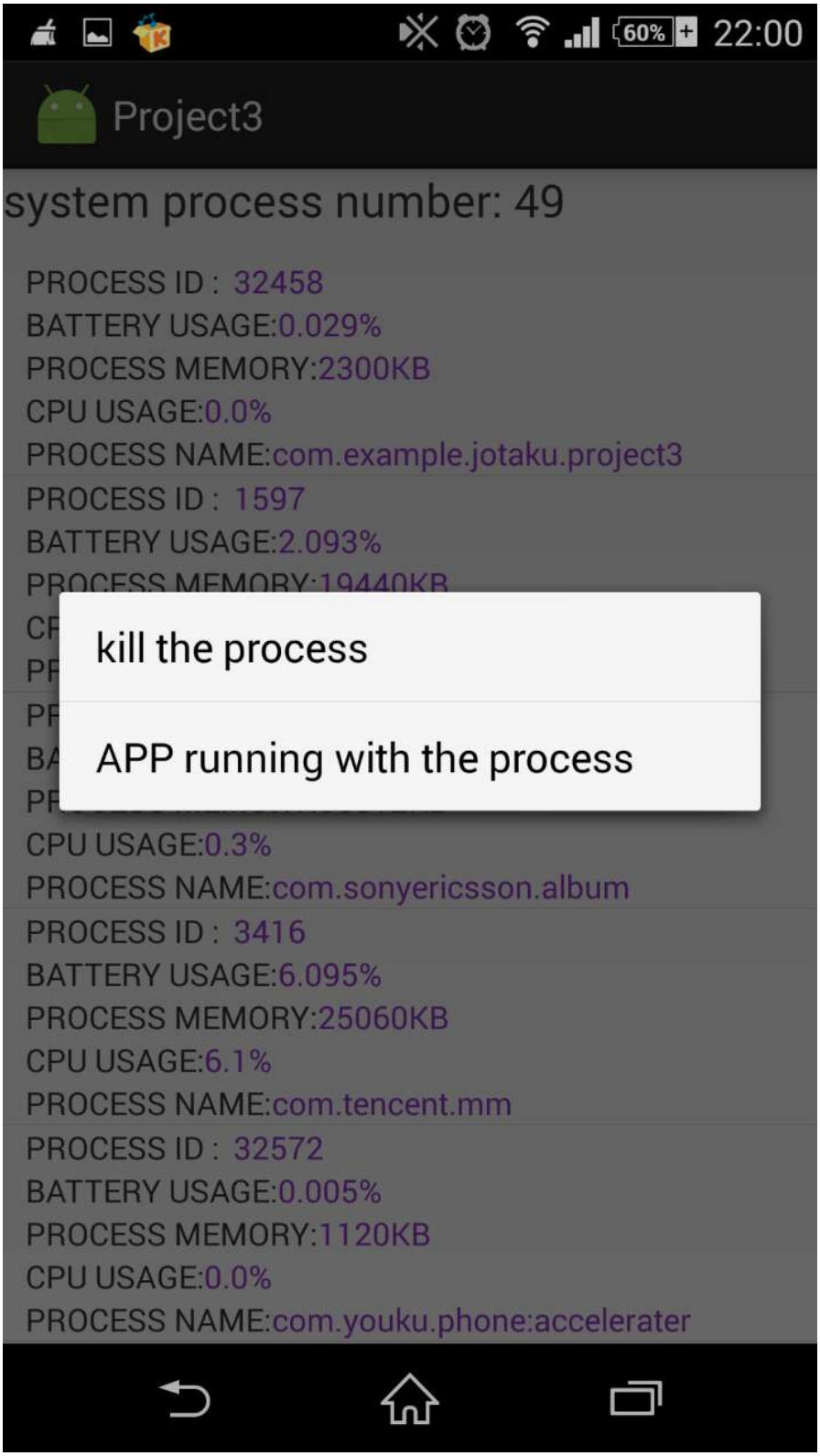

Figure 4.6: Kill the process

In the next state, we should probably use Job Scheduler. Job Scheduler central-

izes tasks that have received, selects an appropriate time, an appropriate network, implements the tasks simultaneously. The process would be like:
Firstly, initiate a JobService.
Secondly, use the monitor and click the button to trigger N tasks and then handled by JobService.

Chapter 5

Conclusions

We have developed an APP to monitor the running process about CPU, memory, and battery usage. By analyzing the relationship between the CPU, memory, and battery usage, we find that The CPU usage is the same with the power usage and the power usage has no relation to the memory usage. Finally, we develop the kill process function for the users to kill the useless process manually, and give the idea about the improvement of our APP to reduce the energy consumption by job scheduler algorithm.
Basically, this project has satisfied the three phases of requirements, but the apk itself have some drawbacks and it will be enhanced in the future.

References

[1] jiangwei0910410003, *Hidden API and Internal use in Android Gets power rankings*, 2014. [Online]. Available: http://blog.csdn.net/jiangwei0910410003/article/details/25994337

[2] K. Hu, *Android Performance Optimization of power articles*, 2015. [Online]. Available: http://hukai.me/android-performance-battery/

Appendices