Battery Optimisation for a given Citrix App

Project progress report submitted

to

MANIPAL ACADEMY OF HIGHER EDUCATION

For Partial Fulfillment of the Requirement for the

Award of the Degree

of

Bachelor of Technology

in

Information Technology

by

Anuraag Palash Baishya
Reg. No. 140911296

Under the guidance of

Mrs. Aparna Nayak	Mr. Rajeev Dixit
Assistant Professor	Software Development Engineer (3P)
Department of I & CT	Mobile Platform Group (Automation)
Manipal Institute of Technology	Citrix R&D India Ltd
Manipal, India	Ulsoor Road, Bangalore, India

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

March 2018

1 Introduction

In a survey of around 50,000 people across 25 countries, conducted by International Data Corporation, it was found that battery life is the most important factor when buying a smartphone, with 56% Android users, 49% iOS users and 53% of Windows phone users stating the same.[1] Mobile device manufacturers and mobile operating system developers pay great attention to battery usage. While device manufacturers are adding larger batteries to their devices, Android has been given a Doze mode to reduce battery usage when phone is idle, along with a Battery Saver mode that disables aggressive network usage, animations, and other elements that drain battery, and iOS comes with a Low Power mode which essentially is similar to Android battery saver mode. Technologies such as Qualcomm Quick Charge and OnePlus Dash Charge have also been developed to reduce the time needed to charge a phone. Keeping such developments in mind, it has become very important for app developers to develop apps with optimized battery performance, so as to use as less battery as possible by default, i.e. even with battery life enhancing aids turned off.

2 Problem Definition

In this project, we will be working towards performing automated battery performance tests for a given Citrix Mobile app.

3 Objective

The objective of this project is as follows:

1. To gauge the current battery usage patterns of a given Citrix Mobile app
2. To determine factors within the app that drain more battery than desired

4 Scope

Beneficiaries of the work include:

1. The app development team who can utilize the results to optimize the apps battery usage
2. The app users, who will have a more battery efficient app

5 Background Theory

There are a large number of factors that determine the battery usage of a device. Some of them are system level, including brightness, hardware power consumption and operating system battery consumption, which are beyond the control of application developers.

However there exist a large number of factors such as CPU usage, network usage, wakelocks, etc which developers can look into, to optimize.

CPU usage results from performing computations. CPU usage is of two types, foreground usage, i.e when the app is running in the foreground, and background usage, for when app is running in the background (to perform tasks such as synchronisation, download, checking for mails, messages, etc). While foreground CPU usage forms a major part of battery consumption by cpu, if unchecked, a lot of unnecessary computation and processing maybe happening in the background, leading to battery drain (Unnecessary computation and processing may be taking place in the foreground as well)

Network usage, including mobile network and WiFi also form a major portion of battery usage. Like CPU usage, network usage is also of two types, background and foreground. Also like CPU usage, the app may be sending and receiving unnecessary packets in the foreground, background, or both. This is a potential candidate in making an app battery heavy.

Wakelock is a feature, which when requested by an app, lets the app to prevent the phone from going to sleep, i.e. the app will run continuously.[2] Wakelocks are handled differently in iOS and Android. While Android provides more control to the developer to invoke and use wakelocks through partial wakelocks, which lets apps run in the background, executing tasks, regardless of screen state or display timeouts, iOS permits partial wakelocks only for VOIP and location services.[3] It is pretty evident that an app requesting for unnecessary wakelocks will have a high energy consumption. This problem is greater for Android, due to availability of partial wakelocks.

There may be external factors in an apps battery consumption, such as bluetooth usage, camera usage or some other function whose power consumption is not directly controllable by the app. Such factors cannot be handled by the developers of the app, and hence such factors will not be taken into account. There may also arise unanticipated bugs in the app which lead to battery drain.

6 Methodology

The battery consumption of an app depends a lot on the device, its make, operating system version, age, etc, especially for Android devices, of which there are thousands of different devices. In contrast, iOS runs only on iPhones, and thus there are very few different models. However, since we will be performing tests to determine usage patterns and battery demanding elements, we will not need the absolute battery consumption levels, and a general overview of the apps battery consumption patterns will fulfil our requirements.

The test environment is as follows:
Device: Samsung Galaxy S6

Tests to Run:

1. Composing and sending emails (emails may contain attachments).
2. Run Build Validation Test (BVT) for running basic app operations. (Build Verification test is a set of tests run on every new build to verify that build is testable before it is released to the test teams)
3. Run detailed tests for problem areas discovered after running BVT tests

NOTE: These tests will be automated through scripts

Factors to be considered for Battery Performance Benchmarking:

- Checking the battery status before the test begins.
- Enabling the location services for the application (if application requires).
- Starting the data sync of the application.
- Checking if the application is sending/receiving the data when in the background.
- Observing the battery consumption while performing the above supported features by the application.

To test battery performance, we will use a tool, Battery Historian, developed by Google. Battery Historian analyses a bugreport taken from an Android phone and shows detailed battery use statistics, including CPU and Kernel uptime, running process information, mobile network, WiFi, GPS, JobScheduler, SyncManager usage details, and usage statistics for each app (such as CPU and network usage information, wakelocks, services, etc) that was running in the duration for which the bugreport was taken.[4]

For automation purposes, we will also develop a script which reports battery usage patterns, as Battery Historian has to be run manual. The tool will read the batterystats file and generate a HTML report showing wakelocks, wakeup alarms, jobs and syncs.

7 Workdone So Far

Setting up battery historian and UI Automator automation setup: I got Battery Historian (a tool to analyze power use on Android devices, developed by Google) running on my machine. Then I learnt how to generate battery statistics and bug reports using adb (Android Debug Bridge). Following are the adb commands to use:[5]

```
# to start the adb server
adb start-server

# to check connected devices
adb devices

```

```
# output of above command
# List of devices attached
# 02157df2d58dae03       device

#  find the IP address of device to connect to adb over WiFi
adb shell ip -f inet addr show wlan0

# output of above command
# 13: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP>
#  mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
#    inet 192.168.2.4/24 brd 192.168.2.255 scope global wlan0
#       valid_lft forever preferred_lft forever
# IP address is 192.168.2.4

# to use TCP for adb we have restart adb in TCPIP mode
adb tcpip 5555 # TCPIP port 5555

# connect to adb over WiFi
adb connect 192.168.2.4

# for resetting battery statistics
adb shell dumpsys batterystats --reset

# to take bugreport
adb bugreport bugreport.zip # for Android 7.0 Nougat and above
adb bugreport bugreport.txt # for Android versions prior to 7.0

# to get batterystats file for a particular app
# if the app package name is com.example.app
adb shell dumpsys batterystats com.example.app > batterystats.txt
```

Finally I learnt to analyze these reports using Battery Historian. I also setup Android Studio, UIAutomator and other tools such as Apache Ant and Robot Framework.

Running UI Automator tests and understanding the flow: I have been given access to the repositories for the app and the automation test suite. The test suite makes use of robot framework and UI Automator library. Hence I had o first train myself in both these tools.

UI Automator testing framework provides a set of APIs to build UI tests that perform interactions on user apps and system apps. The UI Automator APIs allows you to perform operations such as opening the Settings menu or the app launcher in a test device. The UI Automator testing framework is well-suited for writing black box-style automated tests, where the test code does not rely on internal implementation details of the target app.[6]

We are makking use of an open source python wrapper written for UI Automator. For UI Automator to work, an Android device has to be connected to the computer over adb via USB or WiFi. Since my work deals with analysing battery performance, I have to use adb over TCP as connecting adb via WiFi will continuously charge the battery, which leads to these problems:

- Charging the device leads to the batterystats data to be reset, hence information on battery usage patterns over the duration of the test is lost
- Even if the data was not lost, battery usage patterns will be erroneous if read from a device which is charging as we will not be able to gauge how much battery was actually consumed

Some sample code on UI Automator workings:

```
# connecting to ui automator

from uiautomator import Device
d = Device('014E05DE0F02000E') # device serial number
```

The device serial number is found after connecting the device to the computer and running the adb command 'adb devices'. Device serial numbers for devices connected via USB are generally alphanumeric strings, while for devices connected over WiFi the serial numbers are the IP addresses of the device. Note that for adb to work over WiFi, both the device and the computer have to be connected to the same WiFi network

```
# retrieving device information
d.info
```

A sample output for this could be as below:

```
{ u'displayRotation': 0,
  u'displaySizeDpY': 640,
  u'displaySizeDpX': 360,
  u'currentPackageName': u'com.android.launcher',
  u'productName': u'product',
  u'displayWidth': 720,
  u'sdkInt': 18,
  u'displayHeight': 1184,
  u'naturalOrientation': True
}
```

We are making use of UI Automator to mock how a user would use the app. So we are performing clicks and typing in text fields using UI Automator. UI Automator makes use of selectors to identify UI elements on the screen. There are many times of selectors including:

- resourceId, which is the ID of the UI element
- text, the text present in the UI element. We can also use textContains which finds elements which contain the text specified instead of looking for an exact match
- classname, which is the class of the UI element, such as TextView, ListView, etc
- Action based selectors such as checkable, checked, clickable, longClickable, scrollable, enabled,focusable, focused, selected

```
# sample code to perform a click
if device(text='OK').exists:
    device(text='OK').click()
```

Documentation for UIAutomator python wrapper can be found on GitHub. [7]

Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD). It has easy-to-use tabular test data syntax and it utilizes the keyword-driven testing approach. Its testing capabilities can be extended by test libraries implemented either with Python or Java, and users can create new higher-level keywords from existing ones using the same syntax that is used for creating test cases [8]. A sample robot test case:

```
*** Test Cases ***
User can create an account and log in
    Create Valid User    fred    P4ssw0rd
    Attempt to Login with Credentials    fred    P4ssw0rd
    Status Should Be    Logged In

User cannot log in with bad password
    Create Valid User    betty    P4ssw0rd
    Attempt to Login with Credentials    betty    wrong
    Status Should Be    Access Denied
```

Robot Framework allows use of keywords, which can be python functions. We will be using such keywords for our purposes. A simple example is as follows:

```
def sample_test_case(d, username, password):
        if d(text='username'):
                d(text='username').set_text(username)
        if d(text='password'):
                d(text='password').set_text(password)
        if d(text='ok'):
                d(text='ok').click()
# robot test case
Test username and password
```

```
  [Tags]    Test
  ${username}     User
  ${password}     Password
  Wait until keyword succeeds     2 min   0 sec
  sample test case    ${device01}    ${username}     ${password}
```

After learning how to use these tools, I wrote simple test cases for the app I have been given to work with. These test cases were solely written for the purpose of training and were not added to the actual code base.

Writing a shell script to send multiple emails: The task at hand was to write a script that sends a batch of 50 emails in half hour intervals, for a total of 150 mails over 1 and half hour.

Approaches tried:

Postfix: Postfix is a free and open-source mail transfer agent that routes and delivers electronic mail. First I learnt to setup a gmail relay using Postfix. This is the basic Postfix main configuration, which needs to be added at the end of the Postfix main configuration file (usually found at /etc/postfix/main.cf on UNIX based systems)

```
mail_owner = _postfix
setgid_group = _postdrop
relayhost = smtp.gmail.com
```

Most mail servers use SASL (Simple Authentication and Security Layer). Hence we need to configure postfix to enable SASL. This can be done by adding the following lines to the end of the Postfix main configuration file.

```
1 smtpd_sasl_auth_enable = yes
2 smtp_sasl_auth_enable = yes
3 smtp_sasl_password_maps = hash:/etc/postfix/sasl/sasl_passwd
4 smtp_sasl_security_options = noanonymous
5 smtp_sasl_mechanism_filter = plain
```

At line 3 above, we specify the file used for sasl authentication. File contents:

```
smtp.gmail.com username:password
```

Gmail additionally requires TLS (Transport Layer Security) to be configured

```
smtp_use_tls = yes
tls_random_source = dev:/dev/urandom
smtp_tls_mandatory_ciphers = high
smtp_tls_security_level = secure
compatibility_level = 2
```

This is based on an article found on HowtoForge [9]

Compatibility level causes Postfix to run with backwards-compatible default settings after an upgrade to a newer Postfix version [10]. By default, it is set to 0 to enable backwards compatibilty. But the default setting was causing issues with the working of Postfix. Hence I had to set it to 2, to disable backwards compatibilty, as suggested by the error trace.

To run Postfix, first we need to generate a Postfix lookup table by using the following command:

```
postmap /etc/postfix/sasl/sasl_passwd
```

Then the Postfix service has to be started:

```
service postfix start
```

Finally we can use this relay to send emails:

```
echo "This is a test." | mail -s "test message" user@mail.com
```

NOTE: To send emails using this relay, the sender must allow less secure apps to access gmail, which can be done from gmail settings.

Although this worked with the gmail SMTP, due to network restrictions on company network, I could not connect to the internal mail servers, leading to an 'Operation Timed Out' error

msmtp: msmtp is a mail client similar to Postfix. msmtp also needs to be configured. The follwing is a sample configuration file (the file is named msmtprc) [11]

```
defaults
tls on
tls_starttls on
tls_trust_file /etc/ssl/certs/certificate.crt
account default
host ""
port 25
auth on
user ""
password ""
from ""
```

We need to change msmtprc permissions to make it executable.

```
chmod +x msmtprc
```

msmtp usage is pretty similar to Postfix

```
echo "This␣is␣a␣test." |msmtp -t user@mail.com
```

However, since the issue was with the network settings for the company network, msmtp did not work either. After discussion with my guide, I was asked to move on to the next task and leave this for the time being.

Segregating the Test Cases (TCs) which test most commonly used UI interactions (viz. message compose, create event, contact) and add TAG as 'BatteryPerf': The UI Automation Suite already had test cases for almost all functionality available in the app. My task was pick test cases which test UI actions which are commonly performed by users. The test cases included Build Validation Test (BVT) cases and Function Validation Test (FVT) cases.
To get this done, I first had to understand the flow of work of the automation suite. The flow of code execution is as follows:

1. Build and run the MainDriver using Gradle

   ```
           ./gradlew build && ./gradlew run
   ```

   The MainDriver consists of all the arguments needed to run the tests. This file is used in the Jenkins automation server to run the automation process.

2. The MainDriver calls a shell script which sets all the arguments passed to the MainDriver as environment variables in a subshell, which are then used by robot framework when running the robot test cases.

3. The shell script sets the environment variables and starts the robot test cases.

4. Robot Framework runs all test cases which contain the tag specified and generates reports.

I ran the code, but I could not get it right for the first few times. Listed below are some problems I faced:

- Being new to industry level projects, I found navigating through the very deep levels of the project folder a little confusing at first.

- Since the MainDriver is used by Jenkins, where the arguments needed to be passed to the shell script are provided externally, the args[] array was commented out, and I did not know I had to uncommment this line to run the code locally. So since the shell script was being called without any arguments, no environment variables were being set and robot tests did not run.

- After I uncommented the args[] array, robot framework showed 0 test cases passed, 0 test cases failed. Trying to figure why this was happening, I looked into the environment variables and saw that they were not set and decided that this must be the reason. Hence I tried to set the values manually, which interfered with some of my systems necessary variables, leading to issues with the OS, and I had to give my machine to the IT services for reimaging. I did not know that the variables are set for a session of a subshell, to be used by robot framework and not at system level.

Once I received my machine back, I asked my guide to run the entire test suite once, to demonstrate how exactly it is to be run, and how to troubleshoot issues.

Post learning how to run the test suite, I looked into the test cases in the BVT and FVT files to determine which test cases would be appropriate to run for battery performance analysis. I originally shortlisted 20 test cases, but I was asked to restrict to 10-12 test cases. In the end, I shortlisted 12 test cases and added the tag 'BatteryPerf' to them. Now by specifying 'BatteryPerf' as the tag in the MainDriver file, I could run the battery performance test.

Below is the report generated for the tests I ran:

Table 1: General Statistics

Device estimated power use	0.36%
Foreground	156 times over 27m 5s 614ms
CPU User Time	4m 25s 40ms
CPU System Time	1m 42s 130ms
Device estimated power use due to CPU usage	0.0%
Total number of wakeup alarms	20

Table 2: Sync Information

Sync Name	Time (ms)	Count
Tasks Provider	566	1
Note Provider	399	1
Android Contacts	392	1

Table 3: Service Information

Service Name	Time	Starts	Launches
ExchangeService	30m 19s 270ms	8	25
EmptyService	1m 12s 846ms	20	21
AttachmentDownloadService	11s 616ms	27	27
PDLIntentService	7s 360ms	19	19
MailService	6s 843ms	16	16
FCMService	2s 859ms	1	1
ContactSaveService	324ms	2	2
CalendarProviderIntentService	168ms	2	2
AsyncQueryServiceHelper	120ms	2	2
ExchangeBroadcastProcessorService	59ms	1	1
EmailBroadcastProcessorService	43ms	1	1
AccountService	0ms	0	48
EasAuthenticatorService	0ms	0	1
PolicyService	0ms	0	19
LocalContactsSyncAdapterService	0ms	0	1
NoteSyncAdapterService	0ms	0	1
CtxAppManager	0ms	0	17

This information was obtained from Battery Historian

When I was running the test cases, I realised that for some cases emails were being sent to the test account. I consulted my guide, who told me they were sent using the Java Exchange Web Services API and a corresponding Python wrapper. I then looked into the feasibility of using these APIs to send the batch mails. Since it was feasible, I added the code for sending the batch emails and added a test case to the robot test cases file to send and verify batch emails. During developing this test case, one problem I faced was that during the 30 minute intervals between mail batches, since the phone was idle, the phone would go to Android Doze mode, which led to adb over WiFi disconnecting. To solve this issue, I polled the device once every minute to ensure that it does not go into doze mode. Although this is a deviation from a practical use point, this was necessary for us to run tests, and the polling hardly affected the battery usage as it does not do any computation that may require CPU or network usage. This test was primarily to check battery performance for sync operations. Results for this test are as follows:

Table 4: General Statistics

Device estimated power use	0.05%
Foreground	0 times over 1hr 42m 16s 412ms
CPU User Time	22s 30ms
CPU System Time	13s 934ms
Device estimated power use due to CPU usage	0.0%
Total number of wakeup alarms	0

Table 5: Service Information

Service Name	Time	Starts	Launches
ExchangeService	16m 40s 282ms	1	1
EmptyService	1s 8ms	1	1
AttachmentDownloadService	430ms	1	1
MailService	413ms	1	1
PDLIntentService	284ms	1	1
AccountService	0ms	0	2
PolicyService	0ms	0	1
CtxAppManager	0ms	0	1

It can be seen from this even though the batch email sync test case ran for a longer time, the device and the application, apart from running the sync service, were mostly idle and hence the battery consumption is minimal.

From the reports of the BVT and FVT test cases, it can be observed that the Attachment Download Service is running a large number of times (27), even though there were only two test cases that needed to use this service. On reporting this to my guide, I was asked to find the reason for this behaviour. After looking into the source code, excessively logging the workflow of attachment download and consulting the original author of the code, I learnt the following:

- The service is run once every time the app is launched, to check if any of the emails have an attachment, and to auto download the attachment, if auto download is enabled
- The service is also responsible for inline attachment download, and not just file attachments

This justified the large number of starts for the service. I also learnt that the auto download feature only worked when the device memory was less than 75% full. I was given the task to test if indeed the auto download behaviour was behaving as intended. To do this, I wrote a simple C program to allocate large amounts of memory to my program. While doing this I learnt that I needed to use a cross compiler, the ARM Cross Compiler for Android to be precise, as C programs are platform dependent, and hence a program compiled on a computer will not run on the phone.

To compile and run a C program on an Android device, we can follow the following steps:

```
#compile using ARM Cross Compiler
arm-linux-gnueabi-gcc -static -march=armv7-a test.c
#push generated file to device
adb push a.out /data/local/tmp/.
#run the program
adb shell "./data/local/tmp/a.out"
```

After filling the memory, I ran two attachment test cases and the auto download behaviour was as expected and auto download only worked when memory was more than 25% empty.

I also thought of running the entire battery performance test under low free memory conditions, but Android system automatically deallocates memory by shutting down background apps, processes and services when memory consumption becomes high, and hence my program could only keep the memory consumption high for the duration of the attachment tests, hence I could not run the entire battery performance test under low free memory conditions.

Initially I was supposed to carry out the battery performance test under network fluctuations and also on 3G/4G network. But due to unavailability of a SIM card, and issues with creating network fluctuations, these tasks have been suspended for now, and will be added later if a feasible solution is found

Creating a jenkins job and configure it to run Battery Performance automation: Jenkins is an open source automation server written in Java. Jenkins helps to automate the non-human part of software development process, with continuous integration and facilitating technical aspects of continuous delivery. The next task was to automate the entire battery performance test and to run it on the team Jenkins server, end to end without any manual work. For this I needed to do the following:

- Learn how to work with Jenkins, and get familiar with it.
- Develop a simple reporting tool to report the battery usage patterns, as Battery Historian (used previously) has to be run manually, and is also very verbose, which makes it unsuitable for our purposes
- Create a Jenkins job and run it

So, I downloaded Jenkins, and set it up a local Jenkins server on my machine to see and learn the workings of Jenkins. By default a local Jenkins server runs at localhost:8080. When started for the first time, Jenkins needs to be unlocked using a password, which can be found on terminal which is running Jenkins on UNIX based systems.

Steps to create and run a Jenkins job:

1. Create 'new freestyle project' from the 'create new jobs' option on the Jenkins dashboard and give it a name
2. We need to specify the location of files which need to be built. Let's assume that a local git repository(/Users/user/Project) has been setup which contains a 'HelloWorld.java' file. Select Git option and enter the URL of the local git repository. (We can also add the url of a remote git repository, or download and use the File System SCM plugin to directly use directories from the file system, which are not git repositories)
3. In the build section, add build step. Since I am working on a UNIX based machine, I needed to select 'Execute shell' option and add the command javac HelloWorld.java && java HelloWorld
4. Save and run the job
5. Check the output, logs or errors on the console available in Jenkins

Jenkins allows us to set parameters to be used as arguments while running a job. We will be using this to supply arguments to the MainDriver program, instead of specifying the arguments in the MainDriver file itself, as this gives the flexibility to change any arguments if needed before every run. Jenkins also allows setting of build and job run triggers. One such trigger could be committing code to the git repository associated with the Jenkins job.
Moving on, I worked on the development of the simple reporting tool, and testing it. The tool reads the batterystats file and generated a HTML report showing wakelocks, wakeup alarms, jobs and syncs. Screenshots from the tool are on the next page:

Battery Performance Report

General Statistics

Time on battery (realtime): 15m 7s 476ms

Time on battery (uptime): 15m 7s 476ms

Screen on: 15m 7s 476ms

Estimated Power Use (mAh)

Capacity: 2550

Computed Drain: 61.3

Actual Drain: 76.5-102

Screen: 35.6

Unaccounted: 15.2

Idle: 11.2

Wake Lock Information

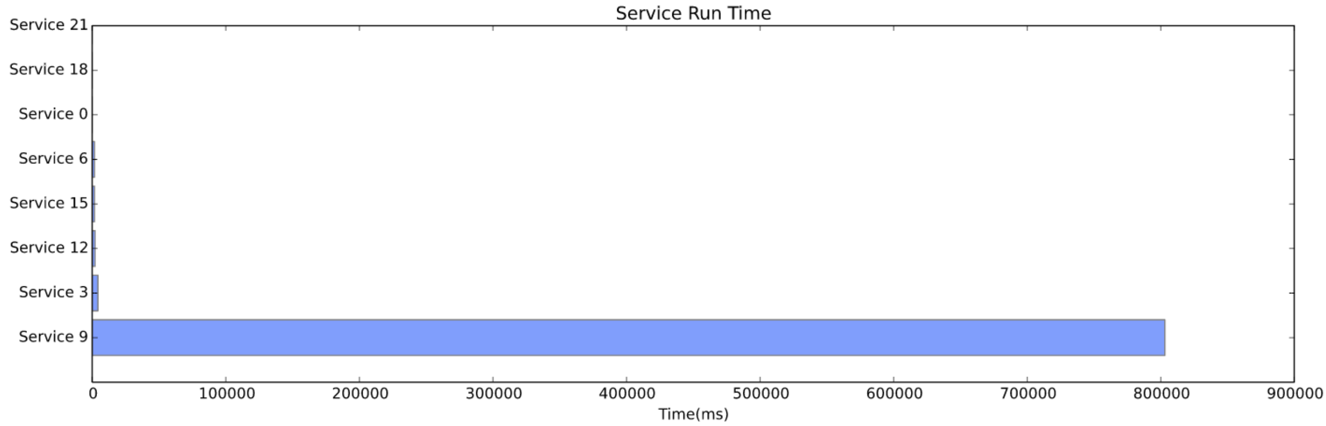
Wake Lock	Time(ms)	Count
alarm	0ms	0
launch	0ms	0
AttachmentService	0ms	0

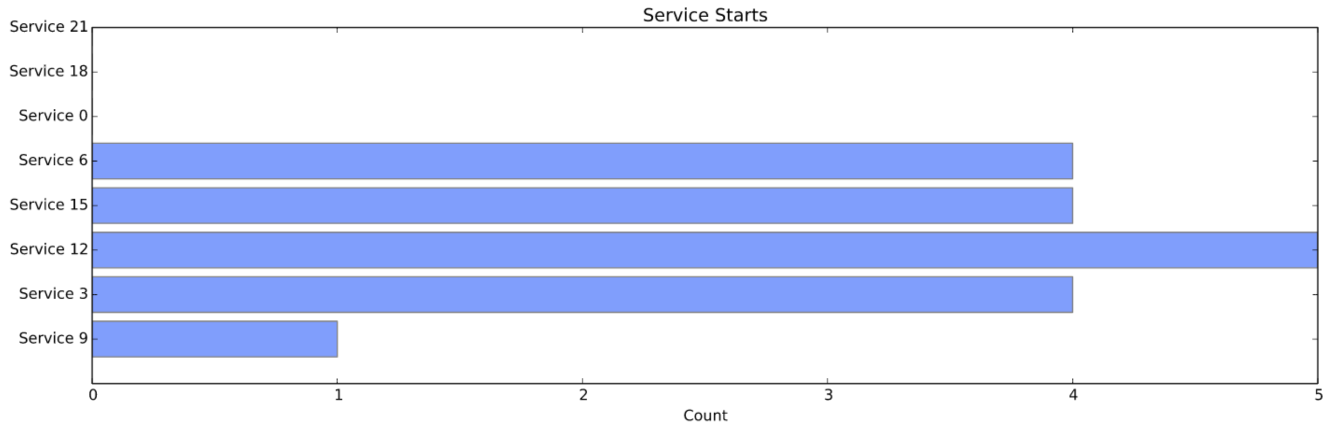
Wakeup Alarm Information

Wakeup Alarm	Count
Alarm	10

Service Information

Service	Time(ms)	Starts	Launches
Service 9	13m 23s 135ms	1	4
Service 3	4s 299ms	4	4
Service 12	2s 172ms	5	5
Service 15	1s 822ms	4	4
Service 6	1s 809ms	4	4





After developing this reporting tool, I had a discussion with my guide and my manager, and we decided that the attachment download service needs to be looked into again, to see if the large number of starts is consuming battery unnecessarily. To do this, I added two attachment test cases:

- To send a number of mails with attachments in one go, and then open the app and open the mails one by one and download the attachments, as an user would when they receive multiple mails together.
- To alternately send and view the emails (Send one mail, open the mail, download attachment and then send next mail)

Test conditions:

- Three cases were tested:
  - Mails with no attachment
  - Mails with a 133KB attachment
  - Mails with a 1.4MB attachment
- 20 mails were sent for every test case

Results for sending 20 mails in one go:

Table 6: Battery Consumption

Case	Battery Consumption
No Attachment	0.20%
133KB Attachment	0.35%
1.4MB Attachment	0.34%

Table 7: Wakelock Count

Wakelock	Start Count		
	No Attachment	133KB Attachment	1.4MB Attachment
EmailSyncAlarmReceiver	20	20	20
AttachmentDownloadWatchdog	1	20	20

Table 8: Service Running Time

Service Name	Time		
	No Attachment	133KB Attachment	1.4MB Attachment
ExchangeService	19m 43s 830ms	22m 53s 615ms	23m 19s 52ms
AttachmentDownloadService	1s 823ms	20s 355ms	36s 267ms
EmptyService	4s 308ms	4s 59ms	4s 213ms
PDLIntentService	2s 246ms	1s 976ms	2s 333ms
MailService	1s 808ms	1s 534ms	1s 742ms

Table 9: Service Start Count

Service Name	Start Count		
	No Attachment	133KB Attachment	1.4MB Attachment
ExchangeService	1	1	1
AttachmentDownloadService	4	99	106
EmptyService	4	4	4
PDLIntentService	7	6	4
MailService	4	4	4

Observations:

- Since the app was opened only once, for the case with no attachments, the Attachment Download Service ran only once to check if there are any attachments.
- With all other services running for same amount of time, it can be seen that increased running of the Attachment Download Service is a potential factor for the 0.15% increase in battery consumption when downloading attachments.
- The battery consumption might have also increased due to transfer of higher number of packets due to downloading of attachments, leading to higher network usage.

Results for sending and reading mails alternately:

Table 10: Battery Consumption

Case	Battery Consumption
No Attachment	0.38%
133KB Attachment	0.45%
1.4MB Attachment	0.44%

Table 11: Wakelock Count

Wakelock	Start Count		
	No Attachment	133KB Attachment	1.4MB Attachment
EmailSyncAlarmReceiver	20	20	20
AttachmentDownloadWatchdog	1	20	20

Table 12: Service Running Time

Service Name	Time		
	No Attachment	133KB Attachment	1.4MB Attachment
ExchangeService	15m 55s 733ms	19m 53s 615ms	21m 11s 931ms
AttachmentDownloadService	9s 418ms	28s 412ms	39s 842ms
EmptyService	24s 31ms	26s 991ms	24s 303ms
PDLIntentService	9s 423ms	10s 481ms	10s 781ms
MailService	8s 108ms	10s 4ms	11s 27ms

Table 13: Service Start Count

Service Name	Start Count		
	No Attachment	133KB Attachment	1.4MB Attachment
ExchangeService	1	1	1
AttachmentDownloadService	22	121	139
EmptyService	22	24	24
PDLIntentService	22	26	24
MailService	22	24	24

Observations:

- The app was launched, the mail read, attachment downloaded and then closed for every mail, the Attachment Download Service ran 22 times even for the case with no attachments, once every time the app was launched.
- With all other services running for same amount of time, it can be seen that increased running of the Attachment Download Service is a potential factor for the 0.06% increase in battery consumption when downloading attachments.
- The battery consumption might have also increased due to transfer of higher number of packets due to downloading of attachments, leading to higher network usage.

8 Remaining work

User Profiles: I have been asked to develop user profiles for three types of users, namely heavy users, moderate users and light users. For this I have to write a script which can be run against a Microsoft Exchange server and will return the count of attachments (both file and inline) and appointments for the past one month, as well as the number of contacts. This will be done using either the Java Exchange Web Service (EWS) API or the Python exchangelib EWS API. Work on this is mostly complete, although I have to fix a few errors and add exception handling to the script. The script will also automatically mail the counts to my email as attachments, with a common subject. After the script is completed, it will be run against exchange accounts of volunteers to gather the data. Once I have received the mails, I will run another script against my MS Exchange Account, to download all the data, and consolidate them into three files, one each for attachment data, appointment data and contact data. Then I plan to run a clustering algorithm on these files, to find the centers for three clusters, which will be determine the average attachment, appointment and contact count for the three types of users. Based on this data, we will modify the test cases to reflect all three types of users and determine the battery usage pattern for the three types of users.

Automating the entire process: The entire test will be run as a Jenkins job. This will consist of the following steps:

1. Run specific BVT and FVT Test Cases
2. Run Email Sync Script
3. Generate battery stats report and dump it in jenkins archive
4. Show the battery stats reports after performing the tests

References

[1] "IDC survey shows battery life is most important when buying smartphone", *GSMArena* [Online]. Available: `http://blog.gsmarena.com/idc-surveys-50000-people-reasons-behind-buying-smartphone-battery-life-deemed-important`

[2] "Keeping the Device Awake", *Android Developers* [Online]. Available: `https://developer.android.com/training/scheduling/wakelock.html`

[3] "Is there any type of partial wake lock mechanism in ios?", *Stackoverflow* [Online]. Available: `stackoverflow.com/questions/20580157/is-there-any-type-of-partial-wake-lock-mechanism-in-ios`

[4] "Analyzing Power Use with Battery Historian", *Android Developers* [Online]. Available: `https://developer.android.com/topic/performance/power/battery-historian.html`

[5] "Android Debug Bridge (adb)", *Android Developers* [Online]. Available: `https://developer.android.com/studio/command-line/adb.html`

[6] "UI Automator — Android Developers", *Android Developers* [Online]. Available: `https://developer.android.com/training/testing/ui-automator.html`

[7] "UIAutomator", *GitHub* [Online]. Available: `https://github.com/xiaocong/uiautomator`

[8] "Robot Framework", *Robot Framework* [Online]. Available: `http://robotframework.org`

[9] "Configure Postfix to use Gmail as a Mail Relay", *HowtoForge* [Online]. Available: `https://www.howtoforge.com/tutorial/configure-postfix-to-use-gmail-as-a-mail-relay`

[10] "Postfix Backwards Compatability Safety Net", *Postfix* [Online]. Available: `http://www.postfix.org/COMPATIBILITY_README.html`

[11] "How to use MSMTP with Gmail, Yahoo and PHP Mail", *Jesin's Blog* [Online]. Available: `https://websistent.com/how-to-use-msmtp-with-gmail-yahoo-and-php-mail`