Battery Performance and Automated UI Testing of 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

Ms. Ipsita Upasana	Mr. Rajeev Dixit
Assistant Professor	Software Development Engineer
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)

June 2018

DEDICATION

I dedicate this project report to my family for all their love, care and support; to the faculty, Department of Information and Communication Technology, Manipal Institute of Technology, Manipal, whose passion for teaching and educating have helped me learn and grow; to my guides and manager at Citrix R&D for guiding and helping me throughout the course of the project, and last but not the least to my friends, for always being by my side.

DECLARATION

I hereby declare that this project work entitled **Battery Performance and Automated UI Testing of given Citrix App** is original and has been carried out by me at Citrix R&D India, Ulsoor Road, Bangalore, under the guidance of **Mr Rajeev Dixit**, **Software Development Engineer**, Mobile Platform Group (Automation), Citrix R&D India Ltd, Ulsoor Road, Bangalore and **Ms. Ipsita Upasana**, **Assistant Professor**, Department of Information and Communication Technology, Manipal Institute of Technology, Manipal. No part of this work has been submitted for the award of a degree or diploma either to this University or to any other Universities.

Place: Manipal
Date : 28th June 2018

Anuraag Palash Baishya
140911296

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

CERTIFICATE

This is to certify that this project entitled **Battery Performance and Automated UI Testing of given Citrix App** is a bonafide project work done by **Anuraag Palash Baishya, (140911296)** at Manipal Institute of Technology, Manipal independently under my guidance and supervision for the award of the Degree of Bachelor of Technology in Information Technology.

Ms. Ipsita Upasana	Dr. Balachandra
Assistant Professor	Head of the Department
Department of I & CT	Department of I & CT
Manipal Institute of Technology	Manipal Institute of Technology
Manipal, India	Manipal, India

ACKNOWLEDGEMENTS

I am highly grateful to Dr. D. Srikanth Rao, Director, MIT, Manipal, Dr. P. Giridhar Kini, Associate Director (ILP & PS), MIT, Manipal, Dr. Balachandra, HOD, Department of I & CT, MIT, Manipal, and the Placement Office of MIT, Manipal for giving me the opportunity to do an internship at Citrix R&D.

I would like to thank Dr. Poornalatha G., Project Coordinator, Department of I & CT, MIT, Manipal for coordinating and liaisoning over the course of the internship, which enabled efficient communication. I would like to express my heartfelt gratitude to my guides Ms. Ipsita Upasana and Mr. Rajeev Dixit, and my manager Mr. Ankur Mathur for guiding me through the course of the project. I would like to thank Mr. Prashant Phatate (Software Development Engineer, Citrix R&D) for his valuable and timely inputs which have helped me overcome certain obstacles in the project. This project has given me first hand experience in the field of automation testing and has helped me in learning and sharpening various new skills.

I would also like to thank my parents and friends for their constant support and motivation.

ABSTRACT

Battery Performance Testing refers to performance of tests to gauge the battery usage patterns of a given app and to find areas which are consuming more battery than anticipated. Such tests can help fix battery consumption issues and make the app much more battery efficient. Automated UI Testing is the practice of testing UI components of an app to ensure that all UI elements are functioning properly and any action or series of actions performed by the user does not lead to app crashes or app freezing.

Battery consumed by an app is a very important factor for user acquisition and retention. Users are more likely to use an app when it is battery efficient. Hence, in this project a Battery Performance Testing Suite for a given Citrix App has been developed, which gauges the battery usage patterns and provides a detailed report on the same. The suite can also be used for comparison of battery consumption for different versions of the app.

The second part of the project involves adding a few new test cases and modifying few existing test cases in the Functional Validation Test (FVT) suite. A few functions were added and a few were modified in internal automation library used to perform Build Validation Test (BVT) and FVT. It also involves running BVT and FVT suites and ensuring changes don't break the test suites.

[**Software and its Engineering**]: Software creation and management – Software verification and validation – Process validation – Walkthroughs, Use cases

List of Tables

List of Figures

Contents

Chapter 1

Introduction

1.1 Battery Performance Testing

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.

1.2 Automated User Interface (UI) Testing

When apps are developed at enterprise level, they are highly complex and include a large number of possible User Interface (UI) elements. Testing all the components manually will require large amounts of time and energy. UI automation testing, is similar to manual testing, but instead of having a user click through the application, and visually verify the data, code is written to perform tests and verify results. Automated UI testing allows developers to "fail faster" which is a key component of agile development. Being able to identify errors sooner, gives developers more time to correct any issues long before your release. UI Automation tests can be re run as frequently as needed to test for any regressions or failures due to some changes in the source code.

1.3 Problem Definition

This project involves development of a Battery Performance Test suite for a given Citrix App, which can be used to gauge battery usage patterns of a given version of the app and perform comparative analysis of the battery usage patterns of two versions of the app. The project further involves addition of a few User Interface (UI) test cases to the Functional Validation Test suite for the given Citrix Mobile app.

1.4 Objective

The objective of this project is as follow

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.
3. To add few UI test cases to the Citrix App test suite and improve existing tests.
4. To monitor tests for any failures or breakages.

1.5 Scope

Beneficiaries of the work include:

1. The app development team who can utilize the results to optimize the apps battery usage and perform more extensive UI tests due to newly added tests.
2. The app users, who will have a more stable and battery efficient app.

Chapter 2

Background Theory

2.1 Battery Performance Testing

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 synchronization, 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.

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.

2.2 Automated UI Testing

Test Automation has become one of the important aspects in the Software development world, especially in places where agile development strategies are followed. Mike Cohn has developed the Agile Test Automation Pyramid[4] as shown in figure 2.1:

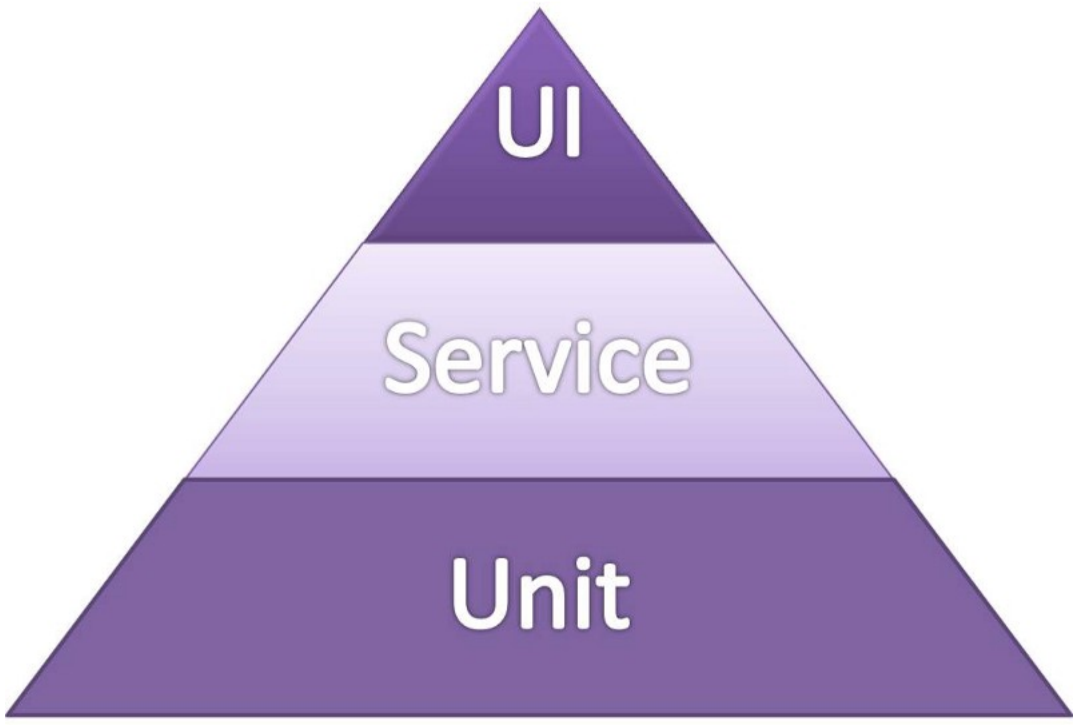


Figure 2.1: Agile Test Automation Pyramid

Unit tests form the base of this pyramid. These are used to test the different components of the software and form the major part of automation testing. The Service level tests form the next layer. These deal with integrating all domain and functionality tools. The User Interface (UI) tests form the apex and are responsible for testing longer-running, core customer usage workflows.
Below are a few cases in which UI test automation is valuable:

- Regression Testing. Automation can free human testers of the boring and repeated process of regression testing.
- Testing applications which do not have unit tests developed. This helps to introduce automated testing in legacy applications.
- Cross-browser and Cross-platform testing. This is mostly executing the same tests on different versions. This is also the case for mobile applications for testing different OS versions and device models.

- Performance Testing as it requires a higher load than that manual testing cannot generate.

Following are some criteria for selecting test cases for automation:

- Test cases which are frequently executed as part of smoke and regression.
- Test cases which implement complex logic and calculations.
- Test cases which need to be executed across multiple platforms.
- Test cases where the manual execution can be difficult eg. Performance.

Chapter 3

Methodology

3.1 Battery Performance Testing

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 tests will be performed to determine usage patterns and battery demanding elements, absolute battery consumption levels will not needed, and a general overview of the apps battery consumption patterns will fulfill requirements.

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.

Battery Historian, a tool developed by Google has been used to test Battery Performance. 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.[5]

For automation purposes, a script has been developed which reports battery usage patterns, as Battery Historian has to be run manually. The tool will read the batterystats file and generate a HTML report showing wakelocks, wakeup alarms, jobs and syncs. A script to generate a report showing difference in battery usage patterns for two versions of the app will also developed. The workflow diagram is shown in figure 3.1.

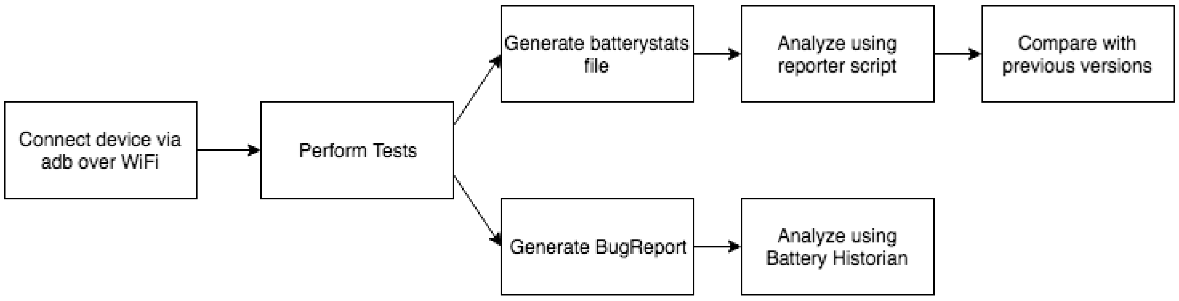

Connect device via adb over WiFi
Perform Tests
Generate batterystats file
Analyze using reporter script
Compare with previous versions
Generate BugReport
Analyze using Battery Historian

Figure 3.1: Workflow Diagram for Battery Performance Testing

3.2 Automated UI Testing

To create a test case, a use case that needs to be tested is identified. Once a use case is identified, the UI interactions required to complete the use case are recorded. Finally a test case is written to perform these interactions and verify if the desired result is obtained. The below example will illustrate this:

Suppose the use case is to compose and send an email with an attachment. The UI interactions to achieve this are shown in figure 3.2:

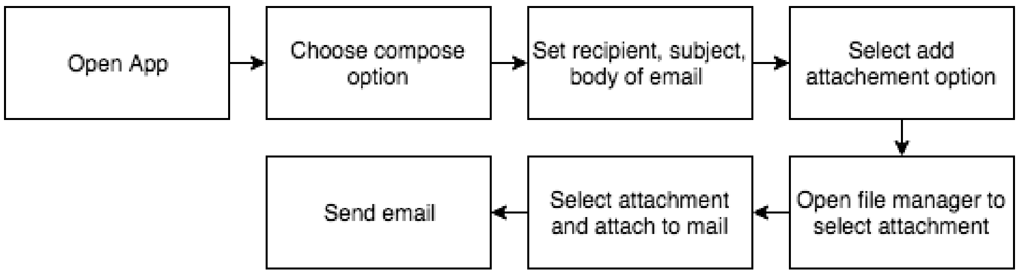

Open App
Choose compose option
Set recipient, subject, body of email
Select add attachement option
Open file manager to select attachment
Select attachment and attach to mail
Send email

Figure 3.2: UI Interactions to compose and send a mail with attachment

When writing the test case, each and every one of the steps shown above has to be automated. This is done using UIAutomator library's python wrapper[6] to automate clicks, swipes and other UI interactions. The recipient is set to the same account that is sending the email, so that the received mail can be used to verify if the email was sent correctly. When the sent email is received, it is opened and checked for presence of attachment. The test can only pass on one condition: the email has the correct attachment. In any other case, like missing UI element, email not sent due to network error, wrong workflow logic, etc. the test is marked as a failure. The workflow diagram is shown in figure 3.3.

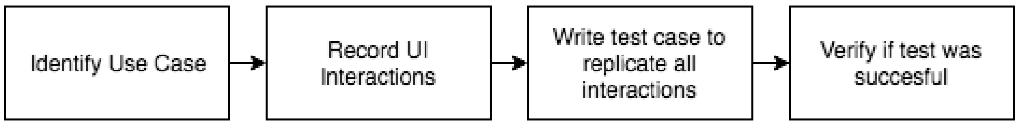

Identify Use Case
Record UI Interactions
Write test case to replicate all interactions
Verify if test was succesful

Figure 3.3: Workflow Diagram for Automated UI Testing

Chapter 4

Battery Performance Testing

4.1 Setting up battery historian and UI Automator automation setup

Firstly, Battery Historian (a tool to analyze power use on Android devices, developed by Google) was setup on the given machine. ADB(Android Debug Bridge) was also set up for the purpose of generating battery statistics and bug reports. Following are the adb commands to use:[7]

```
# 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, 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
```

These reports were analysed using Battery Historian. Android Studio, UIAutomator and other tools such as Apache Ant and Robot Framework were also setup.

4.2 Running UI Automator tests and understanding the flow

The existing test suite which is used in this project makes use of robot framework and UI Automator library. Hence, learning to use these tools was an important part of the project.

4.2.1 UI Automator

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 performing of 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][8]

Citrix makes 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 the project work deals with analysing battery performance, adb was used over TCP as connecting adb via USB 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 gauging how much battery was actually consumed becomes difficult.

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
}
```

UI Automator is also being used to mock how an user would use the app. It is used to perform clicks and type in text fields. 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. textContains can also be used, 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()
```

4.2.2 Robot Framework

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.[9] 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. Such keywords are being used in the test cases. 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, simple test cases were written for the app the project involves. These test cases were solely written for the purpose of training and were not added to the actual code base.

4.3 Segregating the Test Cases (TCs) and adding TAG ‘BatteryPerf’

The User Interface (UI) Automation Suite already had test cases for almost all functionality available in the app. The task at hand was to 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, first and foremost the flow of work of the automation suite had to studied. 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.

The code was run, but a few issues arose the first few times. Listed below are some problems faced:

- Navigating through the very deep levels of industrial level project folder can be 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. To run the code locally this line needed to be uncommented, which was not done due to lack of knowledge about the same. So since the shell script was being called without any arguments, no environment variables were being set and robot tests did not run.

- When the args[] array was uncommented, robot framework showed 0 test cases passed, 0 test cases failed. Trying to figure why this was happening, the environment variables were looked at and it was observed that they were not set, making it the potential reason for the above reason. In order to fix this the environment values were set manually, which interfered with some of the system’s necessary variables, leading to issues with the OS, and the machine had to be given to the IT services for reimaging. The variables are to be set for a session of a subshell, to be used by robot framework and not at system level.

These problems were mitigated with further research.

Post learning how to run the test suite, the next job was to look into the test cases in the BVT and FVT files to determine which test cases would be appropriate to run for battery performance analysis. The original shortlist had 20 test cases, but this list had to be further restricted to 10-12 test cases. In the end, the final shortlist had 12 test cases and the tag ‘BatteryPerf’ was added to these test cases. Now by specifying ‘BatteryPerf’ as the tag in the MainDriver file, the battery performance test could be run.

The report generated for the tests are shown in tables 4.1, 4.2 and 4.3.

Table 4.1: General Statistics obtained from Battery Historian

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 4.2: Sync Information obtained from Battery Historian

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

Table 4.3: 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

When running the test cases, it was observed that for some cases emails were being sent to the test account. These emails were sent using the Java Exchange Web Services API and a corresponding Python wrapper. The feasibility of using these APIs to send the batch mails was looked into and since it was feasible, code was added for sending the batch emails. A corresponding test case was also added to the robot test cases file to send and verify batch emails. During developing this test case, one problem 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, the device was polled 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 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 illustrated in tables 4.4 and 4.5.

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.

Table 4.4: 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 4.5: 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

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. This behaviour seemed erroneous and had to be looked into, to figure what was causing this issue. After looking into the source code, excessively logging the workflow of attachment download and consulting the original author of the code, following was learnt:

- 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. It was also learnt that the auto download feature only worked when the device memory was less than 75% full. Next, a test was performed to check if indeed the auto download behaviour was behaving as intended. To do this, a simple C program was written to allocate large amounts of memory to the program. While doing this a cross compiler, the ARM Cross Compiler for Android to be precise, needed to be used 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, following steps can be followed:

```
#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, two attachment test cases were run and the auto download behaviour was as expected.Auto download only worked when memory was more than 25% empty.

4.4 Running Battery Performance automation job on Jenkins

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 the following needed to be done:

- 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 the intended purposes.
- Create a Jenkins job and run it.

So, a local Jenkins server was setup 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. A ‘new freestyle project’ is created from the ‘create new jobs’ option on the Jenkins dashboard and it is given a name.
2. The location of files which need to be built is specified. Assuming that a local git repository(/Users/user/Project) has been setup which contains a ‘HelloWorld.java’

file, Git option is selected and the Universal Resource Locator (URL) of the local git repository is entered. (The URL of a remote git repository can also be entered, or the files can be downloaded and the File System SCM plugin can be used to directly use directories from the file system, which are not git repositories)

3. In the build section, build step is added. Since work is being done on a UNIX based machine, 'Execute shell' needed to be selected option and the command javac HelloWorld.java && java HelloWorld was added.
4. The job is saved and run.
5. The output, logs or errors are checked on the console available in Jenkins.

Jenkins allows users to set parameters to be used as arguments while running a job. This will be used 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, a simple battery usage reporting tool was developed for automation purposes, as Battery Historian needs to be run manually and hence cannot be integrated into the Jenkins job, and tested. The tool reads the batterystats file and generated a HTML report showing wakelocks, wakeup alarms, jobs and syncs. The reporting tool takes either a single batterystats file or a text file containing a list of batterystats files as input. In case a list of batterystats files is supplied as input, a separate report is generated for every file. The script primarily performs text mining on the batterystats file and stores the required information in dictionaries. These dictionaries are then used to create tables and graphs. The reporter script reports job, service, sync and wakelock information Screenshots from the tool are shown in figures 4.1 and 4.2.

After developing this reporting tool, the attachment download service was looked into again, to see if the large number of starts is consuming battery unnecessarily. To do this, two attachment test cases were added:

- 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

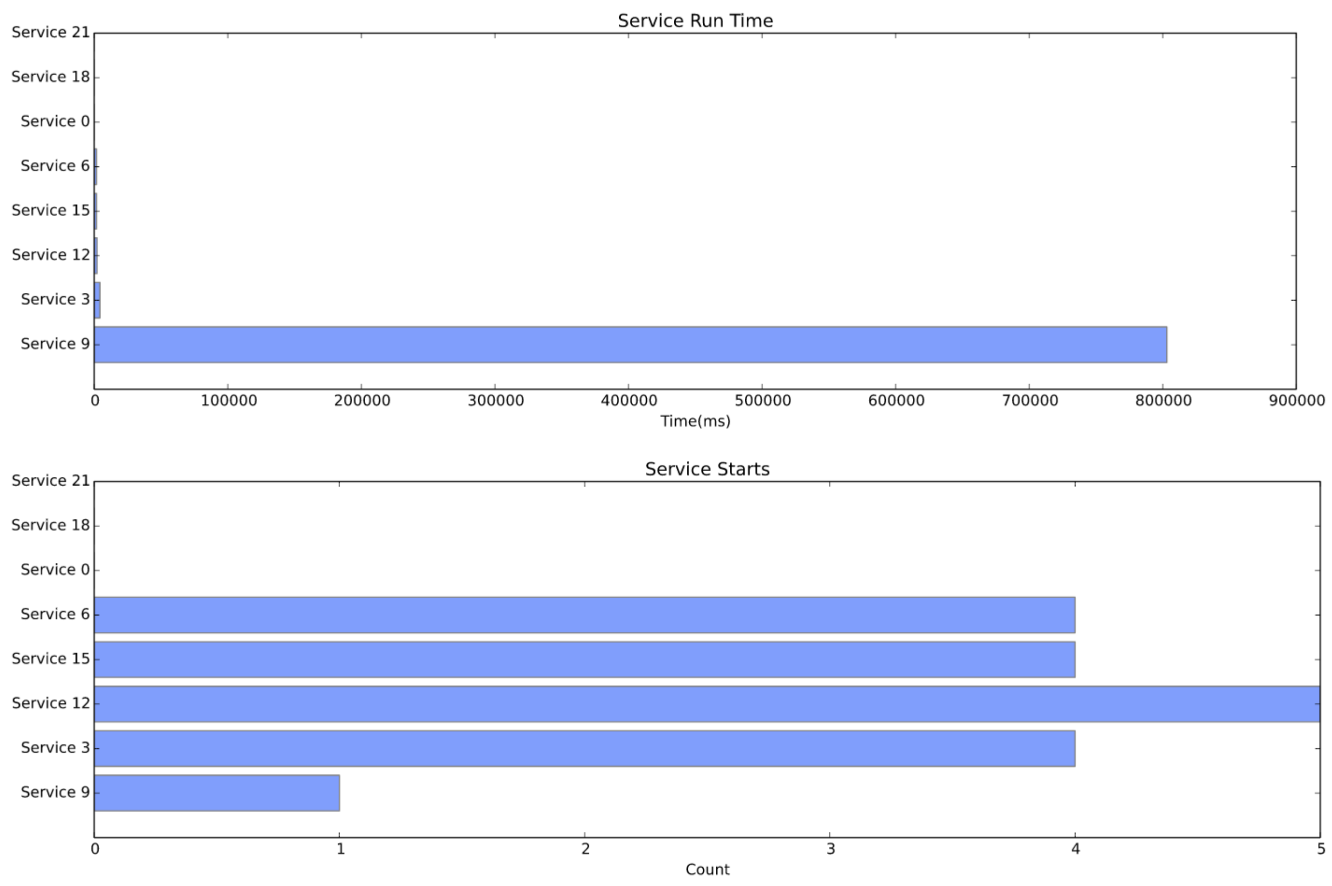
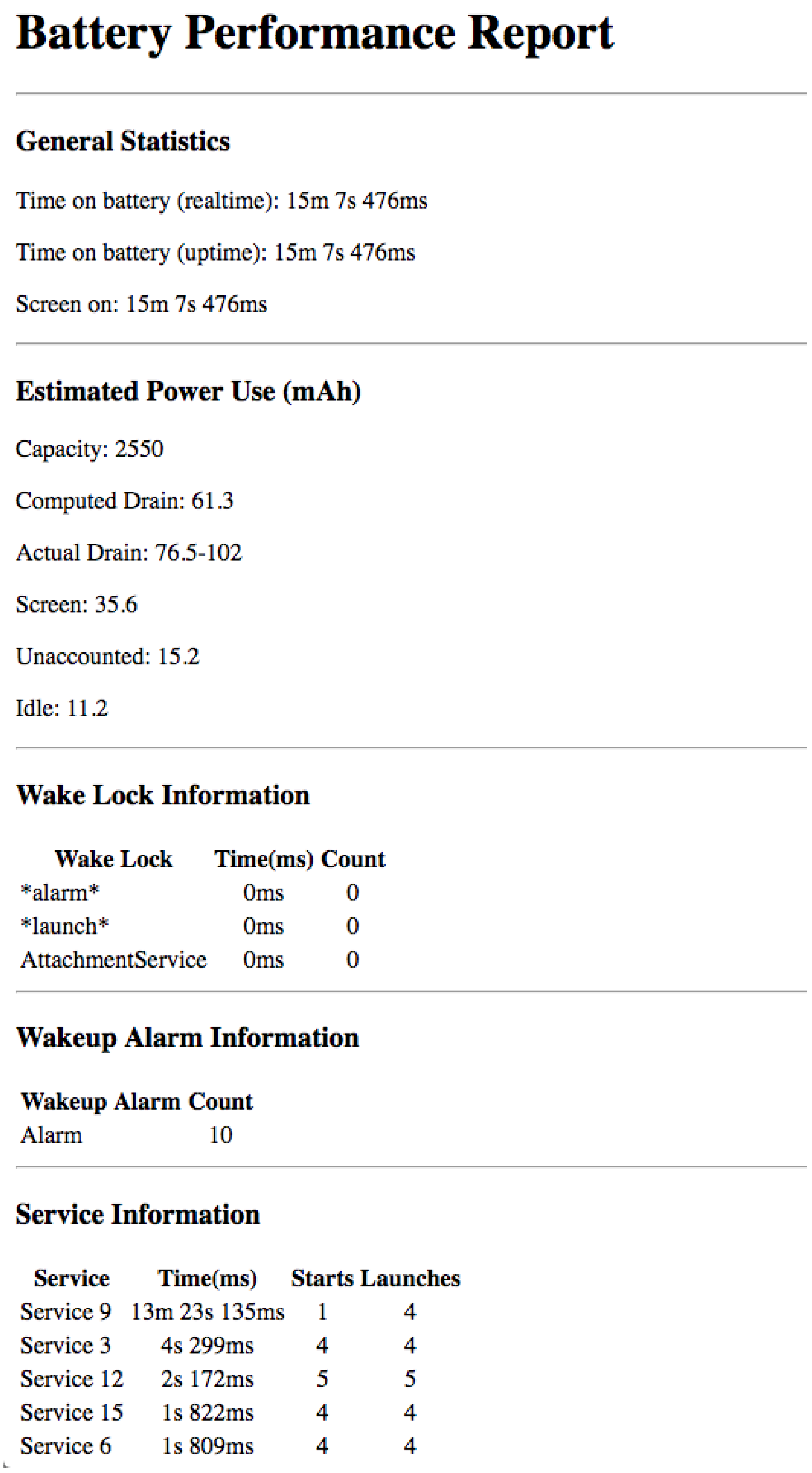


Figure 4.1: Service graphs
Similar graphs are generated for jobs, syncs and wakelocks



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

Figure 4.2: App Statistics

Results for sending 20 mails in one go:

Table 4.6: Battery Consumption

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

Table 4.7: Wakelock Count

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

Table 4.8: 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 4.9: 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.

Inferences:

- 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 4.10: Battery Consumption

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

Table 4.11: Wakelock Count

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

Table 4.12: 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 4.13: 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.

Inferences:

- 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.

4.5 Automating the entire process

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

1. Run specific BatteryPerf tests
2. Run Email Sync Script
3. Generate battery stats report and dump it in Jenkins archive
4. Compare battery stats reports for different versions to check for improvements/deteriorations in battery performance, and dump comparision report to Jenkins archive

To compare battery stats reports another script was developed, which makes use of the reporter script to generate the dictionaries. Then it compares the dictionaries and generates graphs for the differences. Screenshots of this tool can be seen in figures 4.3 and 4.4.

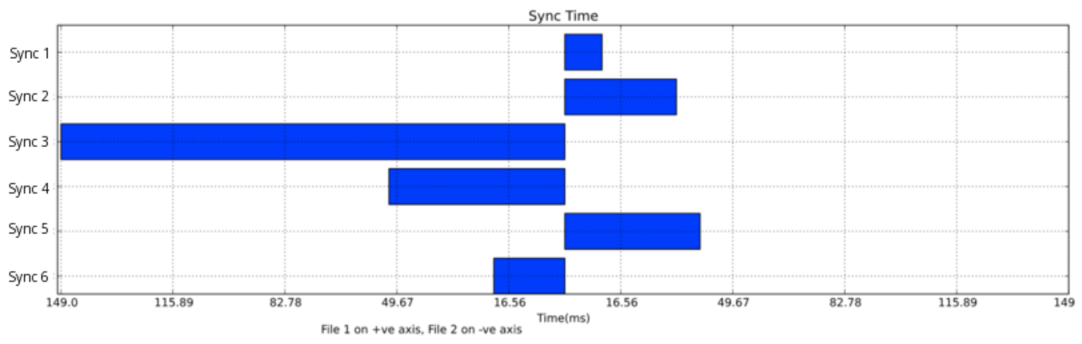


Figure 4.3: Sync Difference Graph
Similar graphs are generated for jobs, services and wakelocks

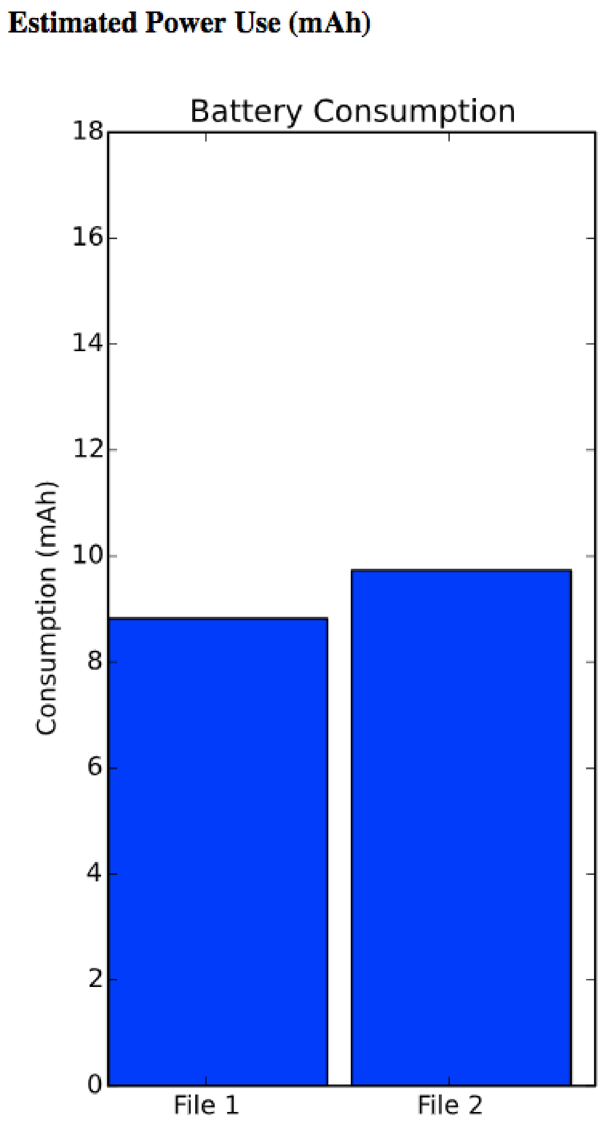


Figure 4.4: Overall Battery Usage

Chapter 5

Automated UI Testing

5.1 Converting Test Cases from Perfecto to UIAutomator

Earlier the automation suite ran on Perfecto, but the suite has now been migrated to use UIAutomator. A total of 6 functional verification tests (FVT) were left to be migrated to UIAutomator. The job involved converting these test cases. For this Java code had to be converted to Python, and helper functions needed to be added to the automation library (an internal library that contains functions which perform tasks needed by many test cases, such as search for attachment or take a picture). Test cases which needed work involved use of camera, attachments in emails and attachments in calendar events.

Citrix follows a process that any changes that may affect suite stability need to be verified by running the suite and making sure the stability is not affected. (Suite stability is the percentage of tests belonging to a suite which passed. So if 9/10 tests pass, stability is 90%). The tests added belonged to the FVT suite, but certain changes to the automation library code could affect Build Validation Test (BVT) too. Hence both these suites were run. The results are shown in table 5.1:

Table 5.1: BVT and FVT Results

Suite	Total Tests	Tests Pass	Tests Fail	Stability
FVT	46	37	9	80
BVT	26	26	0	100

The original FVT suite had 40 test cases. Since the suite was found to be stable, the new test cases were merged and now there are 46 test cases in the FVT suite as 6 more test cases were added.

5.2 Adding Watchers

Automated UI Testing follows a chain of steps. So whenever an element is not found, the test fails. Sometimes, random popup dialogs appear and obscure the UI resulting in

UI elements not being found, leading to failed tests. These pop ups maybe like "Battery Low" or "Rate the App", whose appearance is more or less random and cannot be predicted. Checks for these popup dialogs cannot be added to the test suite directly due to their unpredictable nature. For such cases watchers can be used. Watchers are a collection of steps which are triggered when a certain condition is met. When UIAutomator cannot find an element, it checks if any of the watchers' triggering conditions are met, and runs the watcher in such an event[6]. Watchers can also be used to eliminate checks for commonly occurring popups such as those asking for some confirmation.

A sample watcher would be like this. Suppose the user is dealing with "Low Battery" popup dialog which has an "OK" button.

```
d.watcher("lowBatteryWatcher").when(text="Battery␣Low")
                    .click(text="Okay") #watcher initialised
#test case code
#test case code
#test case code
d.watchers.remove() #remove watcher when work done

#the trigerred status of a watcher can be checked like this:
print d.watcher("lowBatteryWatcher").triggered
#returns true if triggered, else false

'''Watchers stay over sessions. So if a watcher is triggered
in a particular run it will stay triggered until reset.
To reset use'''
d.watchers().reset()

#NOTE: using d.watcher("name") only affects the particular watcher
#while d.watchers() affects all watchers
```

At first an analysis was performed to check if using watchers is feasible. Once it was found to be feasible, some functions in the automation library were modified to include watchers. Watchers were added for Samsung Knox screen and to replace multiple checks for "OK".

Samsung Knox is an enterprise mobile security solution pre-installed in most of Samsung's smartphones, tablets, and wearables. Because the given app would require device administration, Knox comes into affect and an user must confirm they agree with Knox terms and conditions. Most times, the Knox popup's appearance can be predicted and code to confirm Knox terms and condition was added to the tests that needed it. However it was observed at times the Knox popup appears earlier or later than when it's supposed to appear, leading to test failures. After the watcher was added, the test case can now confirm the Knox terms and conditions whenever the popup appears.

It was also observed that many test cases had multiple tests for the element "OK" after interactions that might cause a confirmation (such as deleting something). Tests were performed to see if all these checks could be replaced with a single watcher. The tests were successful and the checks for ”OK” were replaced with a watcher, which reduced the length of the test case code, making it more readable and easier to debug.

As done previously, the BVT and FVT suites had to be run to make sure they are stable. The results are shown in table 5.2:

Table 5.2: BVT and FVT Results

Suite	Total Tests	Tests Pass	Tests Fail	Stability
FVT	40	34	6	85
BVT	26	25	1	96

Since the suite was found to be stable, the original code has been replaced with the modified version. Changes to few other functions were also made to improve readability and make debugging easier, but these were minor changes such as creating function for repetitive code or removing unused code.

Action	Temp Table	Record Status	Main Table	Record Status	Remarks
Maker Actions:					
Creates a New Record	Y	N	-	-	-
Maker Modifies a New record(N -> M)	Y	N	-	-	-
Maker Deletes a New record(N -> D)	-	-	-	-	Hard Delete from Temp Table
Maker Modifies a Modified authorized record(A -> M -> M)	Y	M	Y	A	-
Maker Deletes a Modified record(A -> M -> D)	-	-	Y	A	Hard Delete from Temp Table
Maker Modifies a new Rejected record(N -> NR -> M)	Y	M	-	-	-
Maker Deletes a new Rejected record(N -> NR -> D)	-	-	-	-	Hard Delete from Temp Table
Maker Modifies a Modified Rejected record(N -> MR -> M)	Y	M	Y	A	-
Maker Deletes a Modified Rejected record(N -> MR -> D)	-	-	Y	A	Hard Delete from Temp Table
Maker Modifies an Authorized record(A -> M)	Y	M	Y	A	-
Maker Deletes an Authorized record(A -> D)	Y	D	Y	A	-
Checker Actions:					
Checker Authorizes a new record(N -> A)	-	-	Y	A	Hard Delete from Temp Table
Checker Rejects a new record(N -> R)	Y	NR	-	-	-
Checker Rejects a modified record(M -> R)	Y	MR	Y	A	-
Checker Authorizes a deleted record(A -> D ->A)	-	-	-	-	Hard Delete from Temp Table & Main Table
Checker Authorizes a modified record(A -> M ->A)	-	-	Y	A	-
Checker Rejects a deleted record(A -> D -> R)	Y	DR	Y	A	-

blablalbalIlablblablbala

Chapter 6

Challenges and Restrictions

When the project was conceptualized, it included certain tasks, which could not be completed due to some external restrictions. These restrictions include changes to Android Operating System, unavailability of required hardware and data, and network restrictions. The major challenges faced are explained below:

- **Writing a shell script for batch emails:** The Battery Performance test suite required mails to be sent out in batches of 50 with half hour intervals. For this a shell script was to be written using email transfer agents such as Postfix or msmtp. Although both these agents were configured and tested with a gmail relay, due to restrictions in the company network they could not be used to connect to the internal email servers of the company leading to an 'Operation Timed Out'error. After further research Java Exchange Web Server (EWS) API along with a corresponding Python wrapper was used to send the batch emails.

- **Running tests in low memory condition:** Originally it was planned to run 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 the program could only keep the memory consumption high for the duration of the attachment tests, hence the entire battery performance test could not be run under low free memory conditions.

- **Running tests under different networks:** It was also planned 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 removed from the project.

- **User Profiles:** The original project specification included the development of user profiles based on the attachment (both file and inline) count, appointment count for the past one month, and the number of contacts present in an user's email account. However it was realized that this approach would be highly erroneous as this would account for all attachments, contacts or appointments a user receives or creates irrespective of whether the client used (mobile app, desktop app, web app).

Since the project required only the statistics of mobile app, and since there was no way to aggregate statistics of only the mobile app, it was decided that a relative comparison would be performed, wherein every version of the app will be tested against a fixed arbitrary user profile.

Chapter 7

Conclusion

The internship majorly involved working in the field of automation. Technologies used during the course of the internship are as follows:

- Shell Scripting
- Python Scripting
- Robot Framework
- UIAutomator
- Jenkins

In this project, a Battery Performance Test suite was developed, which performs detailed battery usage analysis of a given version of the given app. The suite is also capable of finding differenced in battery usage patterns for two given versions of the app. Further new test cases were added to the Functional Verification Test (FVT) suite which performed tests related to calendar and attachments. Existing test cases were modified to account for unforeseen popups and remove redundant checks.

To perform Battery Performance Tests, existing test cases for most common user interactions were shortlisted. The test cases were run and the usage patterns for the same were analysed using Battery Historian. Standalone Python scripts were developed for automation purposes as Battery Historian had to be run manually. One script was used to read the BatteryStats file and generate battery usage reports and the other script was used to compare battery usage reports for two given versions of the app. The entire worflow was automated by creating a Jenkins job.

The new test cases added to the FVT suite were converted from Java to Python as the Java library was no longer being used and the current suite uses Python UIAutomator. A total of 6 test cases were migrated from Java to Python. Watchers were also added to prevent unforeseen popups from breaking test cases and to eliminate checks for commonly occurring popups such as those asking for some confirmation.

The Battery Performance Test suite is capable of determining battery usage patterns for a given version, compare battery usage patterns of two version of the app and also find areas which are consuming battery unnecessarily. During the project, the test suite had found areas that need to be looked into to make the app more battery efficient which were fixed in later versions. The test suite will be run for every new version to gauge and compare battery usage patterns and to find battery consuming areas in newer versions as well.

The new test cases added are now part of the Functional Verification Test (FVT) suite and will be run whenever the FVT suite is run. Further the modified test cases are now easier to read and debug due to removal of redundant code sections. These test cases also don't break when unanticipated popups appear as they are taken care of by Watchers.

References

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

[2] Keeping the device awake. [Online]. Available: https://developer.android.com/training/scheduling/wakelock.html

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

[4] M. Rouse. Agile test automation pyramid. [Online]. Available: https://searchitoperations.techtarget.com/definition/agile-test-automation-pyramid

[5] Analyzing power use with battery historian. [Online]. Available: https://developer.android.com/topic/performance/power/battery-historian.html

[6] Xiacong. Uiautomator. [Online]. Available: https://github.com/xiaocong/uiautomator

[7] Android debug bridge (adb). [Online]. Available: https://developer.android.com/studio/command-line/adb

[8] Ui automator — android developers. [Online]. Available: https://developer.android.com/training/testing/ui-automator

[9] Robot framework. [Online]. Available: http://robotframework.org

Table 7.1: Project Detail

Student Details

Student Name	Anuraag Palash Baishya		
Registration Number	140911296	Section/Roll No.	B/36
Email Address	anuraagbaishya@hotmail.com	Phone No.(M)	9538335985

Project Details

Project Title	Battery Performance and Automated UI Testing of given Citrix App		
Project Duration	6 Months	Date of Reporting	02-01-2018

Organization Details

Organization Name	Citrix R&D, India
Full Postal Address	Prestige Dynasty, Ulsoor Road, Bangalore, 560042
Website Address	www.citrix.com

Supervisor Details

Supervisor Full Name	Mr. Rajeev Dixit
Designation	Software Development Engineer
Full Contact Address with PIN Code	Prestige Dynasty, Ulsoor Road, Bangalore, 560042
Email Address	rajeev.dixit@citrix.com

Internal Guide Details

Faculty Name	Ms. Ipsita Upasana
Full Contact Address with PIN Code	Department of Information and Communication Technology, Manipal Institute of Technology, Manipal-576104
Email Address	ipsita.upasana@manipal.edu