ASIA PACIFIC INSTITUTE OF INFORMATION TECHNOLOGY

IN COLLABORATION WITH

STAFFORDSHIRE UNIVERSITY UK

B.Sc (Hons) in Software Engineering

Final Year Project

Android Ad-Hoc Network Monitoring Tool

By

Sankha S. P. Liyanage

CB005447

HF15A1CNSBEng

Date of Submission

18th July 2016

A project submitted in partial fulfilment of the award of the Bachelor of Science (Honors) in Computer Networks & Forensicss

Supervised By

Ms. Neera Jeyamohan

Project Manager Approval

……

Mr. Javed Ahsan

(Project Manager)

I certify that I have read this dissertation and that in my opinion it is adequate in scope and quality for the degree of Computer Networks and Security (BEng)

……

Ms. Neera Jeyamohan

(Supervisor)

I certify that this dissertation satisfies the requirements for the degree of Computer Networks and Security (BEng)

……

Ms. Shammi Hewamadduma

(Assessor)

ACKNOWLEDGEMENT

The author would like to express his deepest gratitude to the Project supervisor **Ms. Neera Jeyamohan**, Project assessor **Ms. Shammi Hewamadduma**, and Project Manager **Mr. Javed Ahsan** for their continuous guidance and support on this final year project. It is much appreciated the guidance and advices provided during problem encounters and doubtful situations.

Furthermore, the author would also like to thank APIIT (Asia Pacific Institute of Technology) library staff members for their support in providing the necessary resources and facilities required to make this project a success. Last but not least, the immense support given by my family members and friends for their support and love and also for being very helpful in various ways throughout the project duration, thus permitting to successfully complete the project.

Finally, the author is responsible for any errors that has been made in the project report.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

With the increased usage of ad-hoc networks, the necessity and importance for a monitoring tool for ad-hoc networks has become an essential feature. The ad-hoc networks are mostly implemented with mobile devices due to the high mobility and ease of setting up without dedicated hardware devices. As the mobile devices carry a limited amount of charge in their batteries, it has become an essential feature to monitor the device activity and efficiently manage the deice power consumption. During network sharing process, the usage monitoring is essential as ISP charge for the amount of bytes used. Since there is no proper implemented method to monitor the above components and a few available applications monitor individual components, the functionality covered by the existing tools are very limited. By analyzing the features and user experience with the existing applications, an application with functionality to monitor network activity, device activity and connected nodes has been implemented by the author to overcome the issues with the existing systems.

Keywords: mobile, ad-hoc networks, network monitoring, device monitoring

CHAPTER 1 - THE PROBLEM

1.1Problem Background

With contrast to infrastructure networking, ad-hoc networks have no dedicated routers or access points for network maintenance. In fact, they run on mobile devices where the host himself act as a node in the network. With the mobility, constant network changes result in frequent routing updates which has to be efficient in order to avoid routing loops and balance network load by notifying the other nodes the changes in the network.

Ad-hoc networks has the ability to add multiple devices for a single shared network and expand the broadcast domain. During the operation as a node; the hardware resources are utilized for general network activities not limited to routing updates, but also acting as an intermediate for data transmissions occurring between nodes which are not directly connected to each other. At the intermediate state, the node system resources are intensively utilized without the knowledge of the user; where power is a limited factor when running on battery power.

The **node device** client has to decide on the authentication process of the network; as other clients connecting to the network will also follow the same security settings. Due to the limited centralized nature of the ad-hoc network structure; the security of the node is solely a responsibility of the user. Lack of a methodology to monitor the number of packets sent and received for the session is a major drawback as the user has no idea about the network activity of the device.

Considering above factors, the following issues are identified as main problems existing within ad-hoc networks. Among identified issues, the absence of a proper software with the ability to monitor the resource utilization and power consumption of the node during an ad-hoc network session and vulnerabilities of security on the ongoing network session are identified.

1.2Problem Statement

During an ad-hoc network session, the users have to consider multiple factors from the process of implementation till the end of the network activity. In an ongoing network session, the user has least idea about the impact of power consumption and resource utilization on the node; as the mobile devices pack up a limited amount of energy within themselves. Though the user has no ongoing direct sessions with the network; the device is used as a node to maintain communication with other nodes; thus utilizing the resources without user's awareness.

As the nodes act as access points in ad-hoc networking, the security of the node has to be maintained at a considerably higher level, depending on the confidentiality of the data transferred. Depending on the limitation of constant SSID broadcast throughout the network activity; any host within the range of the active network can detect the broadcast signal and this can lead to clients with malicious activity to connect to the network actively and participate in the network activity while sniffing network packets pretending as an authenticated host.

With the high mobility of the network nodes, frequent routing updates are required for a smooth continuity of the network activity, thus utilizing the usable network bandwidth reducing the network efficiency and increasing node performance; significantly affecting the power consumption efficiency. The user has no proper methodology to monitor the number of packets sent and received within the network and amount of mobile data used. Only method to monitor is to rely on the built-in mobile data and wireless data counters which start the count from a cycle. The user has no way of checking the number of bytes sent or received within a session.

1.3Problem Analysis

Due to the unavailability of a proper monitoring system to warn and notify the user about heavy resource utilization; the device power consumption will become excessively high depending on network activity; eventually leading to a "dead node" which will adversely affect the user in return. With high resource utilization, heat is produced and mobile devices retain the heat within the device structure as they are closed systems from the hardware aspect. With the high heat produced and retained, battery drainage is quite at a high rate; adversely affecting the lifespan of the battery and the longevity of the device.

Considering the constant SSID broadcast, the hosts within the signal range of the active wireless network can connect to the network and participate in network activity if the network is unencrypted; revealing the packets transmitted within the network. If the network is encrypted using a standard encryption algorithm, the malicious node still has the ability to sniff the packets and passively capture ongoing sessions or hack into the wireless network using a wireless hack tool. The lack of node monitoring ability significantly affects the network if confidential and sensitive data are transmitted within the network.

Monitoring the total number of mobile and wireless bytes is essential as some users share the mobile data with a limited data bundle obtained from the ISP (Internet Service Provider). Exceeding the allocated data will incur heavy mobile bills on clients. The mobile and wireless data used by the host can be monitored using the device built-in data counters; but the existing method has no way of expressing the number of bytes transmitted for the current session, in fact consist of total number of bytes transmitted within a cycle; by default 30 day cycles.

CHAPTER 2 - PROJECT OUTLINE

2.1 Scope of the Project

Considering the major drawbacks in ad-hoc network systems, an application with abilities to monitor network utilization and power consumption is proposed. Other than ad-hoc network implementation and termination, the tool will have the ability to monitor ongoing network traffic and display the details such as upload and download rates, total uploaded and downloaded bytes and packets. The network activity monitoring, system power consumption is monitored through the battery temperature and voltage as mobile devices have power as a limiting factor. Functionality to monitor the connected nodes is implemented which will display their IP address, hardware address, reachable status and the connected adapter type. The system will be developed for the use with mobile devices operating on **Android platform** and the programming structure will be based purely on **java**. The tool will only be developed for **android operating system** and no support for other operating systems including iOS, Windows and Linux variants.

However, the tool will not cover the node monitoring for malicious behavior or to monitor the network for malicious activity. The proposed system will be implemented only on software, therefore the hardware security is ignored and securing the node and maintaining the network security is a responsibility of the user as the proposed software has no provided functionality to cover the node security. Monitoring on system resources will not have an impact on processing power of the device as the tool doesn't limit the device functionality, but will control the network components to conserve power and enable the longevity of the device.

2.2 Project Aims & Objectives

1. Monitor and display the upload and download rate per second, total data used during the current session and number of packets used for the network activity.
2. Monitor the resource utilization of the device and display the readings including the system details, device details and battery information.
3. Toggle between power saving and performance modes by actively enabling/disabling the mobile data connection and power saving modes respectively.
4. Monitor signal strength including the encryption type, network speed, SSID, MAC address, operating frequency and display the values to the user.
5. Notify the connected host details including the IP address, MAC address, and connected hardware type and reachability status.

2.3 The Solution Outline

Addressing the problems faced by ad-hoc network users, the lack of a proper system to monitor the network activity and resource utilization are identified as the main reasons and provides the foreground for the proposed tool implementation. The required components for the system are categorized with the aid of mind mapping effectively solving the issues faced by the ad-hoc network users. Initiation of the development of the proposed system will provide the users with the ability of monitor their device activity and have control of their device resource utilization; thus contributing to the power conservation of the device.

The system mainly consist of device resource utilization and power consumption, wireless network utilization monitoring and connected node identification. The wireless network monitoring process monitors incoming and outgoing network traffic status, notifying the network activity including the uplink and downlink speeds, total uploaded and downloaded bytes and packets. The solution application provides monitoring functionality regardless of its mode of operation. Monitoring the device power consumption is crucial as mobile devices pack a limited amount of energy for its operation, unless plugged into a power source. Considering the threshold values set up to monitor the battery temperature, remaining charge and plugged in source; the program will limit the network activity by enabling and disabling the mobile network connection and prompting the user

to enable power saving mode to conserve the remaining battery. The wireless connection will be suspended when the battery limit falls below 20%. Depending on the battery temperature, remaining charge, wireless network signal strength and mobile signal strength, the device activity will be notified to the user. Autopilot feature implemented will automatically control system resources and conserve the rate of battery drainage.

2.4 Assumptions made

During the upload and download rate implementation, the utilized TrafficStats API has no specific function or a method to implement the wireless upload and download packets. Assuming only mobile and wireless packets are exchanged through the network, the total uploaded and downloaded bytes are obtained by the following method.

- **Total transferred bytes**– **total transferred mobile bytes** = **wireless transferred bytes** displaying the number of uploaded/transmitted bytes
- **Total received bytes** – **total received mobile bytes** = **wireless received bytes** displaying the number of downloaded/received bytes

The total uploaded and downloaded packets are counted by following the same methodology as above. Upload and download rate per second is obtained by dividing the final calculation by 1000 milliseconds displaying the upload and download rate per second.

The wireless packets including authentication authorization requests, routing update packets and broadcast packets are neglected, assuming their activity is negligible compared to the ongoing network activity.

The sensor readings of the device are considered accurate and are displayed to the user; no specific hardware-software integration is implemented within, as the application is designed and developed mainly to monitor networks.

The auto pilot mode is developed in a way such that the processor and RAM utilization of the device is neglected and solely depends on battery readings including temperature, charge percentage and charging/discharging state. Considering the performance ability of late smartphones running on high performance processors; assuming the network activity processing

doesn't significantly affect the processing power of the device. The device activity section of the application is based on battery readings and upload download rates.

Android applications running on threads are suspended when changing the activity on back pressing and will affect the real time calculations of the application. Therefore the application will be minimized and will run on background alongside other applications running on the device.

2.5 Constraints

Considering the project aims and objective completion, the following limitations and problems and limitations arose during the implementation phase. The application is intended to monitor incoming and outgoing packets, but will not capture packets and save as PCAP for further analysis. The protocol analysis is ignored considering the battery limitation and high processing during packet analysis as the node has to decapsulate the incoming packets; analyze the required details in the header, re-encapsulate and forward to the destined user.

The implemented auto-pilot mode will be fully functional only on rooted devices; as root access only give permission enable and disable mobile network states. The application only has the ability to read the power saving mode enabled/disabled status, modification of the existing state is not supported by the APIs. The power saver enabled disabled state can be monitored from versions android **API 21** (**version 5.0 - Lollipop**) and upwards and doesn't support versions below **API 23**.

During high network activity, upload and download rates cannot be restricted due to its passive monitoring nature and has a limitation of controlling wireless network activity as there is no defined API support to have control of the network upload and download rates. The hardware address field of wireless Signal Strength return 02:00:00:00:00:00 for all devices running on android **API 23 (version 6.0 – Marshmallow)** as the function is deprecated by Google considering the node security reasons. (Developer.android.com, 2016). Other than the hardware address, the wireless signal strength (measured in dBm) and encryption type of the current connected network is deprecated on all devices running on android **API 23** and upwards.

Node monitoring is used to identify the connected nodes to the existing and includes host hardware and IP addresses, device hardware type and the reachability status. During the host mode operation, only the connected node details are displayed. All connected host details are displayed when the

device operates as a node. The tool has no added functionality to capture the network packets for further analysis.

The application testing is done on the following devices running on different android APIs. The application supports minimum API 18 till the latest API release. Testing carried out on rooted variants is limited due to the minimal availability of rooted devices.

- Samsung galaxy Tab 3 – API 19 (KitKat 4.4.2) ROOTED
- HTC desire EYE – API 23 (Marshmallow 6.0.1)
- Samsung galaxy Note 3 – API 22 (Lollipop 5.1.1)
- Samsung galaxy S4 – API 22 (Lollipop 5.0.1) ROOTED
- Samsung galaxy On5 – API 22 (Lollipop 5.0.1)

2.6 Deliverables

Power consumption activity will deliver the details of the system RAM, processor details, device information and battery details and will run on user request delivering real-time data at the point of request. The current battery temperature, charging/discharging status, battery technology, current voltage, battery health and power saving mode status are displayed in the "battery" category. The power saving mode status is only available for devices running on API 21 or higher. The "RAM" section will deliver the memory details will include the total available memory for device functionality, memory utilized by currently running applications and the remaining memory available for application usage.

Upload download status – runs at the device startup, total bytes are calculated by recording the instance of the application launch- the instance method is called. Upload/download per second is calculated by initial state-final state/1000. Total transmitted and received bytes and packets are calculated by adding the bytes/packets recorded at the instance with the initial counted stage. The text boxes are updated using a runnable implemented to update real time.

Enabling the declared auto-pilot mode will monitor the device for battery temperature, remaining charge capacity and plugged in source and will decide when to enable/disable the mobile data connection on rooted devices. The tool will prompt the user to turn off mobile data or enter power saving mode to conserve battery in non-rooted devices. The program enables the user to enable

and disable mobile hotspot by directing the user to the settings tab where wireless network settings are found. However, the redirection location slightly change depending on the manufacturer and android version device is currently running on. The auto-pilot mode will make decisions based on the following sensor readings and display the currently operating mode to the user. The application will suspend the existing mobile data connection prompt he user to enable the power saving mode under the following conditions.

- If battery temperature exceeds 45 degrees Celsius, plugged in to an AC source and the device is in not-charging status.
- If battery temperature ranges between 40 degrees Celsius and 45 degrees Celsius, plugged in source is USB and the device is in not-charging status
- If the battery temperature ranging between 35 degrees and 40 degrees Celsius, plugged in source is unavailable and if battery temperature reaches below 20%.

Other than the mobile network termination, the application will prompt the user to enter power saving mode if the battery temperature exceeds the given values. If the above mentioned conditions are met, the application will not continue participating in the network activity.

Node monitoring is carried out to **monitor the number of clients** connected to the network and display the connected hardware type, IP and hardware addresses and reachability status. This feature will provide the user functionality to identify the connected hosts with their hardware addresses. The criteria is implemented in such a way that the hosts connected will know the node details and the node hosting the network will know the connected host details.

The network signal strength details including connected network SSID, BSSID, hardware address, Wireless speed, wireless signal strength, encryption type used in the network, operating frequency and operating channel, IP address and default-gateway of the connected network are displayed to the user. However, the full functionality of signal strength display can be obtained only during the device operation as a host and has limited functionality, as the hotspot mode doesn't support the algorithms used. The wireless signal strength function and network encryption type detection has been deprecated and the hardware (MAC) address returns a value of 02:00:00:00:00:00 regardless of the root status on devices operating on android API 23. As explained in (Developer.android.com, 2016), the programmatic access to hardware identifier is deprecated from API 23 and upwards for applications using Wi-Fi and Bluetooth APIs.

2.7 Report Outline

The outline of the project is given below. Each chapter defined in the document elaborates specific sections related to the implemented application.

Chapter 1: The Problem

The section briefs on Project Background, Problem Statement and Problem Analysis sections.

Chapter 2 Project Outline

This section introduces the Scope of the Project Aims & Objectives, The Solution Outline, Assumptions Made, Project Constraints, Deliverables and Report Outline.

Chapter 3: Methodology

Methodology chapter includes techniques used to execute Requirement Analysis, Literature review, Solution Concept, overall Software Architecture, methods followed during implementation and Software testing phases.

Chapter 4: Requirement Specifications

The requirement specifications section includes resource identification chapter to identify hardware and software resources required for implementation and testing purposes, requirement analysis including functional, non-functional and additional requirements of application resources are defined.

Chapter 5: Literature Review

Includes domain research bearing the details of similar system study and technical research including deep research on the core components required for optimum functionality of the implemented system.

Chapter 6: Solution Concept

Describes different criteria required for research, implementation, testing and methods followed in order to achieve them.

Chapter 7: Software Architecture

Includes the use-case diagrams, activity diagrams including main menu activity, network activity and power consumption activity, class diagram and a detailed design of the software architecture.

Chapter 8: Implementation

Briefly explains the functionality of APIs utilized, methods and functions declared and Android Studio features used.

Chapter 9: Testing & Performance Evaluation

Includes the details of different test types carried out on the implemented system including Black-box testing, White-box testing and user testing. The effectiveness of the implemented system is discussed and an evaluation is carried out comparing the available features with currently existing tools.

Chapter 10: Critical Evaluation

A critical evaluation of the research, methods used to implement and test the system is defined.

Chapter 11: Contribution

Describes the author contribution to develop the existing systems, how far he has developed the features from the existing system, etc. is defined.

Chapter 12: Suggested Future Enhancements

Future modifications and development that can be carried on the implemented system is defined.

Chapter 13: Conclusion

The conclusion after the project completion is defined here.

Chapter 14: References

All the references extracted from websites, journal articles, books and other sources are defined in the references section.

Chapter 15: Appendices

Appendix A includes non-crucial black-box testing results while the Appendix B includes the Project time-line defining Gantt chart. Appendix C includes a few data gathering survey carried out before the system is implemented and the user feedback on the implemented system. Appendix D contains the additional supervisor and assessor meeting log sheets.

CHAPTER 3 – METHODOLOGY

3.1 Requirement Analysis

A requirement analysis is carried out before the proposed application implementation to collect information on the existing systems and create a useful and user-friendly application. The deliverables of the proposed system are decided on the user feedback on existing systems and include existing functions utilized by the users and their limitations and drawbacks. The requirements are achieved by following the methods described.

3.1.1 Interviewing and listening

The information required are gathered primarily by conducting interviews and questionnaires including the details of information on the uses of existing resources and how they are utilized by the users, up to what extent they are utilized and their limitations and drawbacks imposed on the activity. The user facts, opinions and speculations are gathered up and the current existing systems are assessed through that.

3.1.2 Choosing interview questions

The interview questions chosen should be related to the subject and they should cover up the limitations and drawbacks and record the user opinion on current existing systems. The questionnaire should contain **Open-ended questions** and **Closed-ended questions**. Open-ended questions cover the user opinion, their experience and suggestions on improvements of the existing systems. The closed-ended questions are carried out to obtain a broad range of information on the existing systems including their usability, drawbacks, improvement suggestions and opinion. Unlike open-ended questions, closed-ended questions focus on a set of questions designed to obtain a specified and more focused result from the user.

Closed ended questions can be of the following choices.

- True or false selection
- Multiple choice selection (where the user has one or more reasons to select on)

- Ranking the given criteria or rating a given response based on their experience with existing systems.

3.1.3 Interviewing as groups

The group interviews carried out on a specific task can yield more results than an individual interview; especially when considering systems involving multiple users. Considering the field of proposed application implementation, the networks are used by multiple users. Carrying out a group questionnaire will help solve existing limitations and drawbacks in the existing systems as the interaction between the users and the surveyor is high.

3.1.4 Requirement Specifications summary

The survey was carried out on 25 people, where few were users who create ad-hoc networks for file transfer, and many used to share internet connection between multiple computers, while the others were online multi-player gamers.

Considering the first criteria of using ad-hoc networks over infrastructure networks; most of the users find the benefit of easy setup and easy use without changing current adapter settings. Others used as a media of wireless file transfer and a few used it to tether internet connection as a Wireless hotspot. The surveyed clients used the wireless tethering services and file sharing services in ad-hoc networks; the second criteria described in the survey.

Multiple users suggested a method to switch between power saver and performance modes; as no such feature is available in the existing systems. The request was specifically made by the users who use ad-hoc networks to share internet connection. Without the ability to switch between modes, tethering is limited for a shorter time; minimizing the usability. The multi-player game users had no issue on the power limitation, but had issues on network signal loss with distance. With the lack of a method to monitor the CPU and RAM usage, which significantly affect the gaming activity, as the device operating system tend to lag, freeze or crash during intensive system resource utilization. Features to monitor the utilization of system resources including CPU and RAM usage, network usage and a signal strength indicator was requested by the online-gamer users.

A method to monitor the uplink and downlink was specifically requested by users who shared the mobile network through an ad-hoc network as the ISP bill the clients for data blocks used, thus making uplink and downlink monitor a crucial component of the system.

3.2 Literature review

Literature review category defines an evaluative study and a report generated on the selected study area describing, categorizing, summarizing and evaluating the literature; thus providing a proper base for the research. The defined literature review includes journal articles and periodicals from **IEEE Xplore**, **ResearchGate** and **ScienceDirect** published on world security and networking conferences. The required materials were obtained from e-books and printed books available on **Staffordshire University Online Library** resources and **APIIT library**. Other resource material were partially obtained from **official Android developer website**, **Google Scholar**, **Google Play** and **GitHub** to complete the literature review.

To complete the research, the literature review was carried out under Primary research and secondary research categories. The primary research category include similar approaches and their feature comparison, along with a brief definition on the area of research.

The secondary research include all the required core components on the field of interest. Starting from network architecture structures, the crucial components for efficient network activity are deeply discussed along with the protocol and its components used.

3.3 Solution concept

3.3.1 Core component election

The user analysis carried out has identified the major limitations and drawbacks existing in the system and considering the features available in the existing systems, the components for the proposed tool are selected. The upload and download monitor and battery monitor are identified as the core monitoring components and for easy user access, they are displayed on the “main menu” activity launched at the device startup. The core components are selected based on the

reviews of existing applications and the critical activities user was focusing on during the application usage.

3.3.2 APIs and methods selection

Considering the compatibility over multiple different **API levels**, the algorithms for the tool are written using the stock APIs accessible from the android kernel without modifications. This method is followed in order to build an application supporting all devices running on different API level variants. As android is an open source platform, the custom APIs and algorithms can run only on rooted devices, limiting the features for devices running on stock Operating System. Therefore, the decision was made to implement the application using APIs provided by google which are compatible with both rooted and non-rooted devices.

3.3.3 Component election

Analyzing the user reviews on the survey, suggestions were made to implement a system with the ability to monitor the upload and download rates, battery sensor readings, system resource utilization, signal strength monitor and a method to control the signal strength, power consumption. After analysis and thorough inquiry, the suggestions to control the device signal strength and system resource utilization are dropped out of the application scope. However, the RAM utilization is displayed to the user as RAM is extensively used by both system kernel and applications and is required for smooth operation of the device.

3.4 Software architecture

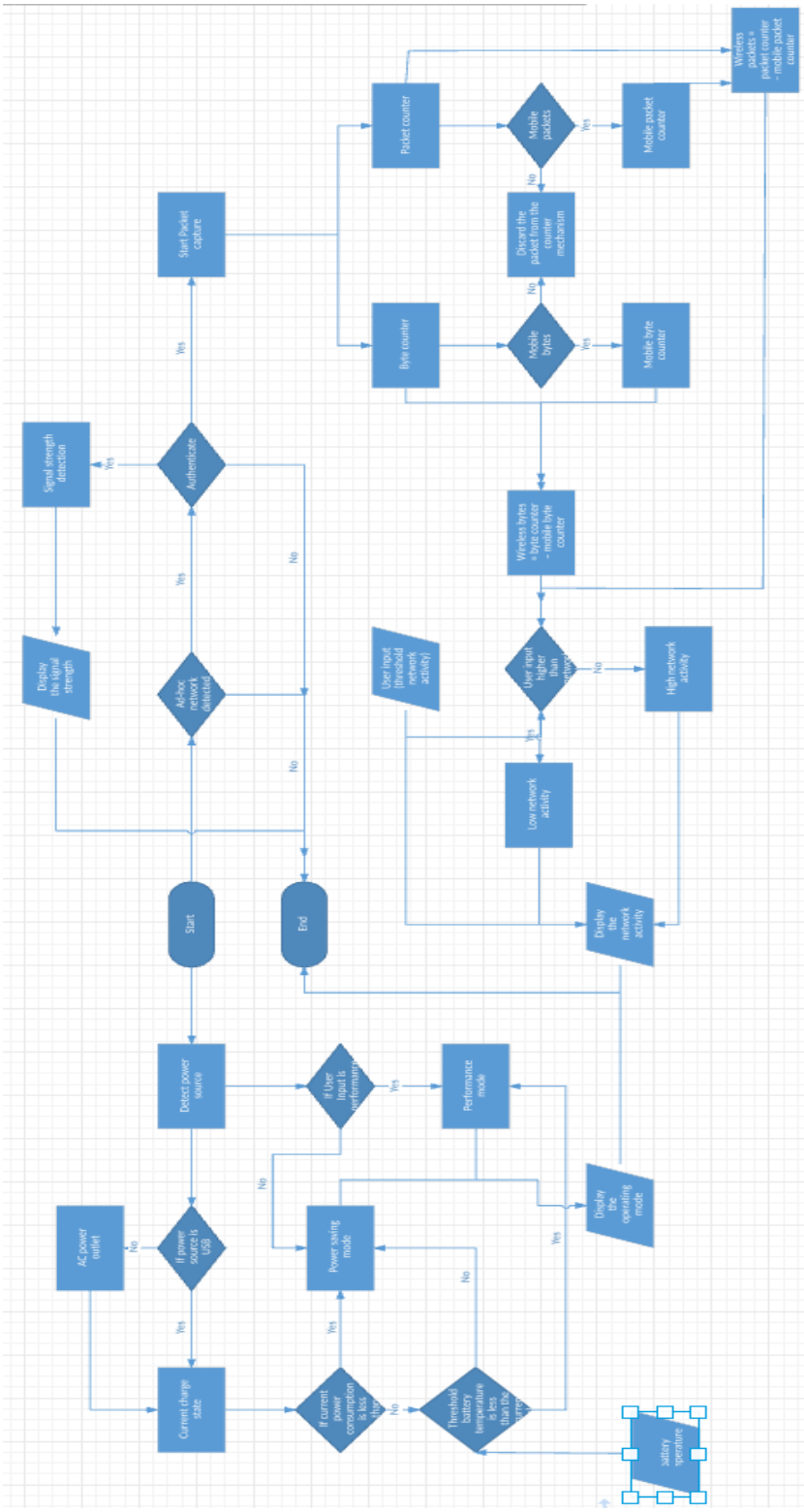

Figure 1- Overall Software Architecture

3.5 Implementation

Analyzing the user requirement criteria, the proposed application implementation is divided into phases and each phase is carried out under a time period defined in the project management plan.

Definition phase – the initial phase of the project and the criteria to achieve and deadlines are declared. Objectives for the project are obtained by analyzing the requirement analysis and primary and secondary researches. Depending on the requirement analysis and research, the feasibility of the proposed system is calculated.

Operation analysis phase – include the interface design which meet the project objectives and involves in identifying the crucial components and placing them accordingly within the system. System architecture is designed following the "Iterative methodology" in SDLC. The different phases include Requirement analysis, Design & Development, Testing and Implementation for individual sections monitored.

Build phase – involves coding and customizing the API methods, User Interface design and source code optimization, documentation of limitations faced during implementation and the enhancements made on the proposed design for optimum functionality. The enhancements are carried out by assigning appropriate load for a thread to handle and minimize resource utilization to execute application methods. The load on the system is reduced by minimizing the number of active threads at the application startup. The application is planned to assign new activities run on new threads and free the running thread and release back to the thread pool upon activity closing. The application is designed to display the most crucial details within the main menu activity itself, avoiding the hassle of switching between activities to obtain information on monitored components.

3.6 Software Testing

The usability and functionality of the designed system is tested using integration testing, unit testing and performance testing. Before implementation, the program is categorized according to the criteria to be met and a method within a category is tested following "unit-wise testing" and the returned values are observed before implementing another method within the same category.

Once all components within a section are implemented and tested, the integration testing is carried out on the completed components by integrating the methods together and testing simultaneously. Once the application passes unit testing and integration testing, performance testing is carried out to optimize the system resource utilization; which can include unit combination for efficient and smooth operation. The user accessibility testing is simultaneously carried out with performance testing; the program optimizations and additions are based on the questionnaire carried out to rate the application. The users who participated for the initial survey are gathered and the implemented tool is handed onto them to check whether their requirement criteria are fulfilled through the application. Their responses are recorded in the second questionnaire and an overall rating of the application is done to view the success rate of the implemented system.

CHAPTER 4 – REQUIREMENT SPECIFICATIONS

4.1 Resource Identification

4.1.1 Hardware Resources

For the application implementation and development, android studio is used as the application is developed to work in accordance with android operating system. The following hardware requirements are identified for android studio operation.

Minimum recommended hardware features for android studio

- OS - Microsoft windows 7/8/10 (32-bit or 64-bit), MAC OS X 10.8.5 or higher
- Processor - 1.5 GHz minimum
- RAM - 2 GB minimum, 8 GB recommended
- Disk space - 2 GB minimum, 4 GB recommended
- Screen resolution - 1280 x 800 minimum

Minimum requirements for the mobile device to run the application

- OS - Android version 4.3 or higher
- Chipset - ARM-based mobile chipset
- Processor - 1GHz processor
- RAM - 512 MB minimum, 1 GB recommended
- Disk space - 4 GB or higher

Hardware components on devices used for application development and testing

Android studio

- OS - Microsoft windows 8.1 (64-bit), MAC OS X 10.8.5 or higher
- Processor – quad core Intel Core i7-4500U 1.8 GHz
- RAM - 12 GB
- Disk space – 250 GB SSD
- Screen resolution – 1366 x 768 px
- Video memory – 4GB NVidia GeForce 750M

Two mobile devices are used for testing purposes, device 01 running on stock android firmware, device 02 is a rooted device with an OS upgraded to Android 4.4.2 (KitKat) from stock firmware 4.2.2 (JellyBean)

Tested mobile device 01

- OS - Android version 6.0.1 (Marshmallow)
- Chipset – Qualcomm Snapdragon 801
- Processor – Quad core 2.3 GHz Krait 400
- RAM - 2 GB
- Disk space – 16 GB

Tested mobile device 02

- OS - Android version 4.4.2 (KitKat)
- Chipset – Exynos 4212 Dual
- Processor – Dual core 1.5 GHz Cortex A9
- RAM - 1.5 GB
- Disk space – 16 GB

4.1.2 Software Resources

For android studio to operate smoothly, multiple software resources are required. The following software resources play a major role during the implementation, compiling and testing.

Intelligent code editor – based on IntelliJ idea, the code editor provides code completion, refactoring and code analysis simultaneously code predictions are suggested in a dropdown list improving productivity and efficiency.

Android SDK manager - different SDK tools, platforms and required crucial components for application development are provided.

Android SDK Build tools – Integrates tools required to build android applications to the Android Studio.

Android SDK Platform tools – tools required by the Android platform are integrated; including the Android Debug Bridge (adb) tool providing command line access to communicate with the emulator/ mobile device the application is tested on. (Developer.android.com, 2016)

(Developer.android.com, 2016)

4.1.3 API Resources

As described in the APIs and method selection section under Methodology section, the application is planned to develop on APIs available for devices running on factory-default operating systems. To meet the required criteria, the given API(s) are planned to use.

- Monitor the upload and download rates – TrafficStats API
- Monitor network status and enable/disable connections – ConnectivityManager API
- Display information related to wireless networks – WifiManager API, WifiInfo API
- Monitor battery related information and read sensor status – BatteryManager API, PowerManager API.
- Read the device information status and system resource details – ActivityManager API

4.2 Requirement Analysis

The author has carried out a questionnaire from different users who use ad-hoc networks or services related to it. The results of the questionnaire was used by the author to implement the key requirements of the proposed system. The key points of focus of the questionnaire is given below.

1. What the user find beneficial on ad-hoc networks over regular Wi-Fi networks

The user required benefits and advantages of using ad-hoc networks over infrastructure networks (Wi-Fi) on different situations are defined.

2. What are the features/functions they use in ad-hoc networks

The ad-hoc network functionality used by the users are defined; the users are questioned for their purpose including network utilization for file transfers, online game activities or mobile network sharing and the results are documented.

3. Are there any specific requirements to improve the services

The ad-hoc network users are asked whether they are satisfied with the existing technologies available and whether they require further development or enhancement of the existing features.

4. What are the drawbacks of ad-hoc networks

The drawbacks the users face during their engagement with ad-hoc networks are documented.

5. What the user will find easy to utilize the ad-hoc networks

The user opinions on easy ad-hoc network utilization are questioned and their responses are documented.

4.2.1 Functional Requirements

The functionality to monitor the network upload and download rates:

- The application should be able to detect any ongoing network activity regardless its mode of operation; as a node providing connectivity to other devices or as a host connected to an existing network

The application should be able to count the total uploaded/ downloaded bytes and packets for a defined session

- The defined application should include functionality to count the total bytes and packets received and transmitted within a defined session.

User accessibility to view the device system details of the device

- The user should be able to access and view the details of the device, regardless of its operation mode as a node or as a host.

View the power usage of the device

- Regardless of the operating mode, the functionality should be provided for the user to monitor the power consumption details of the device.

Display the details of the connected network.

- The network details are displayed to the user connected as a host to the network. No network details are displayed if the user is a node of the network.

Ability to monitor the connected nodes to the implemented network

- The application should be able to provide functionality for node devices to monitor details of the connected hosts. The hosts should be notified with the details of the node device they are connected to.

Functionality to programmatically control the device

- The user should be provided with the functionality to programmatically control the device resources which significantly affect the remaining charge and battery life of the system
-

4.2.2 Non-Functional Requirements

The application functions should be easily accessible by any user

- The users using the system should be able to monitor the required components and should be able to easily understand the layout of the application

Efficiency and performance should be high

- The application should be able to handle multiple functions simultaneously and the user should be provided with a very smooth user experience during their engagement with the application.

Reliability

- Provided that multiple functions are utilized simultaneously, the application should provide accurate details of the ongoing session.

Speed

- The application should maintain the initial speed at the application start up throughout the application, regardless of the load on the system with multiple functions running simultaneously.

The application should be able to handle multiple operations simultaneously

- Functionality to run multiple threads simultaneously without affecting the device performance and application performance should be maintained throughout the session.

Availability

- The application should be available to any user with an android device running on API level 18 and above. As the tool is designed to monitor the network activity, an active network connection is mandatory.

User interface

- The user interface should be simple, yet provide the major monitored components within a single activity for ease of use; the non-crucial monitoring components are displayed only upon user request.

4.2.3 Additional requirements

As additional requirements, a switch to enable and disable auto-pilot mode and a switch to enable and disable hotspot are included within the application. Providing the required functionality to enable and disable the hotspot within the application eases up the user' to navigate to settings tab to enable and disable the hotspot accordingly. The user is given the option to turn the auto-pilot mode on and off according to their preference by linking the methods and functions declared to control the device. Rather than integrating the HotSpot controlling methods to operate independently, high interaction between the user and the application is enabled by allowing the user to have control of the device through the application.

CHAPTER 5 – LITERATURE REVIEW

DOMAIN RESEARCH

5.1 Similar Approaches

	MANET manager	Wi-Fi tethering	FoxFi
Features	Turns the mobile device into a peer-to-peer communication device.	Create ad-hoc network with user-defined IP addresses Select an encryption algorithms and select the transmission power. Creates a peer-to-peer communication device	Create ad-hoc networks using USB, Bluetooth or via the Wi-Fi adapter. Creates a peer-to-peer communication network
Requirements	Multiple devices with ad-hoc connections Root access Kernel supporting wireless extensions (wext) Wireless device supporting ad-hoc mode	Multiple devices with ad-hoc connections Root access Kernel supporting wireless extensions (wext) Wireless device supporting ad-hoc mode	Multiple devices with ad-hoc capabilities No need for root access Active Wireless internet connection
Drawbacks/ weaknesses	No signal strength detection	The application has no option for the user to switch between different modes (Ex:	The application has no option to select

	No data transfer speed notification No processing and power consumption indicators and user has no control over the transmission	power saving and high performance) No signal strength detection User has control only over the transmit power selection	between different power control modes No signal strength detection The free version application requires restart after some usage.
Sources	(Play, 2015), (Code.google.com, 2015)	(Play, 2015),	(FoxFi, 2012), (Play, 2015)
Project information	GNU GPL v3	N/A	N/A

Table 1 - Similar approaches feature comparison

5.2 Ad-hoc Networks

Before going into details, it is important to define the meaning of an ad-hoc network. Upon numerous definitions available, the author has selected the best definitions which define ad-hoc networks. Defined by Margaret Rouse, 2016 as "An ad-hoc network is a local area network (LAN) that is built spontaneously as devices connect. Instead of relying on a base station to coordinate the flow of messages to each node in the network, the individual network nodes forward packets to and from each other. In Latin, '*ad hoc'* literally means 'for this,' meaning 'for this special purpose' and also, by extension, improvised or impromptu".

A basic ad-hoc network creates direct connections with the hosts without the need of a router or base stations; where the individual nodes transmit the data packet. The packets are transmitted under the IEEE 802.11 WLAN architecture which require processing at the nodes; especially for de-capsulation and re-encapsulation processes. To monitor the system resources utilization, the available tools don't provide enough functionality. Therefore, the author has proposed a tool to monitor network activity and meet the criteria missed by other monitoring tools.

TECHNICAL RESEARCH

5.3 Data Communication Network architecture

Data communication is the data transfer which occur between a system of computers or computer peripherals located between two or multiple locations. As described by (Forouzan, 2007), data communications are the transfers of data from one device to another via a method of transmission and the transmission must take place in a timely and an accurate manner to the correct destination; where the transmission can be simplex, half-duplex or full-duplex transmission forms.

Network architecture refers to the services and components required for the data transmissions which occur between two or multiple computer systems. The services and components collectively form the network services which help the data transmission for its functionality.

According to functionality and operation differences, the author has selected the two most related architectures as

5.3.1 Client—Server Architecture

The client-server architecture consists of a distributed communication framework where network processes include service requestors, clients and service providers; where the connections between the clients and the servers are established with the aid of a network.

The client-server model provides the core functionality for email exchange and web based database access. The following protocols and services are built around the client-server architecture.

Hypertext Transfer Protocol (HTTP)	Domain Name System (DNS)
Simple Mail Transfer Protocol (SMTP)	Telnet services

The client-server architecture can be categorized into two main types according to its arrangement.

5.3.1.1 two-tier Architecture

Two-tier architecture consist only servers and clients; where the servers play the role of database along the other services provided for the network activities. The client and server maintain direct connections with each other as servers maintain databases while the clients request access to the information stored in the databases.

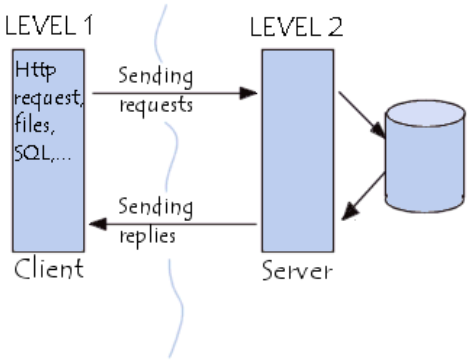

Figure 2 Two-tier Architecture (Knowlin, 2016)

Sources: (Knowlin, 2016)

5.3.1.2 three-tier Architecture

The three-tier architecture has an intermediate level linking the server application and the client application. The client application has to make connection requests to the application server which will then link to the server level. A reply from the server application has to link with the application layer to send the reply to the client. The application server is also known as the "middleware" whose task is to provide the requested resources by contacting the relevant server and vice versa. (CCM, 2016), (Knowlin, 2016)

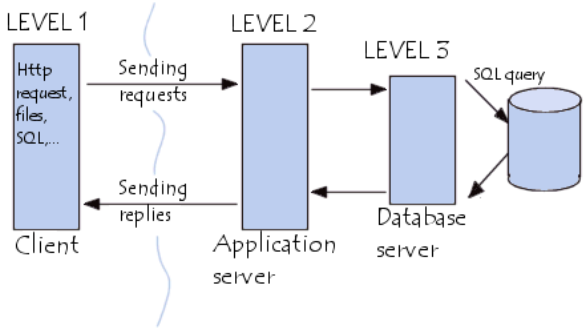

Figure 3 - Three-tier Architecture (CCM, 2016), (Knowlin, 2016)

5.3.1.3 Peer to Peer Architecture

Peer to peer communication architecture involves two or multiple devices communicating with each other directly without connecting to a separate server. The nodes in the network has equal responsibility and capability to actively take part in the data transmission session. As dedicated servers, routers or access-points are unavailable, the nodes have to dedicate its power and system resources for the network activity. Peer to peer networks; also known as P2P networks can be further categorized into

5.3.1.3.1 Pure peer-to-Peer Networks

In pure peer-to-peer networks, all nodes participate in the network activity equally playing client and server roles; where a central server to help control, co-ordinate or manage the packet exchange among peers is absent. Considering the application Freenet designed by Mike Godwin maintain communications by Freenet nodes which are encrypted and routed through other nodes connected. This decentralized approach makes it extremely difficult to determine who is requesting what information. (Schollmeier, 2002), (Godwin, 2016)

5.3.1.3.2 Hybrid peer-to-peer networks

A hybrid peer-to-peer network consists of a central server to support specific functionalities which comes under **administrative** category. To understand how a hybrid peer-to-peer network work, the Torrent applications (Eg: BitTorrent, UTorrent) have a central server to update the tracker list for the user node which are scattered among other users. This is known as the **tracker** and helps by coordinating the communication between nodes to complete a download. (Cope, 2002), (Carmack, 2005)

Comparing the two types of peer-to-peer networks, the hybrid peer-to-peer networks have a central server performing certain administrative functions while no dedicated server in pure peer-to-peer networks. Compared to hybrid peer-to-peer networks, the pure peer-to-peer networks are simple and has higher fault tolerance; but hybrid peer-to-peer networks consume lesser network resources and has a high scalability factor compared to pure peer-to-peer networks.

5.3.1.3.3 Chosen Architecture & Justification

Considering the client-server architecture and peer-to-peer architecture and comparing them with the proposed system; the author declares the proposed system fall under the sub-category pure peer-to-peer architecture under the main category peer-to-peer architecture; as the system have standalone nodes which have equal responsibility to do the processing required for packet forwarding during the network activity. Unlike the client-server architecture, the proposed system has no server administering or helping the nodes to connect each other; or to maintain connections nor any servers configured for data storage or network services.

Mobile ad-hoc networks have the ability to share network where the host devices can communicate with each other and share files or internet connection depending on the client requirement. Due to its ability to operate without a wireless enabled router specifically designated for networking; ad-hoc networks are beneficial for spontaneous network creation for temporary internet and file sharing. During file sharing and network sharing, a node connected to an ad-hoc network will act as a node and a host of the network simultaneously. During an internet connection sharing session, the node will reverse the wireless radio receiver and start transmitting as a wireless router; where nodes can connect to it for web related activities. By connecting to a mobile ad-hoc network, the host doesn't need to have a direct connection with the router/ access point. With the increased popularity in mobile computing devices such as PDAs, laptop computers, smart phones and tablets; wireless networking has been an essential feature which create inter-connection communication sessions. With the increased use of wireless networking, the ad-hoc networks has been playing a major role in implementing, maintaining and terminating network connections where the infrastructure networks are unavailable. The ad-hoc networks work similar to infrastructure networks, where the network Quality of Service, security and the routing protocols are the driving forces of ad-hoc networks.

5.4 Factors Required For Efficient Performance of Mobile Ad-Hoc Networks

5.4.1 Throughput

Throughput is the number of bits transmitted by a node to its destination in a given time. According to (Li, et al., 2010) "the time average of the number of bits that can be transmitted by each node to its destination is called the per-node throughput". The average throughput is directly affected by the factors such as node speed, routing protocol, and the network size; while directly affecting the power consumption and node processing. Analyzing the report by (Perkins, et al., 2002), the average throughput in Bytes per second vs design point (number of trials) directly depend on the node speed (node performance), while routing protocols involved and the network size significantly affect the node throughput.

The node has to increase the processing with the increase of throughput; which directly affect the power consumption of the device. The load on a single node depends on the on-going network activity and the number of nodes connected to the network. The distance directly affect the resource utilization of the device as the node has to maintain the on-going network transfers; regardless of the distance to the adjacent node. The number of nodes and redundant path availability reduces throughput per node; but increases the routing update frequency due to the mobility factor of the network.

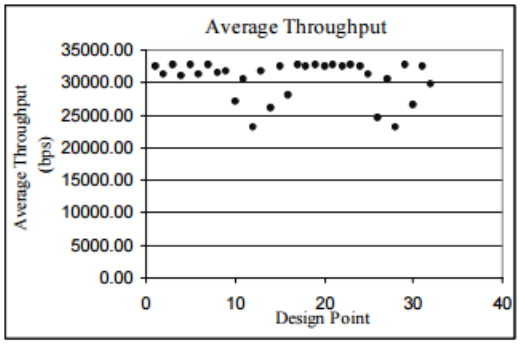

Figure 4 - Average Throughput (Li, et al., 2010)

During a data transfer, nodes transmit data bits between the connected devices. Throughput is the number of information bits processed and transmitted in a given time. The number of bits transmitted by a node to its destination in a given time is the ***per-node throughput***. The ***overall network throughput*** can be measured by analyzing the total number of bits transmitted by the

connected nodes at a given time. (Li, et al., 2010) The throughput of a node vary according to the following factors.

5.4.1.1 Theoretical Maximum Throughput (TMT)

TMT refers to the maximum capacity of the broadcast channel of IEEE802.11 technology used in the network system. TMT can be measured as the MAC layer Service Data Units (MSDU) which can be transmitted at a given unit time. (Jun, et al., 2003), (Lee, 2007), (Bordim, et al., 2010)

5.4.1.2 Peak measured throughput

The throughput capacity of the implemented or simulated network measured using number of bits transmitted in a given time. The peak throughput depends on the hardware compatibility of the node, availability of power and system processing power; which vary according to the configuration of the network. (Gust, 2013)

5.4.1.3 Maximum sustained throughput

Calculated using average throughput of the network monitored over a given time. The network activity is measured for the packet loss, delay/latency for packet delivery and packet routing. The maximum sustained throughput is achieved by calculating the network activity where the network load and throughput values are at an equillibrium (packet loss) and a constant delivery time (latency). (Bolding & Snyder, 1994)

The node throughput is measured by monitoring the upload and download data rates.Considering the wireless networks performance, hardware compatibility plays a major role. The IEEE 802.11 wireless standards describe the hardware capability of the node. The different standards are used to categorize the capability of the hardware according to the following criteria

- Operation frequency
- Modulation
- Maximum data rate
- Antenna/ harware technologies

5.4.2 Routing Overhead

For the routing to be efficient, the nodes produce control messages which include routing tables, route requests, route replies and error messages. The routing overhead refers to the average number of control packets produced per node. As the nodes act as the routers, the control messages have to be processed at the nodes. The node speed, routing protocols involved in the network and the network size directly involve the routing overhead; while the processing and power consumption gets affected significantly by the above factors. According to (Perkins, et al., 2002) the average packet overhead (per node) vs the design points (trials) is as follows.

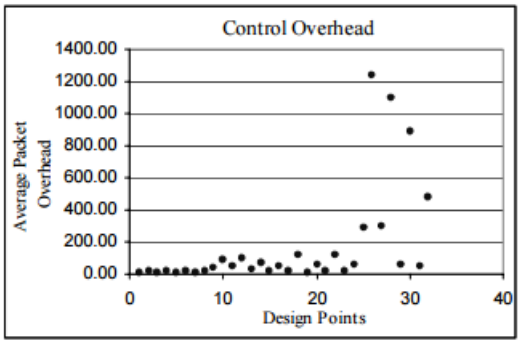

Figure 5 - Control Overhead (Li, et al., 2010)

The packet overhead has a direct interaction with number of nodes and the node speed (node performance); where the nodes with higher performance can handle the packets without much overhead regardless of the network size; but significantly affects the power consumption of the node. With higher node count, the availability of redundant paths reduce the packet overhead per node; but significantly affects the network performance due to high routing update frequency.

A network utilizes a part of its available bandwidth for routing updates. Considering wireless ad-hoc networks, the routing updates are frequent due to the higher mobility of the node and the node activity as a router. The routing updates are sent individually to the nodes within the node's signal range. Individual route maintenance increases the route overhead, thus consuming considerable bandwidth and increasing the throughput per node.

Routing overhead depends on the routing protocols used in the network. The routing protocols can be categorized under the following categories.

5.4.2.1 Proactive routing

Send periodic routing updates to all nodes connected to the network resulting in a routing traffic overhead, but delay is minimized as the nodes have routes to each node connected. OSLR (Optimized Link State Routing) protocol is a table-driven proactive routing protocol optimized to reduce the routing overhead.

5.4.2.1.1 Distance vector routing

Maintains routing tables which contain the direction and the distance to a node connected in the network. The direction is measured by the cost to reach a certain node. The route with the least cost is notified as the route with minimum cost; where direction is calculated by the next hop address and the exit interface. AODV (Ad-hoc On-demand Distance Vector) protocol and DSDV (Destination Sequence Distance Vector) protocol are routing protocols utilizing proactive routing.

The distance vector routing under proactive routing protocols increases the utilization of the network by creating high network overhead traffic. This method is efficient for small networks which minimizes the delay. Due to the high overhead, the network scalability factor is less; but has minimal effect on small scale network with device count less than 10.

5.4.2.2 Reactive routing

The route to an unknown node is determined by broadcasting routing updates to all nodes leading to high latency for a packet to determine its route and excessive flooding affects the network adversely; utilizes node resources excessively leading to high processing and power consumption of the node.

5.4.2.2.1 Flooding

Every incoming packet is broadcasted to every node in the network to determine the best path for the packet delivery. OSPF (Open Shortest Path First) protocol and DVMRP (Distance Vector Multicast Routing Protocol) are protocols utilizing Reactive Routing mechanism.

OSPF and DVMRP routing protocols utilizes the reactive routing mechanism; where the protocols flood the whole network with routing update packets, which will significantly affect network activity. Due to the frequent routing update flooding process, the network scalability factor is less and results in a very high delay and jitter. However, the effects of using this routing protocol is minimal on small networks with a less device number and a lesser network activity.

5.4.2.3 Hybrid Routing

Hybrid routing protocols combine the advantages of reactive and proactive routing protocols. The hybrid routing advantages depend on the number of active nodes in the network (higher the nodes, lesser the efficiency) and response to network traffic depend on the traffic overhead of the node. (Higher the traffic, higher the overhead). Hybrid routing involves ZHLS (Zone-based Hierarchical Link State Routing Protocol) ZRP (Zone Routing Protocol) which uses IARP (Intrazone Routing Protocol) from proactive routing and IERP (Interzone Routing Protocol) reactive routing. The efficiency of hybrid routing depends on the number of connected hosts and network traffic overhead, where the increase in the number of nodes and network traffic can significantly affect the network efficiency.

5.4.3 Power Consumption

The size of the network, number of connected nodes, node processing and average routing overhead per node directly affects the power consumption of the node; where power is a limiting factor for mobile devices. Described by (Perkins, et al., 2002), the number of nodes and the network load (throughput) directly affects the power consumption; where a constant maintenance of network traffic is essential.

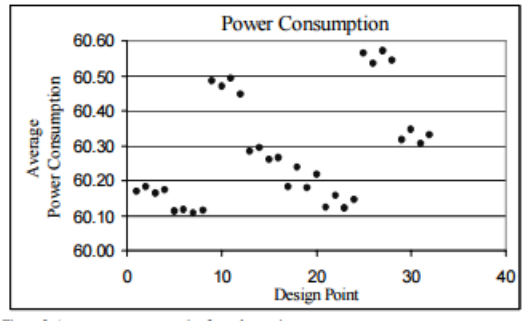

Figure 6 - Power Consumption (Li, et al., 2010)

However, the number of nodes reduces the network load as there are multiple redundant paths for the destination; but affects node performance and power consumption due to constant routing updates, routing requests and responses and node processing.

A main factor affecting the performance of an ad-hoc network is the power consumption. The node power is a critical factor affecting network behavior, network performance, lifetime, scalability and network coverage which solely depend on the power availability of the node. Power consumption factor drastically affects the most important network metrics such as throughput, delay and jitter. Due to the mobility of the wireless networks, the power consumed to maintain the connection vary overtime. According to (ElBatt, et al., 2000), the distance between nodes affect the power consumption of the node significantly. With the distance the signal strength drops and the device has to spend more energy to maintain the network connectivity. Other than the node distance, the following factors affect the power consumption of the node.

- Node throughput – refers to the overall packet handling of a node; including the incoming and outgoing network packets and routing update packets
- Network size – the scalability of the network affects the power consumption of nodes as the number of routing updates increase with the number of hosts connected.
- Number of active sources – the power consumption of a node depends on the activity and the mobility of the source.
- Routing protocol used – as some routing protocols require higher processing power, the node power consumption gets significantly affected depending on the routing protocol used. (Perkins, et al., 2002), (Gupta, et al., 2010), (Bakalis, et al., 2013)

5.4.4 Signal Strength

The wireless network activity involves in maintaining connections between devices which use radio signals to transmit and receive data, where the network connection strength is measured by the radio signal strength maintained by the device's wireless adapter for network activity. The signal strength is measured in decibels (dB) and displayed to the user as a percentage of radio signal emitted.

The signal strength can be calculated utilizing Frei's transmission power equation considering the Transmission power of the sender, remaining power of the wave at receiver, wave length and distance between the sender and the receiver.

$$P_{RX} = P_{TX}\left(\frac{\lambda}{4\pi d}\right)^2$$

P_{TX} = Transmission power of sender
P_{RX} = Remaining power of wave at receiver
λ = Wave length
d = Distance between sender and receiver

Figure 7 Signal Strength Calculation Formulae (Lee, 2007)

5.4.5 Protocol Statistics

Under the protocol statistics category, the user intends displaying the packet types transmitted over the network with the number of packets transmitted by each packet type. The author intends using the TrafficStats API to monitor the number of packets received and transmitted by the network, but faces the following restrictions and the limitations.

- The TCP and UDP packet bytes cannot be indicated separately as the transport layer statistics are unavailable from JELLY_BEAN_MR2 (android version 4.3) onwards.
- The total network packet count (TCP/UDP) and the wireless packet count can be calculated by subtracting the mobile packets from the total packet count; but the TCP and UDP packets cannot be indicated separately. (Developer.android.com, 2016)

5.4 SNMP (Simple Network Management) protocol

SNMP is a protocol used for network management; for collecting information to configuring network devices running on IP (Internet Protocol) networks. With the aid of SNMP protocol, the LAN network can be configured and managed from a management host.

Using SNMP protocol, the management node is given the privilege to monitor network performance, audit network usage, and detect network faults, configure remote devices.

SNMP protocol consist of 3 types as

5.4.1 SNMPv1

The general message format includes the following fields.

1. Version number – Defined in Integer syntax of byte size 4. Ensures compatibility between different SNMP versions; denoted by value 0.
2. Community – defined in Octet string syntax with variable byte size. A simple SNMP community-based security mechanism implementation.
3. PDU – the message body communicated as Protocol Data Unit; declared in variable byte size. (Cozirok, 2005), (Tcpipguide.com, 2005) (Cozirok, 2005)

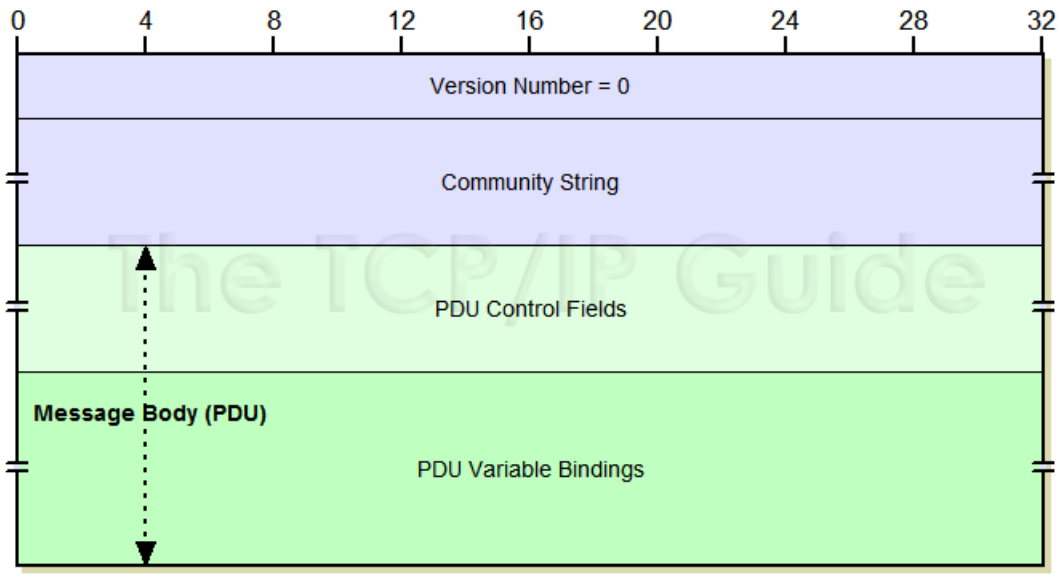

Figure 8 - SNMP message body *(Technet.microsoft.com, 2016)*

5.4.2 SNMPv2

SNMPv2 packet was introduced to overcome the issues faced during usage of SNMPv1. Compared to SNMPv1, SNMPv2 includes the following data in the common PDU format.

PDU type value	PDU type
0	GetRequest-PDU
1	GetNextRequest-PDU
2	Response-PDU
3	SetRequest-PDU
4	Unused (Trap-PDU in SNMPv1)
5	GetBulkRequest-PDU
6	InformRequest-PDU
7	Trapv2-PDU
8	Report-PDU

Table 2 SNMP PDU types and values

1. Request Identifier – integer syntax of byte size 4. A number generated to match requests with replies; generated by a device which send the request and is copied to the Response-PDU field by the replying device.
2. Error status – integer (enumerated) syntax of byte size 4. An integer used in the Response-PDU to inform the result of the request to the SNMP requesting entity.

Error Status value	Error code	Description
0	noError	No error occurred. This code is also used in all request PDUs if there are no error status to report.
1	tooBig	The Response-PDU size is too large to transport.
2	noSuchName	The name of a requested object was not found.
3	badValue	A value in the request doesn't match the structure for the requested object by the host. Eg: object in the request specified with an incorrect length/type
4	readOnly	An attempt to set a variable that has an Access value indicating that it is read-only.
5	genErr	An error occurred other than the defined errors
6	noAccess	Access was denied to the object for security reasons.
7	WrongType	The object variable binding which is incorrect for the object
8	WrongLength	A variable length incorrect for the object.
9	WrongEncoding	A variable declared where encoding is incorrect for the object
10	WrongValue	The value given in a variable binding is not possible for the object.
11	noCreation	A specified variable doesn't exist and can't be created
12	inconsistentValue	A variable binding specifies a value that could be held by the variable but cannot be assigned to it at this time.
13	resourceUnavailable	An attempt to set a variable required a resource that is unavailable

14	commitFailed	An attempt to set a particular variable failed.
15	undoFailed	An attempt to set a particular variable as part of a group of variables failed, and the attempt to then undo the setting of other variables was not successful
16	authorizationError	A problem occurred in authorization.
17	notWritable	The variable cannot be written or created
18	inconsistentName	The name in a variable binding specifies a variable that does not exist.

Table 3 - Error code description of SNMPv2

3. Error index – enumerated type integer with syntax of byte size 4. Contains a pointer specifying which object generated the error. The Error Index is always zero in a request message.
4. Variable bindings – the packet header contains a variable syntax with variable byte sizes. A set of name-pairs identifying the MIB (Management Information Base) objects in the PDU and contain values if the message type is not a request message. (Tcpipguide.com, 2005), (Cozirok, 2005)

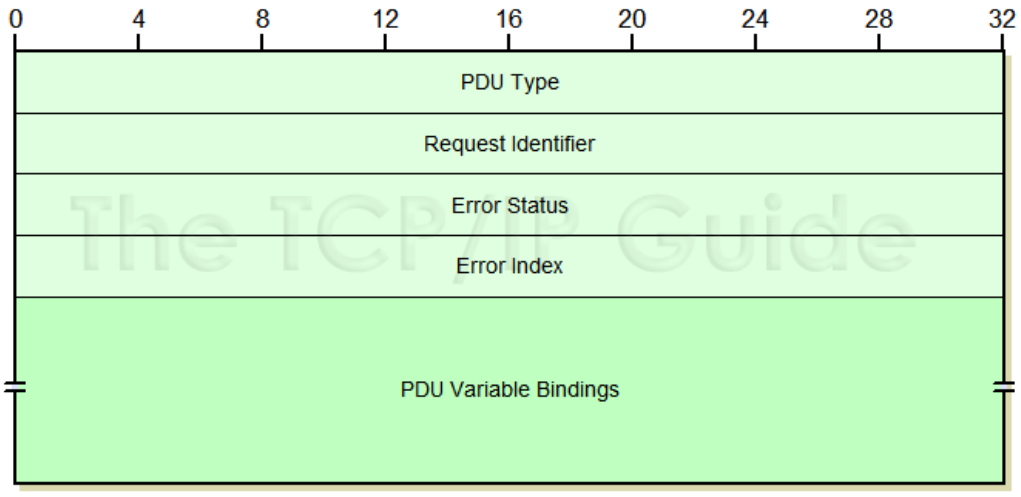

Figure 9 - SNMPvv2 Packet format (Tcpipguide.com, 2005)

5.4.3 SNMPv3

Compared to SMTPv1 and SMTPv2 packets, SMTPv3 packets have the following differences in the packet.

1. Message Maximum Size – the maximum size of the message the sender can receive. Ranging from 484 through $2^3 – 1$.
2. Message flags – consist of an octet string syntax with 1 byte size.
3. Message Security Model – integer syntax of byte size 4; indicates the security model used for the message encryption. For SNMPv3, the value is 3.
4. Scoped PDU – contains parameters that identify SNMP context, describing a set of management information accessible by a particular entity. (Tcpipguide.com, 2005), (Docwiki.cisco.com, 2012)

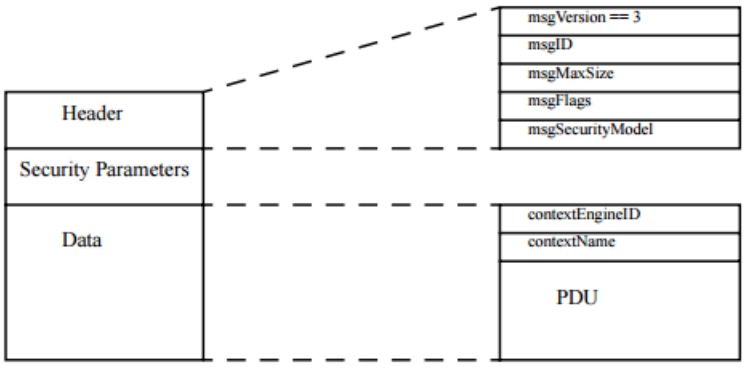

Figure 10 - SNMPv3 decoded packet details (Docwiki.cisco.com, 2012)

Subfield name	Size in bytes	Description
Reserved	5 bits	Reserved for future use
Reportable flag	1 bit	The receiver must send back a Report-PDU to the sender whenever the value is set to 1

Privacy flag	1 bit	Indicates the receiver that an encryption is used to protect the messages. The authentication flag must be set to 1 in order set the Privacy flag to 1
Authentication flag	1 bit	Indicates the receiver an authentication method is used to protect the authenticity of the message received.

Table 4 SNMPv3 header details

5.5 SNMP components

5.5.1 SNMP Managers

An SNMP management is responsible for the communications with the "SNMP agent" configured devices. The system includes a computer running network services and management software. The SNMP management handles IP-related queries and information with SNMP agents other than system configurations. The key functionalities of SNMP manager are

- Query agents and get responses
- Set variables in agents
- Acknowledge asynchronous events

(OpManager, 2016), (Technet.microsoft.com, 2016), (Ellingwood, 2014)

5.5.2 SNMP Agents

The SNMP agents are configured programs which enable the SNMP managers to collect management information and required databases from the device locally. The agent has the ability to monitor network devices and respond to query requests from SNMP managers and has the ability to be configured in a way such; if an illegal access or system reboot occur, a message sent to the management. (Technet.microsoft.com, 2016), (Technet.microsoft.com, 2016), (Ellingwood, 2014)

5.5.3 Management Information Base (MIB)

The database maintained by SNMP agent to record the information of parameters managed by the SNMP manager; which is accessed by NMS (Network Management System). The commonly shared database is known as the MIB (Management Information Base).

The MIB contains

- Statistical and control values defined for nodes of a network
- Object Identifier (OID) – unique identifier denoting specific characteristics of a managed device and a text description of the object
- The object's data type definition, index for objects assigned with complex data types.
- Size restrictions, range information, permission level of access to the object allowed (read/write permissions). (OpManager, 2016), (Technet.microsoft.com, 2016), (Ellingwood, 2014)

5.5.4 SNMP message transport

SNMP messages are transported using the UDP (User Datagram Protocol) running on IP packet type. By default, the UDP port numbers 161 and 162 are used for this. The port 161 is utilized for general SNMP messages, which is utilized by SNMP agents and SNMP managers listen to SNMP requests and replies. The port 162 is utilized for SNMP trap messages; which indicates the SNMP managers the activity of the connected clients via SNMP agents. (Cozirok, 2005), (Technet.microsoft.com, 2016), (Hofman, 2009)

5.6 User Authentication

During the process of adding a new node to a network, an authentication procedure is followed so that the network is only utilized by the privileged users. Authentication procedures follow multiple techniques which include the following encryption algorithms used as a standard practice at present.

5.6.1 WEP (Wired Equivalent Policy)

An open network provides higher network performance as there is no encryption algorithm involved; but the transferred packets are prone to a malicious attack. To overcome the security issues in an open network, the WEP (Wired Equivalent Policy) have been introduced which offers the same level of security as a wired network. WEP uses RC4 stream cipher as the encryption algorithm. The functionality of the WEP encryption algorithm is as follows.

The input data is being encrypted while another process protects the data from unauthorized modifications. The security key produced consist of a 40-bit secret key connected to a 24-bit Initialization Vector (IV); resulting in a final key of 64-bit length or a secret key of length 122-bit connected to a 24-bit IV; resulting in a 128-bit length final key.. The produced key is used as the input for the Pseudo-Random Number Generator (RC4 algorithm) which encrypts the packet; except the packet header including the Initial Vector (IV) which is used to encrypt the packet body.

The RC4 algorithm has an unencrypted header; which is considered as the major weakness in the system. The header is not encrypted as it contains the routing information and the Initial Vector (IV) used for the key generation process as it is required at the decryption process. Other than the unencrypted header and the IV, the WEP doesn't have a specific key management process, 24-bit long Initial Vector and a 4-byte Integrity Check Value (ICV) added at the end of the encryption which is computed using the original packet.

But the RC4 algorithm utilizes minimal system resources as the encryption process utilizes a 64-bit key and the Initial Vector (IV) of size 24-bits which is attached to the header without encrypting. This is quite helpful at the receiver's end for decryption. The header of the WEP packet is unencrypted; where the nodes have no header encryption and decryption processes involved which minimizes the resource utilization of the nodes during the data transfer. (Chang, 2016), (Gralla, 2005), (Johns, 2015)

5.6.2 WPA (Wi-Fi Protected Access)

WPA security encryption integrates IEEE 802.1X standard with TKIP protocol which creates a higher security compared to WEP encryption. The IEEE 802.1X standard enhances the security of wireless LAN networks and provides authentication framework following IEEE 802.11 standard.

IEEE 802.1X uses EAP (Extensible Authentication Protocol) for packet encryption during authentication process. TKIP (Temporal Key Integrity Protocol) as described by has the following enhancements to provide a higher security.

1. Message Integrity Code (MIC) – cryptographic message integrity code designed specifically to detect altered or modified packets.
2. Dynamic initialization vector (DIV) – protects the packet against man-in-the middle "packet alteration" attack.
3. Key scrambling and fragmentation – a packet specific key is produced by the TKIP protocol by combining the temporal key with the packet sequence counter.
4. Extended Initialization Vector (IV) size to 48-bits. (Greenfield, 2003), (Singh, et al., 2014)

5.6.3 WPA2 (Wireless Protected Access 2)

WPA2 has replaced the RC4 encryption algorithm with AES (Advanced Encryption Standard) algorithm with a 48-bit IV (Initial Vector). Instead of TKIP (Temporal Key Integrity Protocol), the encryption uses CCMP (Cipher Block Chaining Message Authentication Code Protocol) which minimizes man-in-the –middle and replay attacks. (Murphy, 2015).

AES encryption algorithm utilizes block ciphers based on Rjindael algorithm; which can provide encryption at 128, 192 and 256 bits. (Chou & Kang, 2010). WPA2 encryption are of two types as

WPA2 Personal – Pre-Shared Key of 256-bit authentication which is designed for home and small networks.

WPA2 Enterprise – IEEE 802.1X encryption offering enterprise grade authentication which requires a RADIUS authentication server to provide automatic key generation and authentication. (Posey, et al., 2008), (Coleman & Westcott, 2009)

Analyzing the above encryption algorithms, the author has identified they are vulnerable and a network attacker can decrypt the encryption algorithm and listen to network traffic. Considering the security, the author has planned implementing the authentication using the one-time pad concept; where the passwords used for authentication is used only once and are discarded

afterwards. Also known as the unbreakable encrypting method, one-time pads consist of the following steps.

1. At the sender's end, the message encryption takes place with a substitution cipher, which is known only by the sender and the receiver.
2. The offset/key varies from encrypted message to message, and a unique offset/key is used for every message transmitted and received.
3. The substitution cipher pad is discarded after encrypting the message at the senders end and the pad at the receivers end is discarded after the decrypting the encrypted message.

(Schemeh, 2001), (Buchmann, 2000)

The author intends using the one-time pad concept for the authentication of the selected system. The following method is proposed by the author for implementation of the user authentication.

- The new node sends a connection request to the existing network, where the network administrator gets the connectivity request.
- The network administrator has to approve the connection request and approval is sent from the network to the new node.
- The acknowledgement is sent to the network administrator from the new node; indicating the connection status.
- The author intends developing the application in a user-friendly manner, where the network administrator has to tap and hold a button to authenticate the new node; where the authentication process takes place in an automated manner.

CHAPTER 6 – SOLUTION CONCEPT

After the problem identification and a similar system study on current available solutions, a requirement analysis is carried out to gather information on existing solutions prior to implementation. As the data gathering method, **listening and interviewing** procedure is followed as the users will describe their real-life experience on interaction with the existing solutions. This method will provide more details on the user experience and the existing system drawbacks rather than **observations** methodology which involve observation of users during their involvement with the existing systems. The **observations** methodology is skipped as the interviewer has to spend more time observing users and the interviews are carried out in a limited timeframe.

The proposed application involves two user interviews, first set of interviews carried out at data gathering stage, where the users are subjected to **open-ended questions** which cover the user opinion and experience on existing systems. Following this methodology help gather more information on existing systems; thus helping in producing an application which will meet the user expectations and requirements. The second set of interviews involve **Closed-ended questions** include true or false selection, multiple choice questions focused to obtain a critical evaluation of the application and the criteria met.

Other than the above mentioned, **Interviews with groups** methodology is carried out on online gamers who share the ad-hoc network to implement a network for the connectivity through mobile ad-hoc networks. Following this methodology help gather more information on required criteria as group information contain more details than the individual surveys.

The application developed has a simple, basic **GUI (Graphical User Interface)** design providing user easy access to monitoring components. Only the main activity is run at the application initiation and other activities are enabled on user request. This methodology is followed to minimize system resource utilization by reducing number of concurrent threads running. On the other hand, launching the activity on user request will generate the values and deliver the most accurate readings at the given instance; thus reducing the necessity to utilize runnables to update the activities frequently. Following the above method will minimize the number of threads and reduce processor load; significantly improving the power usage of the device.

The APIs used are developer provided default Android APIs and modified APIs aren't used to meet the required criteria as the implemented system is developed to support devices running on factory-default OS. The above decision is made as the rooted device count is significantly less than the devices running on factory-default OS and the standard APIs support both rooted and non-rooted devices, the development is carried out using standard android APIs.

The implemented application is subjected to white-box and black-box testing. The application is first tested for its functionality by unit testing, followed by functionality testing. Multiple units together form a component of an application. The functionality of the component depend on the accuracy of the units as they collectively display the output of the component.

The network details were displayed on a separate activity during the early stages of development, but with the application optimization process, the activities are initiated on user request displaying the readings for the current session and terminated when the user leaves the activity; resetting the recorded readings to zero. Therefore the network statistic details are displayed in the main activity to obtain accurate readings. Considering upload and download monitoring, the activities have to be started during the startup of the hotspot as a node or during application launch as a host. Depending on the device operating mode as a host or a node, the network statistics are initiated at the application launch. The most crucial sensor readings including battery temperature, voltage, plugged in source and charge/discharge status are displayed alongside the network statistics in the main activity.

CHAPTER 7 – SOFTWARE ARCHITECTURE & DETAILED DESIGN

7.1 Use-case Diagrams

The Use Case diagram model provided displays the user interaction with the system regardless of its mode of operation. The user interaction with all the components are possible during the operation as a node or as a host.

As the Use Case diagram explains, the user directly interacts with all the features of the provided application; most of the functionality and features are directly accessible by the user by a button click.

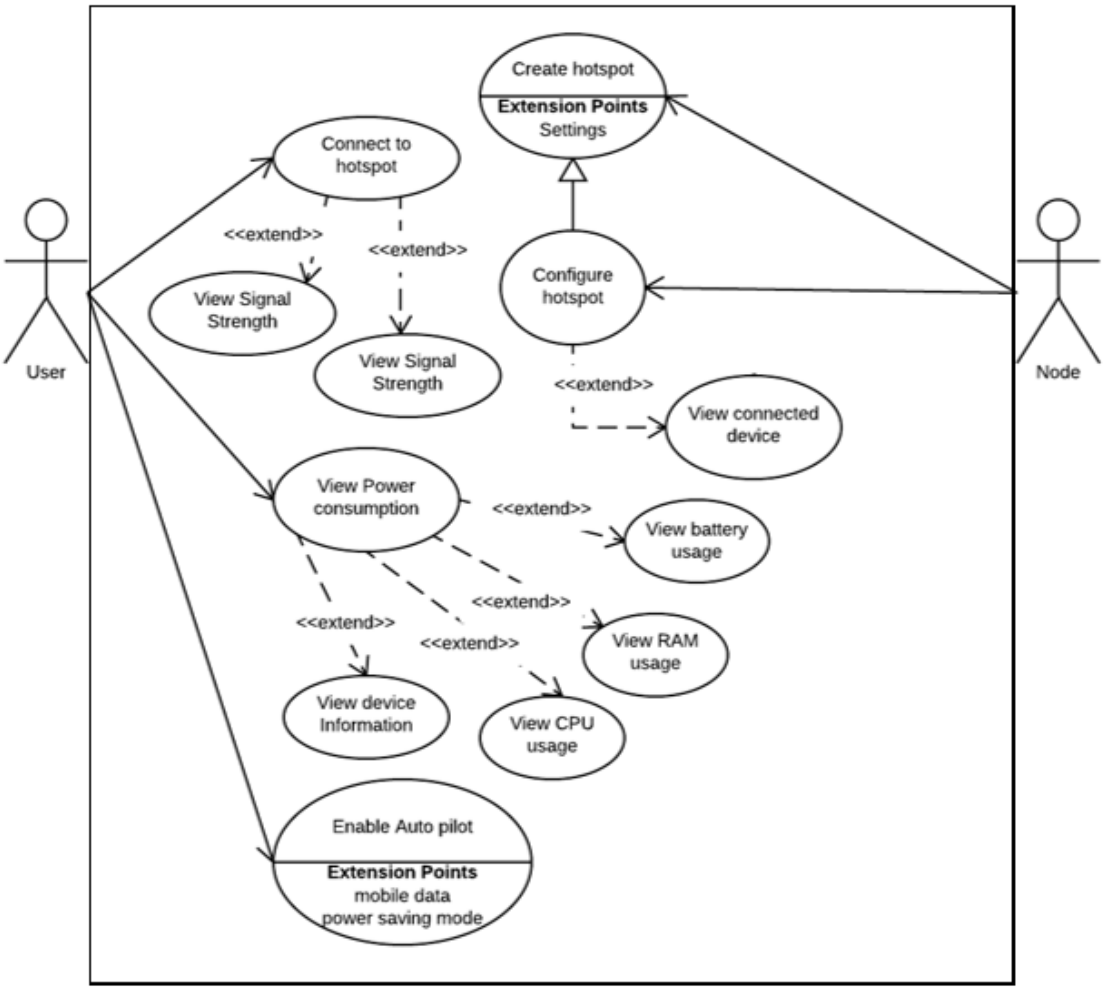

Figure 11- Use- Case diagram

7.2 Activity Diagram

7.2.1 Main Activity

The activity pane describes the flow of the **Main Menu Activity** and reflects a high level flow diagram of the component.

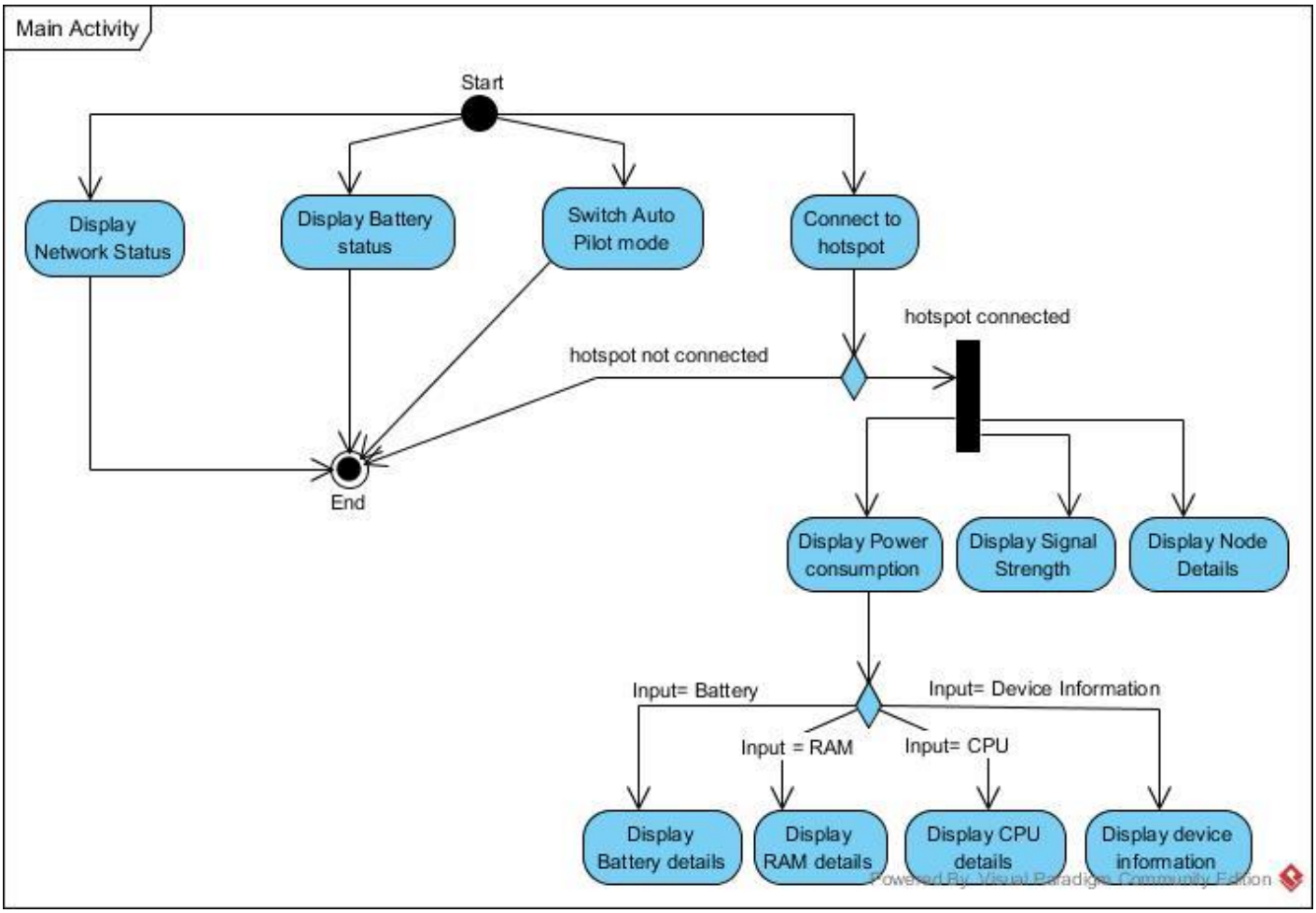

Figure 12 - Main Activity - Activity diagram

7.2.2 Network Activity

The activity pane describes the flow of the **Network Activity** and reflects a high level flow diagram of the component.

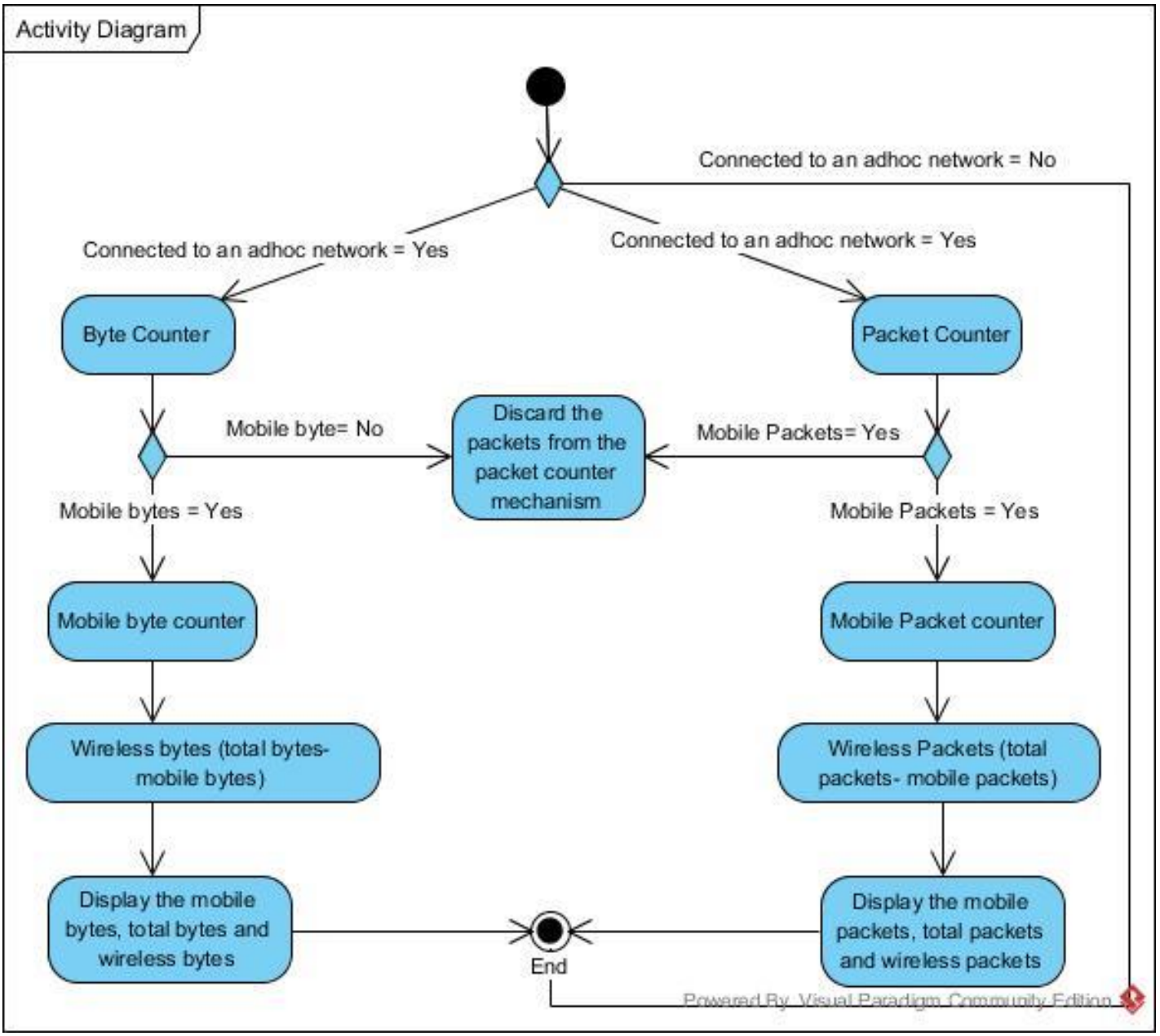

Figure 13 - Network Activity - Activity Diagram

7.2.3 Power Consumption Activity

The activity pane describes the flow of the **Power Consumption Activity** and reflects a high level flow diagram of the component.

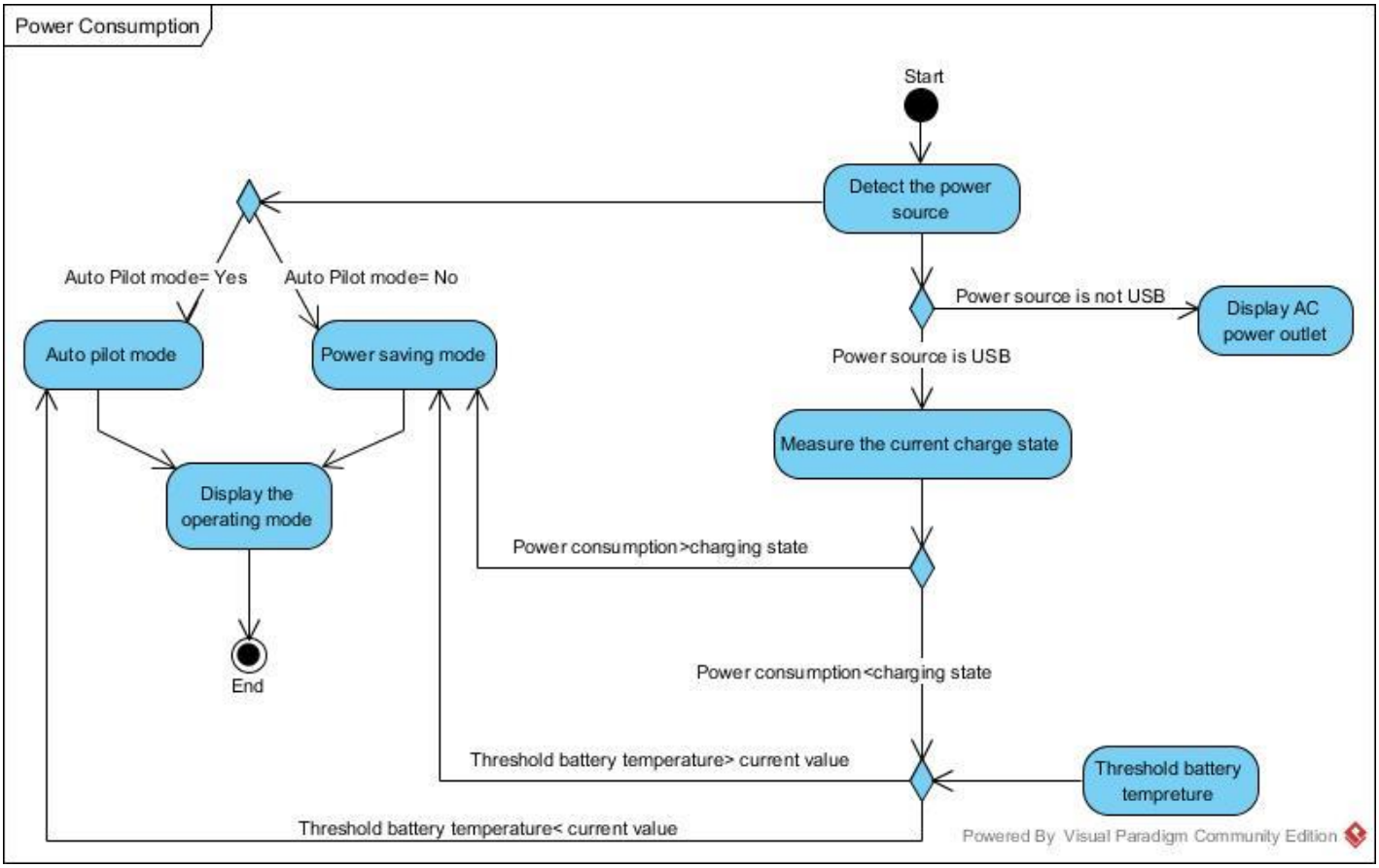

Figure 14 - Power Consumption Activity - Activity Diagram

7.3 Class Diagram

Class diagram represents all the possible classes that will be exploited during the implementation and development of the application. A JPEG image of the class diagram is attached in the CD.

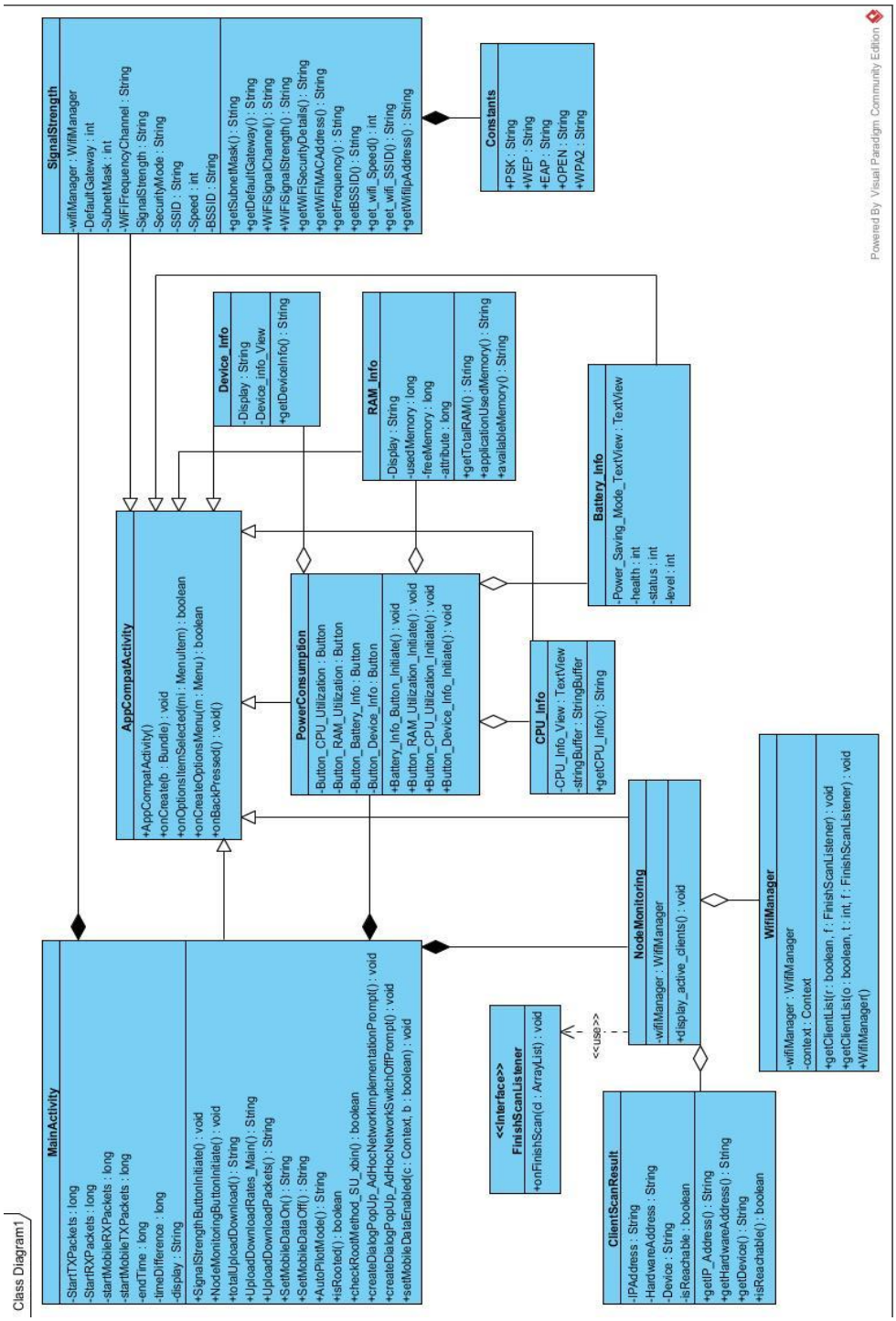

Figure 15 – Class Diagram

7.4 Detailed Design of the Application

The application is designed with a simple GUI structure providing the user with easy access to monitoring components. A main activity is implemented to run at the application startup where thee user get the functionality to implement the hotspot, monitor the network activity and battery readings, navigation buttons set up to enable the user to navigate to different available activities.

The other activities in the application are launched by pressing respective clickable buttons. Until user involvement, the methods and activities remain inactive and gets activated only if the user wish to interact with the respective applications. The activities close on back press and always return to the main activity; except in the power consumption section.

The activities and functions declared within the **Main Menu activity** are described below.

Hotspot on/off switch – the switch is set to **disabled** state (false) as the default. When the switch is enabled, the user is directed to the hotspot setup menu at the settings tab. The user is notified the redirection by a pop-up message. After successful ad-hoc network implementation, the user is notified by a toast displaying **hotspot started**. The switch enters to **enable** state if the ad-hoc network implementation is successful and the device will start acting as a node device of the network.

After successful network activity, the user decision to disable the ad-hoc network can be done by changing the switch from enabled state to disabled state and the user will be redirected to the hotspot setup menu at the settings tab.
Network status field is displayed by checking the ad-hoc network status and the variable value is returned as a text.

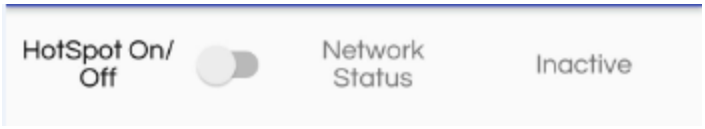

Figure 17 - "HotSpot disabled" notification

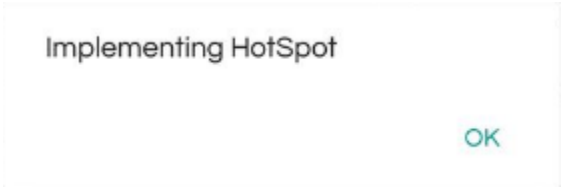

Figure 16 - "HotSpot implementing phase"

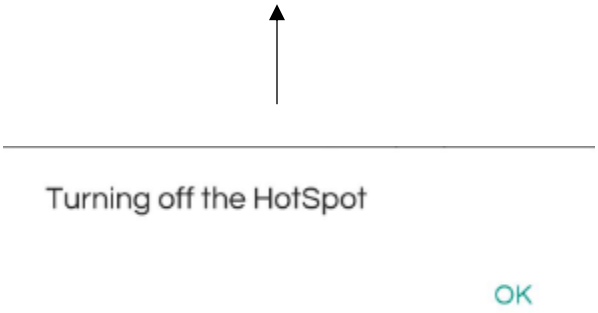

Figure 18 - HotSpot disabling phase

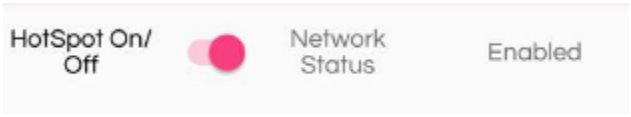

Figure 19 - "HotSpot Enabled" state

Battery status field include details of the charging/discharging status and plugged in source details while battery voltage includes the remaining charge of the battery displayed in millivolts (mV) and current battery temperature displayed in degrees Celsius.

Battery plugged into AC source and device is charging

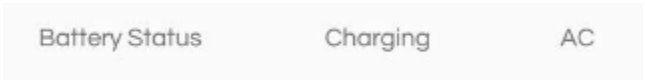

Figure 20 - Plugged-in to AC source and Battery Charging

Battery plugged into USB source and device is charging

Figure 21 - Plugged-in to USB source and Battery Charging

Battery plugged into AC source and device is charging

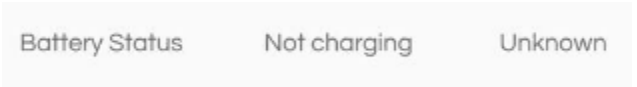

Figure 22 - Device disconnected from the power source

Device activity field displays the current device operating status and depending on the battery temperature sensor reading and upload/download rate monitor, the user will be displayed the operating mode of the device; out of the activity modes **low**, **high** and **inactive**.

Low device activity notification

Figure 23 -" Low device activity" notification

The upload and download rates, total uploaded and downloaded bytes and number of packets received and transmitted are displayed real-time and each method runs on separate runnables to minimize the risk of the application crashing.

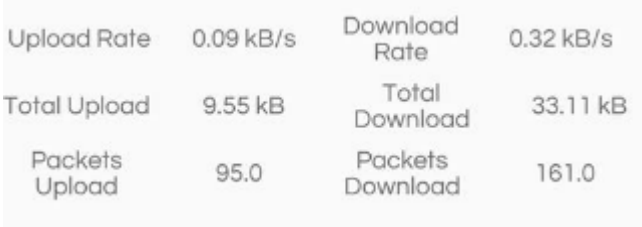

Figure 24 - Network Activity display

Auto-pilot mode on/off switch – the auto-pilot mode is **disabled** (set to false) by default. On enabling, the root status of the device is checked. If the device is rooted, the method will check for a series of conditions and compare the declared variables with battery readings. If device readings exceed the set threshold values, the application is enables and disables the mobile data connection accordingly. If the device is not rooted, the user will be prompted to turn off the mobile data connection through pop-ups and toast messages. The auto-pilot mode will check for the power saving mode status and prompt the user to enable/disable accordingly, regardless of thee rooted status of the device. However, this feature is only available for devices running on android API level 21 upwards. The devices running android API 20 and below will not support the power saving notification feature.

Auto-pilot mode display of a device running on stock android OS

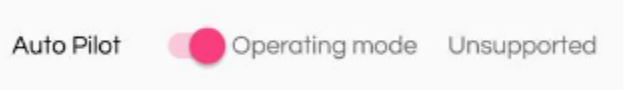

Figure 25 - "Auto-Pilot mode" display of a non-rooted device

Auto-pilot mode display of a rooted device

Figure 26 - "Auto-Pilot mode" display of a rooted device

The user is notified if the device is not rooted when enabling the Auto-Pilot mode

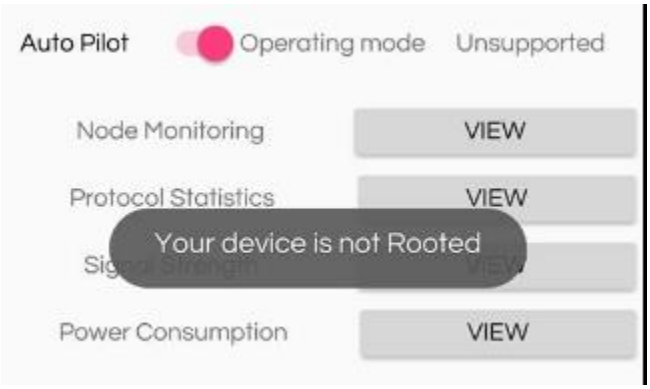

Figure 27 - Root status check when enabling the Auto-Pilot mode

The activity **node monitoring** display the details of the connected hosts including the device hardware and IP addresses, reachability status and hardware type. Depending on the current operating mode, the displayed details differ. If the device acts as a host, only the network details of the connected node are displayed. If the device acts as a node, details of all connected hosts and nodes are displayed.

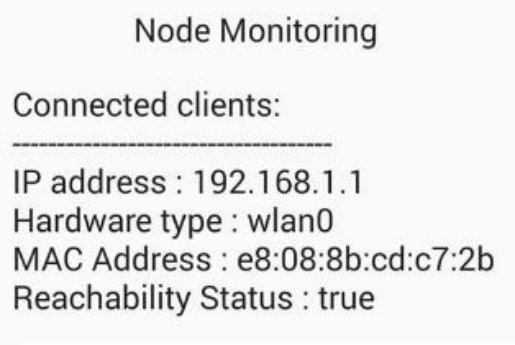

Figure 28 - Connected node detail display during device operation as a host

Figure 29 - Connected client details display during device operation as a node

On opening the **Signal strength** activity, wireless and mobile network details are displayed to the user. The wireless network details include SSID, BSSID, device hardware address, wireless

network speed and signal strength, encryption type, operating frequency and channel, device assigned IP address, network mask and default gateway of the connected network.

Power consumption activity houses the details of the device battery sensor readings, RAM utilization, CPU details and information of the device. Individual activities are created for the criteria and launches separately on button click. The decision to launch separate activities on button click is to increase the efficiency of the program and reduce the number of running threads and processes.

Figure 31 - Signal Strength activity details display on a device with android version 4.4.2

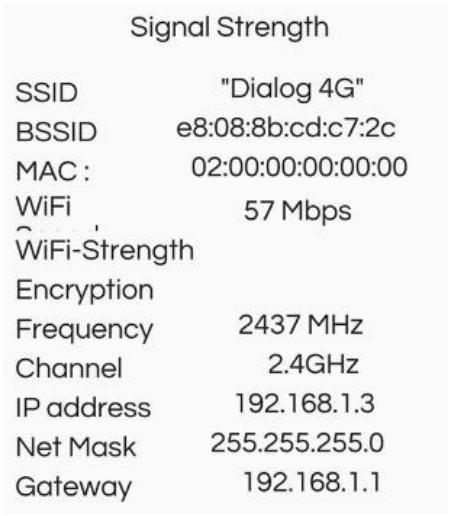

Figure 30 - Signal Strength activity details display on a device with android version 6.0.1

The activity launched on the button **Battery** displays the charging/discharging status, plugged in source, battery technology, voltage, temperature, battery health and power saving mode status. However, the power saving mode status is only available for devices running on android API 21 and above. For the devices with android API 20 or lower, the system returns with **SDK unsupported** in the textbox given. A progress bar is used to display the remaining charge of the battery and a textbox is used to display the remaining charge in integer value, providing user easy understandability of the details displayed.

Figure 33- Battery Info display on a device running on android 4.4.2

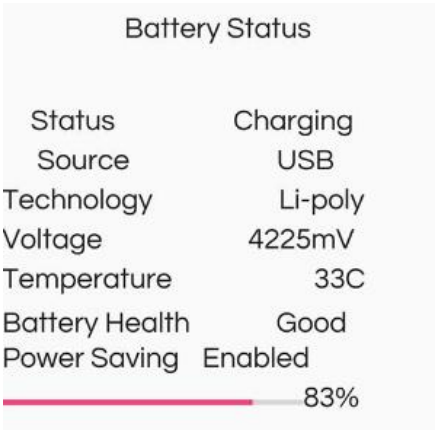

Figure 32- Battery Info display on a device running on android 6.0.1

The CPU information activity will display the processor details of the device; including the processor type, number of cores in the processor, features, hardware modules used and their serial numbers are displayed. Depending on the device type, the displayed details can vary. Therefore the values are displayed in a Scrollable, allowing the user to view all the details available. The CPU information are displayed by reading **/proc/cpuinfo** file in the android system. This procedure enables the application to read the device details within the file and reduces the number of methods; thus reducing the system load and increasing the efficiency of the program.

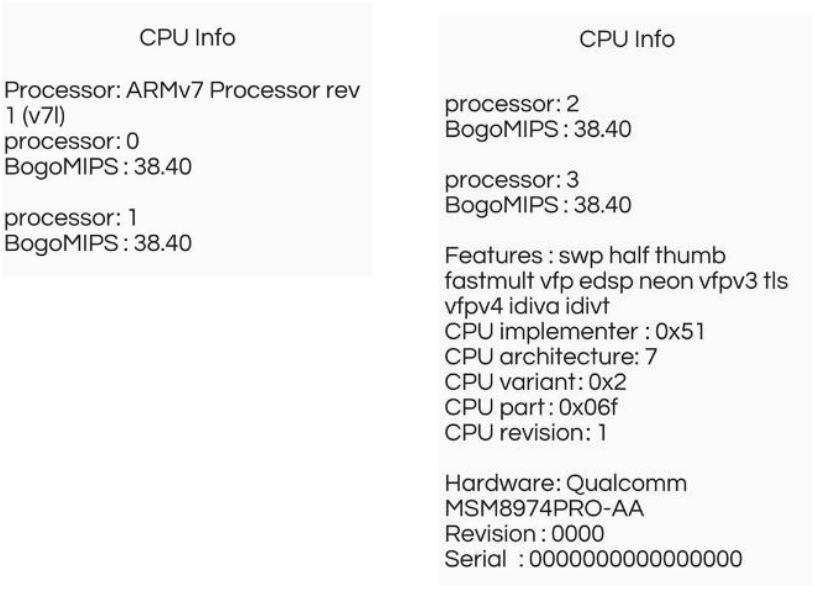

Figure 34- CPU info details display

The activity **RAM Manager** will display the total available memory of the device, memory used by the applications and free memory to be utilized. Monitoring the RAM is crucial as the running applications start to lag and freeze, ending up crashing the operating system.

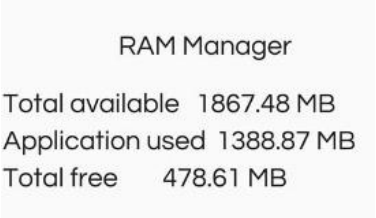

Figure 35 -- RAM info display

The **device information** activity displays the current running operating system version and release, API level, build ID, device type, device model and brand, manufacturer, hardware serial number of the device to the user. With the aid of the above given details, the user has an idea about the deprecated methods and the application support against the device.

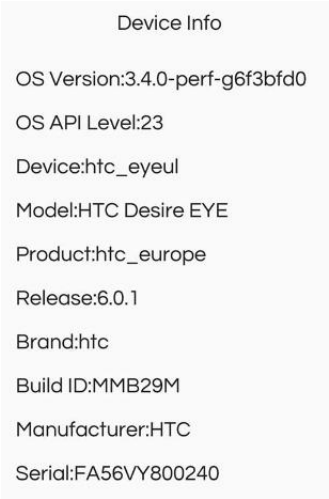

Figure 36- Device info display

CHAPTER 8 – IMPLEMENTATION

The implementation of the proposed tool is discussed in this section which includes design, methods used and APIs used to achieve various criteria described in functionalities.

8.1 APIs used and API levels supported

Android platform utilizes framework APIs to interact with the Android system consisting of core packages and classes, XML elements and attributes to declare manifest file and access resources, intents and permission requests and permission enforcements included within the system. API levels are integer values used uniquely to identify the framework API revisions offered by platform versions and one platform supports exactly one API level (Android 6.0 Marshmallow – API 23).

APIs define specific functionality of an android system and certain APIs require a minimum API level to operate and certain APIs and their functions are deprecated with the upcoming API levels. Application designed and implemented on android platform require the declaration of following API levels in the android Manifest file.

```
android {
    compileSdkVersion 23
    buildToolsVersion "23.0.3"

    defaultConfig {
        applicationId "com.network.ad_hoc.fidelity_fi"
        minSdkVersion 19
        targetSdkVersion 23
        versionCode 1
        versionName "1.0"
    }
```

Figure 37 - APIs used and API levels supported code

- **targetSdkVersion**: declares the API level on which the application is designed to run optimally. All the functions in the declared APIs should have the ability to run on the application.
- **minSdkVersion**: declares the minimum API level application supports. The devices running on API levels below the declared have no support; thus limiting the usability and functionality

of the application. The declared minSdkVersion should be less than or equal to the system's currently operating API level.

- **maxSdkVersion**: declares the maximum API level supported by the application. The devices running on API levels above the declared supports the APIs; however, depending on the API deprecation level, the functionality of the application can be limited. The declared maxSdkVersion should be greater than or equal to the system's currently operating API level.

(Developer.android.com, 2016), (Developer.android.com, 2016), (Developer.xamarin.com, 2016)

8.1.1 TrafficStats API

Utilized to display the network components declared in main menu activity. The class provides network traffic statistics including the bytes and packets transmitted and received during network activity over mobile and wireless interfaces. The API requires API level 8 as the minimum supported SDK and provides support up to the latest API level (currently API level 23).

The following methods are utilized to meet the required criteria.

- Calculate the transmitted and received mobile bytes and packets

getMobileRxBytes( ) – displays the number of bytes received across mobile networks.

getMobileTxBytes( ) – displays the number of bytes transmitted across mobile networks.

getMobileRxPackets( ) – displays the number of packets received across mobile networks.

getMobileTxPackets( ) – displays the number of packets transmitted across mobile networks.

- Calculate the total transmitted and received bytes and packets

getTotalRxBytes( ) – displays the total number of bytes received across mobile networks.

getTotalTxBytes( ) – displays the total number of bytes transmitted across mobile networks.

getTotalRxPackets( ) – displays the total number of packets received across mobile networks.

getTotalTxPackets( ) – displays the total number of packets transmitted across mobile networks.

The API provides no functionality to monitor the received and transmitted wireless bytes and packets. To obtain the wireless bytes, the mobile byte count is subtracted from the total byte count;

assuming there are no other bytes exchanged other than wireless bytes and the bytes utilized for network routing updates and broadcasts are neglected.

Initially, the program was designed to capture TCP and UDP packets from the network and display their respective counts to the user. However, the methods **getUidUdpTxBytes( ), getUidUdpRxBytes( ) and getUidTcpRxBytes( ), getUidTcpTxBytes( )** used to calculate transmitted and received UDP and TCP bytes respectively are deprecated from API level 18 (Jellybean version 4.3.1) along with **getUidUdpRxPackets( ), getUidUdpTxPackets( ) and getUidTcpRxPackets( ), getUidTcpRxPackets ( )** used for total transmitted and received UDP and TCP packet count calculations. Since the application supports minimum API 19 and upwards, the implementation of above methods will return "unsupported".

8.1.2 ConnectivityManager API

The API is used to notify queries on the network connectivity state and notifies applications about the network connectivity changes. To obtain an instance from this class by calling **Context.getSystemService (Context.CONNECTIVITY_SERVICE)** to the respective method. The class is responsible for monitoring network connection changes including wireless and mobile networks and send broadcast intents on network connection changes and supports API level 1 through API 23 (latest available API at the time of documentation).

ConnectivityManager API is utilized in the method **setMobileDataEnabled (**) and mobile data connection is enabled/disabled accordingly. Mobile network state can be accessed on both rooted and non-rooted devices, but the mobile network state modification can be done only on rooted devices. The Boolean value false passed from **SetMobileDataOff ( )** will turn off the mobile data while Boolean true value passed from **SetMobileDataOn ( )** will turn on the mobile data connection. (Developer.android.com, 2016)

8.1.3 WifiManager API

WifiManager API class provides functionality to manage all Wi-Fi related aspects. A class instance can be called by **Context.getSystemService (Context.WIFI_SERVICE)** and integrates features as viewing list of configured Wi-Fi networks and display information on currently

connected access points and networks. Introduced at API level 1, the WifiManager supports all APIs up until the latest release at the time of documentation. (API 23). (Developer.android.com, 2016)

The API is used in multiple activities in the program. In the main menu activity, API is used to check the existing state of the ad-hoc network utilizing the instance **android.net.wifi.WIFI_AP_STATE_CHANGED** and checks the hotspot enabled state by calling **WifiManager.WIFI_STATE_ENABLED** which returns an integer value stating the existing state. Depending on the returned value, the network status is displayed to the user. The system service **Context.WIFI_SERVICE** is called in WifiApManager class and is used on host devices to identify the nodes and identify the hosts connected to a node within a network.

The API is extensively used in the Signal Strength activity, by utilizing the methods **getDhcpInfo ( ), getConnectionInfo ( ), getWifiState ( )** and **getScanResults ( )** in the functions and methods declared to retrieve information related to the connected network.

WifiManager.getDhcpInfo ( ) in the methods

- **getSubnetMask ( )** – displays subnet mask of the connected network
- **getDefaultGateway ( )** – displays the default gateway of the connected network

WifiManager.getConnectionInfo ( ) in the methods

- **WiFiSignalChannel ( )** – displays the operating channel of the connected network
- **getWiFiMACAddress ( )** – displays the hardware address of the device
- **getFrequency ( ) -** displays the operating frequency of the network
- **getBSSID ( ) -** displays the hardware address of the connected node
- **get_wifi_Speed ( )** – displays the available transfer speed of the connected network
- **get_wifi_SSID ( ) -** displays the broadcast SSID of the connected network

WifiManager.getScanResults ( ) in the methods

- **WiFiSignalStrength ( )** – measures the signal strength of the existing network by calculating the difference of signal strength at the sender's and receiver's end.
- **getWiFiSecurityDetails ( )** – displays the type of encryption used in the existing network.

8.1.3 WifiInfo API

Declared in API level 1, WifiInfo API class provides details on wireless connections which are currently active or in the configuration process. The integrated methods and functions supports up to the latest API as at the time of documentation. (Developer.android.com, 2016). The WifiInfo API displays the methods **getBSSID ( ), getFrequency ( ), getFrequency ( ), getLinkSpeed ( )** and **getSSID ( )** with the combination of WifiManager API; described above.

8.1.4 BatteryManager API

BatteryManager class calls strings and constants used in the Intent **ACTION_BATTERY_CHANGED** and displays the user methods for querying battery details and display battery charging properties. BatteryManager API was introduced with API level 1 and supports all API levels at the time of documentation (latest API level 23). In the **main menu activity**, the BatteryManager class is used to display the methods for the most crucial components.

- **Voltage** – the method **getIntExtra ( )** under **ACTION_BATTERY_CHANGED** intent is called and the **voltage** value is read from the intent; referring to **EXTRA_VOLTAGE** constant to display the current battery voltage.
- **Temperature** – the method **getIntExtra ( )** is called and the **temperature** variable is read from the intent **BatteryManager.EXTRA_TEMPERATURE** to display the current battery temperature; displayed in degrees Celsius (**ºC**).
- **Status** – **getIntExtra ( )** method is called and the **status** variable is read from the intent **BatteryManager.BATTERY_STATUS_UNKNOWN** and checked for the state by the following methods.
  - If the constant **BatteryManager.BATTERY_STATUS_CHARGING** returns true, the user is notified the device is charging.
  - If the constant **BatteryManager.BATTERY_STATUS_DISCHARGING** returns true, the user is notified the battery is discharging.
  - If the constant **BatteryManager.BATTERY_STATUS_FULL** returns true, the user is notified the device battery is fully charged.

- If the constant **BatteryManager.BATTERY_STATUS_NOT_CHARGING** returns true, the user is notified the battery isn't charging regardless of its plugged source.
- If the existing battery status is not identified by the device, **UNKNOWN** constant is returned.

- **Plugged in source** – **getIntExtra ( )** method is called under the **ACTION_BATTERY_CHANGED** intent and refers to the constant **BatteryManager.EXTRA_PLUGGED** and checks for the state by the given methods.
  - If the constant **BatteryManager.BATTERY_PLUGGED_AC** returns true, the user is notified the device is charging and is plugged into an AC power source.
  - If the **BatteryManager.BATTERY_PLUGGED_US**B constant returns true, the user is notified the device is charging and is plugged into a USB power source.
  - If the **BatteryManager.BATTERY_PLUGGED_WIRELESS** constant returns true, the user is notified the device is charging and is plugged into a wireless charging source.
  - If the return constant value doesn't meet the above, the user is notified that the plugged in power source is unknown.

The battery status method uses the **battery health**, battery technology and display the power saving mode; alongside the remaining battery charge in a progress bar and the remaining charge percentage of the device.

8.2.6 PowerManager API

The **POWER_SERVICE** function is called within the **getSystemService ( )** method and by utilizing the **isPowerSaveMode ( )** method included in the PowerManager API; the power saver mode status is checked. The **isPowerSaveMode ( )** method requires minimum API level 21 to operate and when enabled, the application functionality is reduced conserving the remaining charge. However, the use of the PowerManager API controls the power state of the device and is advised to minimize the use of the API as it significantly affects the battery life of the device.

(Developer.android.com, 2016)

8.2.7 ActivityManager API

The **MemoryInfo ( )** constructor of the ActivityManager API is used to retrieve the memory details of the device. The API supports all API levels till the latest API level 23 (at the time of documentation) and provides functionality to retrieve information on the total available memory, application used memory and available memory for application activities.

The following methods and constructors are used in the application to detect memory information.

- **availMem -** displays the available memory for new activities. The figures displayed aren't absolute values as a significant amount of available memory is utilized by the kernel for the system's smooth operation.
- **totalMem** – displays the total memory accessible by the device kernel. The displayed values doesn't include the memory used for kernel fixed allocations like DMA buffers, RAM for the baseband CPU, etc.

(Developer.android.com, 2016)

8.2 Methods & Functions Used

To obtain specific functionality of the program, the following methods are used. The methods used are categorized according to their respective activity and their functionality is explained.

8.2.1 Main Menu activity

8.2.1.1 onBackPressed ( )

onBackPressed ( ) is called when the user presses the return/exit key in the device and a toast message is displayed indicating the user that he chose the exit option and **createDialogPopUp_ExitPrompt ( )** method is called.

```
@Override
public void onBackPressed() {
    Toast.makeText(getApplicationContext(), "You selected Exit option", Toast.LENGTH_LONG).show();
    createDialogPopUp_ExitPrompt();
}
```

Figure 38 - onBackPressed ( ) method code

8.2.1.2 createDialogPopUp_ExitPrompt ( )

The method is called by the **onBackPressed ( )** method and prompts the user whether he wants to exit the program by creating a dialog box; achieved by creating an AlertDialog in the name **YesNoPrompt**. Two buttons are declared within the AlertDialog as an **OnClickListener ( )** method set with **Yes** as the button display are declared within the **YesNoPrompt.setPositiveButton ( )** method. **YesNoPrompt.setNegativeButton ( )** includes an **OnClickListener ( )** method set with **No** as the display button. If the user clicks on the **Yes** button, the application suspends. If **No** button is selected, the application will return to the running instance of the main activity.

Figure 39 - Exit Prompt Pop-up display

```
private void createDialogPopUp_ExitPrompt() {
    AlertDialog.Builder YesNoPrompt = new AlertDialog.Builder(this);
    YesNoPrompt.setMessage("Are you sure you want to exit?");
    YesNoPrompt.setCancelable(false);

    YesNoPrompt.setPositiveButton("Yes",(dialog, which) → {
            MainActivity.super.onBackPressed();
    });
    YesNoPrompt.setNegativeButton("No", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
        }
    });
    YesNoPrompt.create().show();
}
```

Figure 40 - Exit Prompt Pop-up code

8.2.1.3 CreateDialogPopUp_AdHocNetworkImplementationPrompt ( )

The method builds an AlertDialog in the name **AdHocNetworkImplement** within the activity and a message is displayed to the user displaying **Implementing HotSpot**. A button displaying **Ok** is

set within the **AlertDialog ( )** method along with an **OnClickListener ( )** method utilizing the **AdHocNetworkImplement.setPositiveButton ( )** method. If the user clicks the **Ok** button, the wireless settings activity is opened by calling the intent **Settings.ACTION_WIRELESS_SETTINGS** within the method **startActivityForResult (**).

```
private void createDialogPopUp_AdHocNetworkImplementationPrompt() {
    AlertDialog.Builder AdHocNetworkImplement = new AlertDialog.Builder(this);
    AdHocNetworkImplement.setMessage("Implementing HotSpot");
    AdHocNetworkImplement.setCancelable(false);
    AdHocNetworkImplement.setPositiveButton("Ok", (dialog, which) -> {
            startActivityForResult(new Intent(Settings.ACTION_WIRELESS_SETTINGS), 0);

    });
    AdHocNetworkImplement.create().show();
}
```

Figure 41- CreateDialogPopUp_AdHocNetworkImplementationPrompt ( ) method code

8.2.1.4 CreateDialogPopUp_AdHocNetworkSwitchOffPropmpt ( ) method

The method follows the same steps as the above described method, only varies at the message displayed to the user containing **Turning off the HotSpot**.

```
private void createDialogPopUp_AdHocNetworkSwitchOffPropmpt() {
    AlertDialog.Builder AdHocNetworkTurnOff = new AlertDialog.Builder(this);
    AdHocNetworkTurnOff.setMessage("Turning off the HotSpot");
    AdHocNetworkTurnOff.setCancelable(false);
    AdHocNetworkTurnOff.setPositiveButton("Ok", (dialog, which) -> {
            startActivityForResult(new Intent(Settings.ACTION_WIRELESS_SETTINGS), 0);
    });
    AdHocNetworkTurnOff.create().show();
}
```

Figure 42- CreateDialogPopUp_AdHocNetworkSwitchOffPropmpt ( ) method code

8.3.1.5 BroadcastReceiver Battery_Info_Status

The BroadcastReceiver declares an onReceive () method with Context and Intent functions which will be active and valid only during the duration of the specific function. Once returned from the specific method/activity, the system discards the values and clears the threads considering the object is no longer active. (Developer.android.com, 2016). The BroadcastReceiver is used to

display the device battery sensor information to the user by calling the **Intent.ACTION_BATTERY_CHANGED** embedded in the BatteryManager API. The following components are monitored by the application.

```
public BroadcastReceiver Battery_Info_Status = (context, intent) -> {

        TextView Battery_Status_Display_MaintextView = (TextView) findViewById(R.id.Battery_Status_Display_MaintextView);
        TextView Battery_Status_Source_textView = (TextView) findViewById(R.id.Battery_Status_Source_textView);
        TextView Battery_Temperature_Display_Main_textView = (TextView) findViewById(R.id.Battery_Temperature_Display_Main_textView);
        TextView Battery_Capacity_Display_Main_textView = (TextView) findViewById(R.id.Battery_Capacity_Display_Main_textView);
        if (intent.getAction().equals(Intent.ACTION_BATTERY_CHANGED)) ;
        {
            int status = intent.getIntExtra("status", BatteryManager.BATTERY_STATUS_UNKNOWN);
            switch (status) {
                case BatteryManager.BATTERY_STATUS_CHARGING:
                    Status = "Charging";
                    break;
                case BatteryManager.BATTERY_STATUS_DISCHARGING:
                    Status = "Discharging";
                    break;
                case BatteryManager.BATTERY_STATUS_NOT_CHARGING:
                    Status = "Not charging";
                    break;
                case BatteryManager.BATTERY_STATUS_FULL:
                    Status = "Full";
                    break;
                default:
                    Status = "";
            }
            Battery_Status_Display_MaintextView.setText(Status);
```

```
            int source = intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
            switch (source) {
                case BatteryManager.BATTERY_PLUGGED_USB:
                    Source = "USB";
                    break;
                case BatteryManager.BATTERY_PLUGGED_AC:
                    Source = "AC";
                    break;
                case BatteryManager.BATTERY_PLUGGED_WIRELESS:
                    Source = "Wireless";
                    break;
                default:
                    Source = "Unknown";
            }
            Battery_Status_Source_textView.setText(Source);

            Temperature = intent.getIntExtra(BatteryManager.EXTRA_TEMPERATURE, 0);
            Battery_Temperature_Display_Main_textView.setText(String.valueOf(intent.getIntExtra("Temperature", Temperature / 10) + " C"));

            Battery_Capacity_Display_Main_textView.setText(String.valueOf(intent.getIntExtra("voltage", 0) + "mV"));

            Battery_level = intent.getIntExtra("level", 0);
        }

};
```

Figure 43 - Battery_Info_Status BroadcastReceiver code

8.2.1.5.1 Charging status

By equalizing the integer variable **status** to the **BATTERY_STATUS_UNKNOWN** method in BatteryManager API, the battery charge/discharge status is discovered.

- If the **BatteryManager.BATTERY_STATUS_CHARGING** returns true, the user is notified the battery is charging.
- If the **BatteryManager.BATTERY_STATUS_DISCHARGING** returns true, the user is notified the battery is discharging.
- If the **BatteryManager.BATTERY_STATUS_FULL** returns true, the user is notified the battery is plugged in to a power source and has been charged to the full capacity.
- If the **BatteryManager.BATTERY_STATUS_NOT_CHARGING** returns true, the user is notified the battery is plugged into a power source, but the battery isn't charging.

A string variable **Status** is declared and the text message relevant to the condition is assigned. The return value from the **Status** variable is displayed on the textbox **Battery_Status_Display_MaintextView**.

```
int status = intent.getIntExtra("status", BatteryManager.BATTERY_STATUS_UNKNOWN);
switch (status) {
    case BatteryManager.BATTERY_STATUS_CHARGING:
        Status = "Charging";
        break;
    case BatteryManager.BATTERY_STATUS_DISCHARGING:
        Status = "Discharging";
        break;
    case BatteryManager.BATTERY_STATUS_NOT_CHARGING:
        Status = "Not charging";
        break;
    case BatteryManager.BATTERY_STATUS_FULL:
        Status = "Full";
        break;
    default:
        Status = "";
```

Figure 44 - Charging Status code

8.3.1.5.2 Plugged in source

The **EXTRA_PLUGGED** method in BatteryManager API is exploited to detect the plugged in charger type. The method is called within the intent **getIntExtra** and is assigned to the integer variable source.

- If the **BatteryManager.BATTERY_PLUGGED_AC** returns true, the user is notified the device is plugged into an AC power source.
- If the **BatteryManager.BATTERY_PLUGGED_USB** returns true, the user is notified the device is plugged into a USB power source.
- If the **BatteryManager.BATTERY_PLUGGED_WIRELESS** returns true, the user is notified the device is plugged into a wireless charging source.
- If the BatteryManager I unable to identify the plugged in source, **Unknown** is displayed.

The text message relevant to the method is assigned to the String variable **Source** and is displayed in the textbox **Battery_Status_Source_textView**.

```
int source = intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
switch (source) {
    case BatteryManager.BATTERY_PLUGGED_USB:
        Source = "USB";
        break;
    case BatteryManager.BATTERY_PLUGGED_AC:
        Source = "AC";
        break;
    case BatteryManager.BATTERY_PLUGGED_WIRELESS:
        Source = "Wireless";
        break;
    default:
        Source = "Unknown";
```

Figure 45 - Plugged in source

8.2.1.5.3 Temperature

The **EXTRA_TEMPERATURE** method returns the existing battery temperature if a battery is present in the device. Using the **getIntExtra** intent, the method is declared and is assigned to the **Temperature** variable declared as a public integer within the main class. The return value of Temperature variable is converted to a string value using the function **String.valueOf** and

displayed to the user through the textbox **Battery_Temperature_Display_Main_textView**; the variable **Temperature** is set to return the value in Degrees Celsius (ºC)

```
Temperature = intent.getIntExtra(BatteryManager.EXTRA_TEMPERATURE, 0);
Battery_Temperature_Display_Main_textView.setText(String.valueOf(intent.getIntExtra("Temperature", Temperature / 10) + " C"));
```

Figure 46 - Temperature display code

8.2.1.5.4 Voltage

The **voltage** method is referred by the **intent ACTION_BATTERY_CHANGED**. The returned integer value is converted into a string value using the function "String.valueOf" and assigned to the textbox **Battery_Capacity_Display_Main_textView** and displayed to the user in millivolts (mV).

```
Battery_Capacity_Display_Main_textView.setText(String.valueOf(intent.getIntExtra("voltage", 0) + "mV"));
```

Figure 47 - Voltage display code

8.2.1.5.5 Battery level

The **battery level** is referred by the intent **ACTION_BATTERY_CHANGED**. The returned integer value is converted into a string value using the function "String.valueOf" and assigned to the textbox **Battery_Capacity_Display_Main_textView** and displayed to the user in an integer value

```
Battery_level = intent.getIntExtra("level", 0);
```

Figure 48 - Battery level display code

8.2.1.6 totalUploadDownload ( )

As explained in the API description, the TrafficStats API provides functionality to monitor the total uploaded and downloaded bytes and packets from the device start up. The application captures transmitted and received bytes and packets for the ongoing session and reset the values when the activity is suspended. The above limitation is overcome by calling the API at the application startup and record the existing values; then again record the values when the method

is called. Once the two instances are obtained, the values read at the application startup are subtracted from the values read within the method. This method displays the total bytes and packets transmitted and received within the ongoing session. The methodology followed to calculate received and transmitted bytes are as follows.

- **startRXBytes = TrafficStats.getTotalRxBytes ( )** – variable **startRXByte**s is defined and the values from the method **getTotalRXBytes ( )** under the TrafficStats API is assigned.
- **startMobileRXBytes = TrafficStats.getMobileRXBytes ( )** – values from the method **getMobileRXBytes ( )** under TrafficStats API is assigned to the variable s**tartMobileRXBytes**.
- **wirelessRXBytes = startRXBytes – startMobileRXBytes** - The **wireless RX bytes** are calculated by subtracting the variable **startMobileRXBytes** from the **startRXBytes**; as there is no specific method defined within the API to calculate the wireless bytes.

Following the above calculation, total received bytes can be calculated. **RX** defines "**received**" bytes/packets while **TX** defines "**transmitted**" bytes/packets.

The total transmitted bytes are calculated following the same procedure described above. The variables are assigned as follows.

- **startTXBytes = TrafficStats.getTotalTxBytes ( )**
- **startMobileTxBytes = TrafficStats.getMobileTxBytes ( )**
- **wirelessTXBytes = startTXBytes – startMobileTXBytes**

The API requires minimum API level 8 to operate. To check the device status, the condition **TrafficStats.UNSUPPORTED** is checked on **startTXBytes** and **startRXBytes** variables. If the above condition returns true, the user is displayed the SDK is not supported in the device. Else the Runnable "**runnable_total_bytes**" is called is called within a **handler.Post ( )** and operates in a UI thread.

```
public String totalUploadDownload() {
    startRXBytes = TrafficStats.getTotalRxBytes();
    startMobileRXBytes = TrafficStats.getMobileRxBytes();
    startWirelessRXBytes = startRXBytes - startMobileRXBytes;

    startTXBytes = TrafficStats.getTotalTxBytes();
    startMobileTXBytes = TrafficStats.getMobileTxBytes();
    startWirelessTXBytes = startTXBytes - startMobileTXBytes;

    if (startRXBytes == TrafficStats.UNSUPPORTED || startTXBytes == TrafficStats.UNSUPPORTED) {
        Display = "SDK Unsupported";
    } else {
        handler.post(runnable_total_bytes);
    }
    return Display;
}
```

Figure 49 - totalUploadDownload ( ) method code

8.2.1.6.1 runnable_total_bytes

A new runnable to operate on a new thread is declared and instances of TrafficStats API methods **getTotalTxBytes ( ), getMobileTxBytes ( ), getTotalRxBytes ( )** and **getMobileRxBytes ( )** are assigned to variables **EndTXBytes, EndMobileTXBytes, EndRXBytes** and **EndMobileRXBytes** respectively. The wireless **TX** and **RX** bytes are calculated by subtracting the respective mobile bytes from the total bytes.

Variable **EndWirelessTXBytes** is obtained by subtracting **EndMobileTXBytes** from **EndTXBytes** and the **EndWirelessRXBytes** is obtained by subtracting the **EndMobileRXBytes** from **EndRXBytes** respectively. To calculate the transmitted wireless bytes for the session, the byte count at the application launch (**startTXBytes/startRXBytes**) is subtracted from the byte count at the second instance (**EndTXBytes/EndRXBytes**). "**totalSentKBytes**" variable is declared a double data type and the value **EndTotalWirelessTXBytes / 1000** is assigned and type casted to data type "**double**".

The API provides functionality to provide the output in byte format. But displaying in byte format will make the return values incredibly long, making it unable to be read and understood by the user. Therefore, the bytes are divided by 1000 to convert to Kbytes, the fundamental count display of the application. The variable "**totalSentKBytes**" is passed down to a condition checker and checked for the following conditions.

- If the **totalSentKBytes** is lesser than 1024, the output is converted to a String value and the decimal count restricted up to 2 decimal places by utilizing "**%.2f**" and displayed in kB format.
- If the totalSentKBytes is greater than 1024 and lesser than 1048576, the output value is divided by 1024 to convert kB to MB. The value is assigned to the totalSentMB variable and capped at the 2nd decimal place utilizing the "%.2f". The resulting value is converted to a String is presented to the user in MB format.
- If the **totalSentKBytes** is greater than 1048576, the output is divided by 1048576 to convert kB to GB. The resulting value is assigned to the totalSentGB variable and capped at the 2nd decimal place using "%.2f" converting to a String format; presenting to the user in the GB format.

The total download byte calculation involves in the same procedure described above. The received/downloaded bytes are displayed to the user in the similar format. The application processes the methods real-time as a post handler is used to handle the runnable.

```
public Runnable runnable_total_bytes = () → {
        TextView Throughput_Total_Upload = (TextView) findViewById(R.id.Throughput_Total_Upload);
        long EndTXBytes = TrafficStats.getTotalTxBytes();
        long EndMobileTXBytes = TrafficStats.getMobileTxBytes();
        long EndWirelessTXBytes = EndTXBytes - EndMobileTXBytes;
        long EndTotalWirelessTXBytes = EndWirelessTXBytes - startWirelessTXBytes;
        double totalSentKBytes = (double) EndTotalWirelessTXBytes / 1000;

        if (totalSentKBytes < 1024) {
            String output = String.format("%.2f", totalSentKBytes);
            if (Throughput_Total_Upload != null) {
                Throughput_Total_Upload.setText(output + " kB");
            }
        } else if (totalSentKBytes >= 1024 && totalSentKBytes < (1024 * 1024)) {
            double totalSentMB = totalSentKBytes / 1024;
            String output = String.format("%.2f", totalSentMB);
            if (Throughput_Total_Upload != null) {
                Throughput_Total_Upload.setText(output + " MB");
            }
        } else if (totalSentKBytes >= 1048576) {
            double totalSentGB = totalSentKBytes / 1048576;
            String output = String.format("%.2f", totalSentGB);
            if (Throughput_Total_Upload != null) {
                Throughput_Total_Upload.setText(output + " GB");
            }
        }
```

(Developer.android.com, 2016). The timers and counters are set to initiate with the method declaration; not at the initiation of the application. This procedure is followed to increase the efficiency and the accuracy of the application.

Figure 50 - total bytes runnable code

8.2.1.7 UploadDownloadRates_Main ( )

The method **UploadDownloadRates_Main ( )** follows the similar concept of the above described **totalUploadDownload ( )** method; slightly varying as the application timestamps the instance at the method call and at the calculation declared within the runnable. The initial timestamp is assigned to the variable "**startTime**" using the System method **currentTimeMillis ( ).** The second timestamp is assigned to the variable "**endTime**" and executes after calculating the total wireless bytes sent/received in the runnable. A new runnable is assigned under the name "**runnable_wireless_speed**" and follows the same steps followed in runnable_total_bytes except for the differences given below.

```
        TextView Throughput_Total_Download = (TextView) findViewById(R.id.Throughput_Total_Download);
        long EndRXBytes = TrafficStats.getTotalRxBytes();
        long EndMobileRXBytes = TrafficStats.getMobileRxBytes();
        long EndWirelessRXBytes = EndRXBytes - EndMobileRXBytes;
        long EndTotalWirelessRXBytes = EndWirelessRXBytes - startWirelessRXBytes;
        double totalReceivedKBytes = (double) EndTotalWirelessRXBytes / 1000;
        if (totalReceivedKBytes < 1024) {
            String output = String.format("%.2f", totalReceivedKBytes);
            if (Throughput_Total_Download != null) {
                Throughput_Total_Download.setText(output + " kB");
            }
        } else if (totalReceivedKBytes >= 1024 && totalReceivedKBytes < (1024 * 1024)) {
            double totalReceivedMB = totalReceivedKBytes / 1024;
            String output = String.format("%.2f", totalReceivedMB);
            if (Throughput_Total_Download != null) {
                Throughput_Total_Download.setText(output + " MB");
            }
        } else if (totalReceivedKBytes >= (1024 * 1024)) {
            double totalReceivedGB = (totalReceivedKBytes) / 1048576;
            String output = String.format("%.2f", totalReceivedGB);
            if (Throughput_Total_Download != null) {
                Throughput_Total_Download.setText(output + " GB");
            }
        }
        handler.post(runnable_total_bytes);
};
```

```
public String UploadDownloadRates_Main() {
    startTime = System.currentTimeMillis();
    startRXBytes = TrafficStats.getTotalRxBytes();
    startMobileRXBytes = TrafficStats.getMobileRxBytes();
    startWirelessRXBytes = startRXBytes - startMobileRXBytes;

    startTXBytes = TrafficStats.getTotalTxBytes();
    startMobileTXBytes = TrafficStats.getMobileTxBytes();
    startWirelessTXBytes = startTXBytes - startMobileTXBytes;

    if (startRXBytes == TrafficStats.UNSUPPORTED || startTXBytes == TrafficStats.UNSUPPORTED) {
        Display = "SDK Unsupported";
    } else {
        handler.post(runnable_wireless_Speed);
    }
    return Display;
}
```

Figure 51 - UploadDownloadRates_Main ( ) method code

8.2.1.7.1 runnable_wireless_speed

The time interval between the timestamps are calculated by subtracting the initial timestamp from the final timestamp and the resulting value is assigned to the variable timeDifference. The above procedure is explained below.

timeDifference = (endTime - startTime)/1000

As timestamping is done in milliseconds, the resulting value should be divided by 1000 to obtain the time difference in seconds; as the upload and download rates per second are displayed.

The variable "**EndTotalWirelessTXBytes**" is assigned the difference in the values of initial and final wireless bytes received/transmitted within the network for the given session. The assigned value of the "**EndTotalWirelessTXBytes**" variable is divided by 1024 to display the result in kB and divided by the timeDifference and assigned to the "**sentKBytes**" variable to find the upload/download rate per second.

The variable sentKBytes is then passed down to the condition checker to check whether the variable matches the given conditions.

- If the sentKBytes is lesser than 1024, the output is converted to a String value and the decimal count restricted up to 2 decimal places by utilizing "%.2f" and displayed in kB/s format.

- If the sentKBytes is greater than or equal to 1024, the output value is divided by 1024 to convert kB to MB. The value is assigned to the SentMBytes variable and capped at the 2nd decimal place utilizing the "%.2f". The resulting value is converted to a String is presented to the user in MB/s format.
- The upload and download rates are not displayed in GB/s as the maximum network speed achievable through wireless is within the range of MB/s at the time of documentation.

A similar procedure to the above explained is used to calculate the download rates assigned to the "**receivedKBytes**" variable.

```
Runnable runnable_wireless_Speed = () → {

        TextView Upload_Rate_Display_Main_textView = (TextView) findViewById(R.id.Upload_Rate_Display_Main_textView);
        long EndTXBytes = TrafficStats.getTotalTxBytes();
        long EndMobileTXBytes = TrafficStats.getMobileTxBytes();
        long EndWirelessTXBytes = EndTXBytes - EndMobileTXBytes;
        //the code below added the infinity loop; mention it in documentation
        double EndTotalWirelessTXBytes = EndWirelessTXBytes - startWirelessTXBytes;
        endTime = System.currentTimeMillis();
        timeDifference = (endTime - startTime) / 1000;
        double SentKBytes = ((EndTotalWirelessTXBytes / 1024) / (timeDifference));
        if (SentKBytes < 1024) {
            String output = String.format("%.2f", SentKBytes);
            Upload_Rate_Display_Main_textView.setText(output + " kB/s");
        } else if (SentKBytes >= 1024) {
            double SentMBytes = SentKBytes / 1024;
            String output = String.format("%.2f", SentMBytes);
            Upload_Rate_Display_Main_textView.setText(output + " MB/s");
        }

        TextView Download_Rate_Display_Main_textView = (TextView) findViewById(R.id.Download_Rate_Display_Main_textView);
        long EndRXBytes = TrafficStats.getTotalRxBytes();
        long EndMobileRXBytes = TrafficStats.getMobileRxBytes();
        long EndWirelessRXBytes = EndRXBytes - EndMobileRXBytes;
        double EndTotalWirelessRXBytes = EndWirelessRXBytes - startWirelessRXBytes;
        endTime = System.currentTimeMillis();
        timeDifference = (endTime - startTime) / 1000;
        double ReceivedKBytes = ((EndTotalWirelessRXBytes / 1024) / timeDifference);
        if (ReceivedKBytes < 1024) {
            String output = String.format("%.2f", ReceivedKBytes);
            Download_Rate_Display_Main_textView.setText(output + " kB/s");
        } else if (ReceivedKBytes >= 1024) {
            double ReceivedMBytes = ReceivedKBytes / 1024;
            String output = String.format("%.2f", ReceivedMBytes);
            Download_Rate_Display_Main_textView.setText(output + " MB/s");
        }
        handler.post(runnable_wireless_Speed);
};
```

Figure 52 - wireless speed runnable code

8.2.1.8 UploadDownloadPackets ( )

The total uploaded and downloaded packets are displayed with the methods and functions called within the **UploadDownloadPackets ( )** method. The **UploadDownloadPackets ( )** operates similar to the method **totalUploadDownload ( )** and follows the same concept, but varies with the following methods.

Initial capture is done on packets using the methods **getTotalRxPackets ( ), getMobileRxPackets ( ) and getTotalTxPackets ( ), getMobileTxPackets ( )** to calculate the mobile **RX** and **TX** packets and total RX and TX packets respectively.

The wireless RX and TX packets are calculated by subtracting the mobile packets from the total packets accordingly.

```
public String UploadDownloadPackets() {

    startRXPackets = TrafficStats.getTotalRxPackets();
    startMobileRXPackets = TrafficStats.getMobileRxPackets();
    startWirelessRXPackets = startRXPackets - startMobileRXPackets;

    startTXPackets = TrafficStats.getTotalTxPackets();
    startMobileTXPackets = TrafficStats.getMobileTxPackets();
    startWirelessTXPackets = startTXPackets - startMobileTXPackets;

    if (startTXPackets == TrafficStats.UNSUPPORTED || startRXPackets == TrafficStats.UNSUPPORTED) {
        Display = "SDK Unsupported";
    } else {
        handler.post(runnable_total_packets);
    }
    return Display;
}
```

Figure 53 - UploadDownloadPackets ( ) method code

8.2.1.8.1 runnable_total_packets

A new runnable is declared in the name "**runnable_total_packets**", where the second instance of capture is carried out similar to the initial capture. The difference between the initial and final wireless packet counts display the packets uploaded and downloaded at the given instance. The application uses multiple handlers and runnables running on multiple threads. This will increase the system resource utilization, but will benefit the program as overloading all runnables on a single thread can cause the application to crash.

```
private Runnable runnable_total_packets = () → {
        TextView Wireless_Upload_Packets = (TextView) findViewById(R.id.Wireless_Upload_Packets);

        long EndTXPackets = TrafficStats.getTotalTxPackets();
        long EndMobileTXPackets = TrafficStats.getMobileTxPackets();
        long EndWirelessTXPackets = EndTXPackets - EndMobileTXPackets;

        long EndTotalTXPackets = EndWirelessTXPackets - startWirelessTXPackets;
        double sentTXPackets = (double) EndTotalTXPackets;
        if (Wireless_Upload_Packets != null) {
            Wireless_Upload_Packets.setText(Double.toString(sentTXPackets));
        }

        TextView Wireless_Download_Packets = (TextView) findViewById(R.id.Wireless_Download_Packets);

        long EndRXPackets = TrafficStats.getTotalRxPackets();
        long EndMobileRXPackets = TrafficStats.getMobileRxPackets();
        long EndWirelessRXPackets = EndRXPackets - EndMobileRXPackets;

        long EndTotalRXPackets = EndWirelessRXPackets - startWirelessRXPackets;
        double sentRXPackets = (double) EndTotalRXPackets;
        if (Wireless_Download_Packets != null) {
            Wireless_Download_Packets.setText(Double.toString(sentRXPackets));
        }
        handler.post(runnable_total_packets);
};
```

Figure 54 - total packets runnable code

8.2.1.9 **SetMobileDataOn ( )**

The **mobileDataOn ( )** method is used to integrate the functions of **setMobileDataEnabled ( )** method with the "**Auto_Pilot_OnOff_Switch**" toggle button activity. The process allows the activity to enable and disable the mobile data connection programmatically on rooted devices. The process is achieved by following the given procedure.

If the **OnClickListener ( )** method returns true, the user is notified through a Toast displaying "**Switching on Mobile Data**". The application implements a new Runnable within a new Thread and call **setMobileDataEnabled ( )** method within the main menu activity and passes the value Boolean "**true**" to the method activating the disabled mobile network connection. If the task returns successful, the user is notified with a Toast "**Mobile data switched on**" which runs on a UI thread. The toast messages are set to display for a short time by configuring "**Toast.LENGTH_SHORT**".

```
public void SetMobileDataOn() {
    Auto_Pilot_OnOff_Switch.setOnClickListener((view) → {

            Toast.makeText(MainActivity.this,"Switching on mobile data",Toast.LENGTH_SHORT).show();

            (new Thread((Runnable) () → {

                    setMobileDataEnabled(MainActivity.this,true);

                    runOnUiThread(() → {
                            Toast.makeText(MainActivity.this, "Mobile data switched on", Toast.LENGTH_SHORT).show();
                    });
            })).start();
    });
}
```

Figure 55 - SetMobileDataOn ( ) method code

8.2.1.10 SetMobileDataOff ( )

mobileDataOff ( ) method operates similar to the functionality of the **SetMobileDataOn ( )** method explained above and integrates the **setMobileDataEnabled ( )** functions and pass the Boolean value "**false**" to the method, thus disabling the active mobile network connection.

```
public void SetMobileDataOff() {
    Auto_Pilot_OnOff_Switch.setOnClickListener((view) → {
            Toast.makeText(MainActivity.this,"Switching off mobile data",Toast.LENGTH_SHORT).show();

            (new Thread((Runnable) () → {

                    setMobileDataEnabled(MainActivity.this,false);

                    runOnUiThread(() → {
                            Toast.makeText(MainActivity.this, "Mobile data turned off", Toast.LENGTH_SHORT).show();
                    });


            })).start();
    });
}
```

Figure 56 - SetMobileDataOff ( ) method code

8.2.1.11 AutoPilotMode ( )

The **AutoPilotMode ( )** method is used to control the device activity programmatically. The method first checks for the rooted status of the mobile device by calling the method **isRooted ( )** and checks for the return Boolean value. If the **isRooted ( )** returns true, the application checks the following conditions and operates the device accordingly. Sensor readings from battery

temperature, plugged in source, battery level and battery status are called to check for the conditions.

- If the device battery temperature is above 45 degrees Celsius, and plugged into an AC power source and the battery is at charging status, mobile network connection is turned off by calling the method **SetMobileDataOff ( )** and the user is notified with "**Suspended**" text in the operating mode display textbox and the device activity is displayed as "**High**" in the device activity textbox.
- If the battery temperature is less than 45 degrees Celsius and greater than or equal to 40 degrees Celsius, and plugged into a USB power source and the device is at charging state, the mobile network is turned off by calling the **SetMobileDataOff ( )** method and the user is notified that the application is suspended through the "**Suspended**" text in the operating mode display textbox and the device activity is displayed as "**High**" in the device activity textbox.
- If the device is not plugged into any power source and battery is operating between the temperatures 35 and 40 degrees Celsius and the battery level reaches below 20%, the mobile network is suspended and the user is notified through the "**Suspended**" text in the operating mode display textbox and the device activity is displayed as "**Inactive**" in the device activity textbox.
- If the device is operating below the given threshold values, the device turns on the mobile data connection with the aid of the **SetMobileDataOn ( )** method and displays the user "**Active**" text within the operating mode display textbox and the device activity is displayed as "**Low**" in the device activity textbox.

Depending on the root status of the device, the mobile data will be enabled and disabled programmatically. However on non-rooted devices, the auto-pilot mode will check the above conditions similarly, but mobile data will not be controlled programmatically. In fact, the user is advised to follow steps in order to conserve the remaining power of the device.

```
public String AutoPilotMode() {
    TextView Operating_Mode = (TextView) findViewById(R.id.Operating_Mode);
    TextView Device_Activity_Display_MaintextView = (TextView) findViewById(R.id.Device_Activity_Display_MaintextView);
    if (isRooted() == true) {
        if ((Temperature >= 45) && (Source == "AC") && (Status == "Charging")) {
            SetMobileDataOff();
            Operating_Mode.setText("Suspended");
            Device_Activity_Display_MaintextView.setText("High");
            Toast.makeText(getApplicationContext(), "Enable Power Saving mode", Toast.LENGTH_LONG).show();
        } else if ((Temperature < 45) && (Temperature >= 40) && (Source == "USB") && (Status == "Charging")) {
            SetMobileDataOff();
            Operating_Mode.setText("Suspended");
            Toast.makeText(getApplicationContext(), "Enable Power Saving mode", Toast.LENGTH_LONG).show();
            Device_Activity_Display_MaintextView.setText("High");
        } else if ((Temperature > 40) && (Temperature >= 35) && (Battery_level < 20)) {
            SetMobileDataOff();
            Operating_Mode.setText("Suspended");
            Toast.makeText(getApplicationContext(), "Enable Power Saving mode", Toast.LENGTH_LONG).show();
            Device_Activity_Display_MaintextView.setText("Inactive");
        } else {
            SetMobileDataOn();
            Operating_Mode.setText("Active");
            Device_Activity_Display_MaintextView.setText("Low");
            Toast.makeText(getApplicationContext(), "Disable Power Saving mode", Toast.LENGTH_LONG).show();
        }
    } else {
        Toast.makeText(getApplicationContext(), "Your device is not Rooted", Toast.LENGTH_LONG).show();
        Operating_Mode.setText("Unsupported");
        if ((Temperature >=45) && (Source =="AC") && (Status == "Charging")){
            Device_Activity_Display_MaintextView.setText("High");
            Toast.makeText(getApplicationContext(), "High temperature. Please turn on the power saving mode", Toast.LENGTH_LONG).show();
        }
        else if ((Temperature < 45) && (Temperature >= 40) && (Source == "USB") && (Status == "Charging")){
            Device_Activity_Display_MaintextView.setText("High");
            Toast.makeText(getApplicationContext(), "High temperature. Please turn on the power saving mode", Toast.LENGTH_LONG).show();
        }
        else if ((Temperature > 40) && (Temperature >= 35) && (Battery_level < 20)){
            Device_Activity_Display_MaintextView.setText("Inactive");
            Toast.makeText(getApplicationContext(), "Battery low. Turn off the hotspot", Toast.LENGTH_LONG).show ();
        }
        else {
            Device_Activity_Display_MaintextView.setText("Low");
        }
    }
    return "";
}
```

Figure 57 - AutoPilotMode ( ) method code

8.2.1.12 setMobileDataEnabled ( )

The method utilizes ConnectivityManager API and creates an instance with the system service context **Context.CONNECTIVITY_SERVICE** and the accessibility to the "**Service**" field declared under ConnectivityManager class is set "**true**". The method is used to enable and disable the mobile data connection within the ConnectivityManagerClass and assigned the values to "**setMobileDataEnabled**" and the Boolean value passed down from the **SetMobileDataOn ( )** and **SetMobileDataOff ( )** methods respectively. Depending on the **Boolean.TYPE** variable, the method switches mobile data on/off programmatically.

```
private void setMobileDataEnabled(Context context, boolean enabled) {
try {
    final ConnectivityManager connectivityManager = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
    final Class connectivityManagerClass = Class.forName(connectivityManager.getClass().getName());
    final Field ConnectivityManagerField = connectivityManagerClass.getDeclaredField("Service");
    ConnectivityManagerField.setAccessible(true);
    final Object ConnectivityManager = ConnectivityManagerField.get(connectivityManager);
    final Class ConnectivityManagerClass = Class.forName(ConnectivityManager.getClass().getName());
    final Method setMobileDataEnabledMethod = ConnectivityManagerClass.getDeclaredMethod("setMobileDataEnabled", Boolean.TYPE);
    setMobileDataEnabledMethod.setAccessible(true);
    setMobileDataEnabledMethod.invoke(ConnectivityManager, enabled);
} catch (NoSuchMethodException e) {
    e.printStackTrace();
} catch (InvocationTargetException e) {
    e.printStackTrace();
} catch (NoSuchFieldException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
}
```

Figure 58 - setMobileDataEnabled ( ) method code

8.2.1.13 isRooted ( )

The rooting process available for android devices allow the user to unlock the operating system features and enable customizations and modifications on hardware and software components. The methods **setMobileDataEnabled ( )** programmatically enables and disables the mobile network connectivity status; thus requiring hardware modification. The root status of a device can be checked by calling the following methods which check for the modifications and return "**true**" if the conditions checked are available in the system.

```
public static boolean isRooted() {
    return checkRootMethod_buildTags() || checkRootMethod_SU_Path() || checkRootMethod_SU_xbin();
}
```

Figure 59 - isRooted ( ) method code

8.2.1.13.1 checkRootMethod_buildTags ( )

The method checks for the "**android.os.Build.TAGS**" and assign the return value to the String variable "**buildTags**". The method returns true if the buildTags variable is not a null value and contains "**test-keys**" within the called method.

```
private static boolean checkRootMethod_buildTags() {
    String buildTags = android.os.Build.TAGS;
    return buildTags != null && buildTags.contains("test-keys");
}
```

Figure 60 - checkRootMethod_buildTags ( ) method code

8.2.1.13.2 checkRootMethod_SU_Patch ( )

Checks for the different modified paths during the root process. The different paths are added to a string array "**pathChecker**" and checks the system for existing paths with the paths defined in the array. If one or multiple paths are detected, the method will return true.

```
private static boolean checkRootMethod_SU_Path() {
    String[] pathChecker = { "/system/app/Superuser.apk",
            "/sbin/su",
            "/system/bin/su",
            "/system/xbin/su",
            "/data/local/xbin/su",
            "/data/local/bin/su",
            "/system/sd/xbin/su",
            "/system/bin/failsafe/su",
            "/data/local/su" };
    for (String pathExisting : pathChecker) {
        if (new File(pathExisting).exists()) return true;
    }
    return false;
}
```

Figure 61 - checkRootMethod_SU_Patch ( ) method code

8.2.1.13.3 checkRootMethod_SU_xbin ( )

The method creates a process and adds a **runtime ( )** and executes String array with variables "**/system/xbin/which**" and "**su**". Using a BufferedReader, the input is read with the aid of InputStreamReader and if the buffered reader returns a value, the method will return Boolean "true" to the main method. The above method is executed within a "try" block and a "throwable" exception is caught within the "catch" block. If the "catch" block returns "throwable", the method returns Boolean "**false**". After the execution of the try and catch block, and the process is not empty, the process is destroyed by calling **process.destroy ( )** method and release the thread the process is currently running on.

```
private static boolean checkRootMethod_SU_xbin() {
    Process process = null;
    try {
        process = Runtime.getRuntime().exec(new String[] { "/system/xbin/which", "su" });
        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(process.getInputStream()));
        if (bufferedReader.readLine() != null) return true;
        return false;
    } catch (Throwable throwable) {
        return false;
    } finally {
        if (process != null) process.destroy();
    }
}
```

Figure 62 - checkRootMethod_SU_xbin ( ) method code

8.2.2 Signal Strength activity

8.2.2.1 getSubnetMask ( )

The method obtains the subnet mask of the network by calling the DhcpInfo method declared in WifiManager API and assign it to the integer variable **SubnetMask**. The subnet mask of the network is obtained by the netmask function in the DhcpInfo method and the value returned is passed to the textbox **WiFi_NETMASK_DISPLAY**. The return value is passed to the bitwise shift operator (“>>>”) and masked with the value of “**& 0xff**” to format the integer value into dotted-decimal format.

```
private String getSubnetMask() {
    DhcpInfo dhcpinfo = wifiManager.getDhcpInfo();
    int SubnetMask = dhcpinfo.netmask;
    return ((SubnetMask) & 0xFF)+"."+
            ((SubnetMask>>>=8)&0xFF)+"."+
            ((SubnetMask>>>=8)&0xFF)+"."+
            (((SubnetMask)>>>=8)&0xFF);
}
```

Figure 63 - getSubnetMask ( ) method code

8.2.2.2 getDefaultGateway ( )

The declared method obtains the default gateway of the network through the DhcpInfo method found in WifiManager API and assigns it to the integer variable “**DefaultGateway**”. Following similar methodology to the above defined **getSubnetMask ( )** method, the default gateway is obtained by calling the gateway function declared in the DhcpInfo method. The return value passes

through the bitwise operator (“>>>”) and the “& 0xff” mask to format the “**DefaultGateway**” integer into dotted decimal format.

```
private String getDefaultGateway(){
    DhcpInfo dhcpinfo = wifiManager.getDhcpInfo();
    int DefaultGateway = dhcpinfo.gateway;
    return (DefaultGateway & 0xFF)+"."+
            ((DefaultGateway>>>=8)&0xFF)+"."+
            ((DefaultGateway>>>=8)&0xFF)+"."+
            ((DefaultGateway>>>=8)&0xFF);

}
```

Figure 64 - getDefaultGateway ( ) method code

8.2.2.3 WiFiSignalChannel ( )

The method **WiFiSignalChannel ( )** will display the operating frequency range of the connected network. The method checks the operating frequency by calling the function **getConnectionInfo ( )** from WifiManager API and checks the wireless network frequency is within the declared ranges. If the read frequency is within the ranges 2412 and 2484 MHz, the user is notified the network operates within the range 2.4GHz. If the frequency is between 3657.5 and 3690, the network operates in the 3.6GHz range. Else, if the range varies between 5825MHz and 5180MHz, 5GHz range is displayed.

```
private String WiFiSignalChannel() {
    WifiInfo wifiInfo = wifiManager.getConnectionInfo();
    String WiFiFrequencyChannel = "";
    if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.LOLLIPOP) {
        double Frequency = wifiInfo.getFrequency();
        if ((Frequency>= 2412)&& (Frequency<=2484)) {
            WiFiFrequencyChannel = "2.4GHz";
        } else if (Frequency >= 3657.5 && Frequency <= 3690) {
            WiFiFrequencyChannel = "3.6GHz";
        } else if  (Frequency >= 5825 && Frequency <= 5180) {
            WiFiFrequencyChannel = "5GHz";
        }
    }
    else {
        WiFiFrequencyChannel = "SDK Unsupported";
    }
    return WiFiFrequencyChannel;
}
```

Figure 65 - WiFiSignalChannel ( ) method code

8.2.2.4 WiFiSignalStrength ( )

The **WiFiSignalStrength ( )** method reads the value from the method **getWifiState ( )** declared in WifiManager API and assigns the value to the integer variable **wifiState**. If the WifiManager returns **WIFI_STATE_ENABLED** true, the program obtain the details of the wireless network scan results from the **getScanResults ( )** method in WifiManager API. The **BSSID** result is compared and if the information is similar to the information on **WifiManager.getConnectionInfo ( ).getBSSID ( )** method, the signal level is obtained by calling the method **WifiManager.calculateSignalLevel ( )** which include the methods **getConnectionInfo ( ), getRssi ( )** and the signal level.

Two instances of signal levels are calculated, one instance at the node and the other instance at the host. The difference in the signal levels display the signal drop between the sender and the receiver. As the signal levels are displayed in decibel meters. Considering the difference in signal levels, the available signal strength is displayed to the user in 5 different levels in the **WiFi_SIGNAL_STRENGTH_DISPLAY** textbox. The signal strength is displayed to the user in **deciBel meters (dBm)**

- If the signal strength difference is greater than or equal to 80, the user is displayed the signal strength is "**Excellent**".
- If the signal strength is between 80 and 60, the user is displayed the signal strength is "**Average**".
- If the signal strength is between 60 and 40, the user is notified the signal strength is "**Medium**".
- If the signal strength is between 40 and 20, the user is notified the signal strength is "**Weak**".
- If the signal strength is less than 20, the user is notified the signal strength is "**Poor**".

```
private String WiFiSignalStrength () {
    int wifiState = wifiManager.getWifiState();
    String SignalStrength = "";
        if (wifiState == WifiManager.WIFI_STATE_ENABLED) {
            List<ScanResult> results = wifiManager.getScanResults();
            for (ScanResult result : results) {
                if (result.BSSID.equals(wifiManager.getConnectionInfo().getBSSID())) {
                    int level = WifiManager.calculateSignalLevel(wifiManager.getConnectionInfo().getRssi(), result.level);
                    int difference = ((level) * 100) / (result.level);
                    String deciBelmeters = String.valueOf(result.level);
                    if ((difference>=80)) {
                        SignalStrength = "Excellent " + deciBelmeters + " dBm";
                    } else if ((difference <80) && (difference>=60)) {
                        SignalStrength = "Average " + deciBelmeters + " dBm";
                    } else if ((difference <60) && (difference>=40)) {
                        SignalStrength = "Medium " + deciBelmeters + " dBm";
                    } else if ((difference <40) && (difference>=20)) {
                        SignalStrength = "Weak " + deciBelmeters + " dBm";
                    } else
                        SignalStrength = "Poor " + deciBelmeters + " dBm";
                }
            }
        }
    return SignalStrength;
}
```

Figure 66 - WiFiSignalStrength ( ) method code

8.2.2.5 getWiFiSecurityDetails ( )

The method **getWiFiSecurityDetails ( )** is used to display the encryption type used in the currently active network. The method checks the currently available network list by calling **getScanResults ( )** method from the WifiManager API. The network encryption capability is checked by calling the network.capabilities method and currently available security encryption types are added to the String array **SecurityModes [ ].** A for loop with the length of SecurityModes array is called and the return value from the network.capabilities is checked against the variables available within the SecurityModes array. The detected encryption algorithm is displayed in the textbox **ENCRYPTION_TYPE_DISPLAY**.

```
private String getWiFiSecurityDetails () {
    String SecurityMode = "";

     List<ScanResult> networkList = wifiManager.getScanResults();
      if (networkList != null) {
          for (ScanResult network : networkList) {
              String Capabilities = network.capabilities;
              String SecurityModes[] = {Constants.WEP, Constants.PSK, Constants.EAP, Constants.OPEN, Constants.WPA2};
              for (int i = 0; i < SecurityModes.length; i++) {
                  if (Capabilities.contains(SecurityModes[i])) {
                      SecurityMode =  SecurityModes[i];
                  }
              }
          }
      }
    return SecurityMode;
    }
```

Figure 67 - getWiFiSecurityDetails ( ) method code

8.2.2.5.1 public class Constants

The class holds the different types of encryption algorithm declaration and the respective variable assignment for the encryption algorithm. All variables are declared under String data type.

```
public class Constants {
    public static final String PSK = "PSK";
    public static final String WEP = "WEP";
    public static final String EAP = "EAP";
    public static final String OPEN = "Open";
    public static final String WPA2 = "WPA2";
}
```

Figure 68 - public class Constants code

8.2.2.6 getWiFiMACAddress ( )

The method calls the **getConnectionInfo ( )** method from the WifiManager API and returns the **getMacAddress ( )** method from WifiInfo API. As hardware address variables are displayed in upper case letters, the return value is elevated to upper case letters using **toUpperCase ( )** method.

```
public String getWiFiMACAddress()
{
    WifiInfo wifiInfo = wifiManager.getConnectionInfo();
    return(wifiInfo.getMacAddress().toUpperCase());
}
```

Figure 69 - getWiFiMACAddress ( ) method code

8.2.2.7 getFrequency ( )

The network operating frequency is calculated by calling the **getFrequency ( )** method in the WifiInfo API. The returned value from the **getFrequency (**) method is an integer value, therefore the return value is converted to a String variable before displaying it within the textbox **Wi_Fi_FREQUENCY_DISPLAY** through the method **String.valueOf ( ).** The **getFrequency ( )** method supports android API 21 and upwards. The devices running below android API 21 displays "**SDK Unsupported**" within the frequency display textbox.

```
public String getFrequency() {
    WifiInfo wifiInfo = wifiManager.getConnectionInfo();
    String WiFiFrequency = "";
    if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.LOLLIPOP) {
        int Frequency = wifiInfo.getFrequency();
         WiFiFrequency = String.valueOf(Frequency + " MHz");
    } else
    WiFiFrequency = "The SDK unspported";
    return WiFiFrequency;
}
```

Figure 70 - getFrequency ( ) method code

8.2.2.8 getBSSID ( )

BSSID is the hardware address of the Access Point and is displayed to the user with the aid of **getConnectionInfo ( )** method included in WifiManager API and **getBSSID ( )** method of WifiInfo API. The reading of the getBSSID ( ) method is assigned to the String variable **BSSID** and the value is returned and is displayed in the textbox "**BSSID_DISPLAY**".

```
public String getBSSID() {
    WifiInfo wifiInfo = wifiManager.getConnectionInfo();
    String BSSID = wifiInfo.getBSSID();
    return BSSID;
}
```

Figure 71 - getBSSID ( ) method code

8.2.2.9 get_wifi_Speed ( )

The maximum speed achievable between the node and the host is read by calling the **getLinkSpeed ( )** method of WifiInfo API and the **getConnectionInfo ( )** method of WifiManager API. The read value is assigned to the integer variable "**Speed**" and the value is returned and displayed in the textbox "**WiFi_SPEED_DISPLAY**".

```
public int get_wifi_Speed() {
    WifiInfo wifiInfo = wifiManager.getConnectionInfo();
    int Speed = wifiInfo.getLinkSpeed();
    return Speed;
}
```

Figure 72 - get_wifi_Speed ( ) method code

8.2.2.10 get_wifi_SSID ( )

The **SSID** (Service Set Identifier) of the connected network is displayed to the connected host devices. The SSID is displayed in the "**SSID_DISPLAY"** textbox by calling **getSSID ( )** method of WifiInfo API and **getConnectionInfo ( )** method of WifiManager API.

```
public String get_wifi_SSID() {
    WifiInfo wifiInfo = wifiManager.getConnectionInfo();
    String SSID = wifiInfo.getSSID();
    return SSID;
}
```

Figure 73 - get_wifi_SSID ( ) method code

8.2.2.11 getWifiApIpAddress ( )

The IP address of the host device assigned by the node is displayed on the **SSID_DISPLAY** textbox. The system service (WIFI_SERVICE) is called by the WifiManager API and the IP address is called by the methods **getConnectionInfo ( )** and **getIpAddress ( )** under WifiManager API. The return integer value is formatted to dotted decimal format before assigning to the String variable "**ip_address**" and the value is returned.

```
public String getWifiIpAddress () {
    WifiManager wifiManager = (WifiManager)getSystemService(WIFI_SERVICE);
    String ip_address = android.text.format.Formatter.formatIpAddress(wifiManager.getConnectionInfo().getIpAddress());
    return ip_address;
}
```

Figure 74 - getWifiApIpAddress ( ) method code

8.2.3 Node Monitoring Activity

8.2.3.1 public class ClientScanResult

The ClientScanResult class consist of methods **getIP_Address ( ), getHardwareAddress ( ), getDevice ( )** and **isReachable ( )** which return the declared variables **IPAddress, HardwareAddress, Device** and **isReachable** to the java Interface collectively displaying all the available abstract methods; thus inheriting them. The values passed from the array split_point declared in the WifiApManager class are assigned to the methods in the given order.

- getIpAddress ( ) – split_point [0]
- getHardwareAddress ( ) – split_point [3]
- getDevice ( ) – split_point [5]
- isReachable ( ) – isReachable

The process of variable assignment to the split_point array are further explained in the WifiApManager class.

```
public class ClientScanResult {
    private String IPAddress;
    private String HardwareAddress;
    private String Device;
    private boolean isReachable;

    public ClientScanResult(String IP_Address, String HW_Address, String device, boolean isReachable) {
        super();
        this.IPAddress = IP_Address;
        this.HardwareAddress = HW_Address;
        this.Device = device;
        this.isReachable = isReachable;
    }

    public String getIP_Address() { return IPAddress; }

    public String getHardwareAddress() {

        return HardwareAddress;
    }

    public String getDevice() {

        return Device;
    }

    public boolean isReachable() { return isReachable; }


}
```

Figure 75 - public class ClientScanResult code

8.2.3.2 public interface FinishScanListener

The java public interface creates an **onFinishScan ( )** method with an array list in the name "**clients**" holding the ClientScanResult java method and links the NodeMonitoring, WifiApManager and ClientScanResult java classes together.

```
public interface FinishScanListener {

    public void onFinishScan(ArrayList<ClientScanResult> clients);
}
```

Figure 76 - public interface FinishScanListener code

8.2.3.3 WifiApManager class

8.2.3.3.1 getClientList ( )

The method is declared with variables "**onlyReachables**", "**reachableTimeout**" and "**finishListener**" variable referring to the FinishScanListener java interface. A new runnable has been configured under the name "**runnable**" and within the **run ( )** of the runnable, a new BufferedReader and an array list holding the ClientScanResult java class has been initialized.

Within a try block, the bufferedReader is directed to the file "**/proc/net/arp**" and the values in the file are read using a fileReader. As the file structure has the given format, an array is used to read the values in each field with the respective array position.

IP address	HW type	Mask	MAC address	Flags	Device
[0]	[1]	[2]	[3]	[4]	[5]

The read values are assigned to the string array split_point and the MAC address is assigned the array position [3] and checked for the MAC address format equals to **".. : .. : .. : .. : .. : .."** standard. The IP address is checked and assigned the array position [0] and the Boolean variable "**isReachable**" returns true if IP addresses are detected within the "**reachableTimeout**" which is configured to stop the method after 300 seconds of execution.

If reachable values are found, their details are passed to the "**ClientScanResult**" class in the format split_point [0], split_point [3], split_point [5] and isReachable.

```
public void getClientList(final boolean onlyReachables, final int reachableTimeout, final FinishScanListener finishListener) {


    Runnable runnable = () -> {

            BufferedReader bufferedReader = null;
            final ArrayList<ClientScanResult> clientScanResult = new ArrayList<>();

            try {
                bufferedReader = new BufferedReader(new FileReader("/proc/net/arp"));
                String line;
                while ((line = bufferedReader.readLine()) != null) {
                    String[] split_point = line.split(" + ");
                    if ((split_point != null) && (split_point.length >= 4)) {
                        String MAC_Address = split_point[3];
                        if (MAC_Address.matches("..:..:..:..:..:..")) {
                            boolean isReachable = InetAddress.getByName(split_point[0]).isReachable(reachableTimeout);
                            if (!onlyReachables || isReachable) {
                                clientScanResult.add(new ClientScanResult(split_point[0], split_point[3], split_point[5], isReachable));
                            }
                        }
                    }
                }
```

Figure 77 - getClientList ( ) code

8.2.3.4 Node monitoring.

8.2.3.4.1 display_active_clients ( )

The method **display_active_clients ( )** refers to the method "**getClientList**" of the WifiApManager class and displays the values within the array list ClientScanResult within textboxes. The textboxes are called within **append ( )** method. This procedure is followed to maximize the program efficiency and reduce the load on the system. The host details found within the "**/proc/net/arp**" file are assigned to the array list "**clientScanResult**" and displayed within the appended textboxes.

The "**IP address**" field is assigned the value in the method **getIP_Address ( )** method.

The "**hardware type**" field is assigned the value in the method **getDevice ( )** method.

The "**MAC Address**" field is assigned the value in the **getHardwareAddress ( )** method.

The "**Reachability status**" field is assigned the value of **isReachable ( )** method.

"**Append**" textbox types are used over "**setText**" text boxes as the return variables determine the number of textboxes (one return variable per textbox).

```
    private void display_active_clients() {
        wifiApManager.getClientList(false, (clients) -> {
                TextView Node_Monitoring_TextView = (TextView) findViewById(R.id.Node_Monitoring_TextView);
                Node_Monitoring_TextView.append("Connected clients: \n");
                for (ClientScanResult clientScanResult : clients) {
                    Node_Monitoring_TextView.append("-------------------------------------\n");
                    Node_Monitoring_TextView.append("IP address : " + clientScanResult.getIP_Address() + "\n");
                    Node_Monitoring_TextView.append("Hardware type : " + clientScanResult.getDevice() + "\n");
                    Node_Monitoring_TextView.append("MAC Address : " + clientScanResult.getHardwareAddress() + "\n");
                    Node_Monitoring_TextView.append("Reachability Status : " + clientScanResult.isReachable() + "\n");
                    }
        });
    }
}
```

Figure 78 - display_active_clients ( ) method code

8.2.4 Power Consumption

8.2.4.1 Button_Device_Info_Initiate ( )

The button "**Button_Device_Info**" is assigned a **SetOnClickListener** method and an Intent is declared within the OnClickListener method redirecting the button click to the Device_Info java class. The activity "**Device_Info**" is started by calling the method **startActivity ( ).**

```
public void Button_Device_Info_Initiate() {
    Button_Device_Info = (Button)findViewById(R.id.Button_Device_Info);
    Button_Device_Info.setOnClickListener((v) -> {
            Intent Device_Info = new Intent(PowerConsumption.this, com.network.ad_hoc.fidelity_fi.ResourceMonitoring.Device_Info.class);
            startActivity(Device_Info);
    });
}
```

Figure 79 - Button_Device_Info_Initiate ( ) method code

8.2.4.1.1 getDeviceInfo ( )

The method includes a try block which read the OS version from "**System.getproperty ("os.version")** and assigned to the String variable "**Display**"; the other methods read the values by extracting from the system properties "**android.os.Build**". The method display the API level, the device code, model, device product, the OS version release, device brand, device build ID, device manufacturer and the serial number of the device. Each read variable is assigned to the String variable "**Display**" and passed to the textbox "**Device_info_View**" declared within a scrollable. The scrollable allows the user to scroll down and view the values. A catch block is declared with an exception and displays the message "**Error getting device information**" if the "**android.os.Build**" system properties return a null value.

```
    private String getDeviceInfo() {
        try {
            Display += "\n OS Version:"+ System.getProperty("os.version") +"\n";
            Display += "\n OS API Level:"+ android.os.Build.VERSION.SDK_INT +"\n";
            Display += "\n Device:"+ android.os.Build.DEVICE +"\n";
            Display += "\n Model:" + android.os.Build.MODEL +"\n";
            Display += "\n Product:"+ android.os.Build.PRODUCT +"\n";
            Display += "\n Release:"+ android.os.Build.VERSION.RELEASE +"\n";
            Display += "\n Brand:"+ android.os.Build.BRAND +"\n";
            Display += "\n Build ID:"+ android.os.Build.ID +"\n";
            Display += "\n Manufacturer:"+ android.os.Build.MANUFACTURER +"\n";
            Display += "\n Serial:"+ android.os.Build.SERIAL +"\n";
        } catch (Exception e) {
         Display = "Error getting device information";
        }
return Display;
    }
```

Figure 80 - getDeviceInfo ( ) method code

8.2.4.2 Button_CPU_Utilization_Initiate ( )

The button "**Button_CPU_Utilization**" is assigned a SetOnClickListener and an Intent declared within the OnClickListener method programmed to redirect the user to the "**CPU_Info**" activity by initiating the **startActivity ( )** method.

```
public void Button_CPU_Utilization_Initiate() {
    Button_CPU_Utilization = (Button)findViewById(R.id.Button_CPU_Utilization);
    Button_CPU_Utilization.setOnClickListener((v) -> {
            Intent CPU_Info = new Intent(PowerConsumption.this, com.network.ad_hoc.fidelity_fi.ResourceMonitoring.CPU_Info.class);
            startActivity(CPU_Info);
    });
}
```

Figure 81 - Button_CPU_Utilization_Initiate ( ) method code

8.2.4.2.1 getCPU_Info ( )

The method reads the CPU details of the device by checking the system for the file "**/proc/cpuinfo**". If the file exists in the device, a file reader is called within the buffered reader and the values in the "/proc/cpuinfo" file are read and added to the buffer. The String variable "**ReadLine**" is assigned the values added to the buffered reader by the appending process. The buffered reader is closed at the end of the activity by calling the method **bufferedReader.close (**

) and the return value is converted to a string variable using "**.toString ( )"** method. If the "/proc/cpuinfo" read returns false, the InputOutputException is called within the catch block.

```
private String getCPU_Info() {
    StringBuffer stringBuffer = new StringBuffer();
    if (new File("/proc/cpuinfo").exists()) {
        try {
            BufferedReader bufferedReader = new BufferedReader(new FileReader(new File("/proc/cpuinfo")));
            String ReadLine;
            while ((ReadLine = bufferedReader.readLine()) != null) {
                stringBuffer.append(ReadLine + "\n");
            }
            if (bufferedReader != null) {
                bufferedReader.close();
            }
        } catch (IOException e) {
        }
    }
    return stringBuffer.toString();
}
```

Figure 82 - getCPU_Info ( ) method code

8.2.4.3 Battery_Info_Button_Initiate ( )

"**Button_RAM_Utilization_Initiate**" is assigned a **SetOnClickListener** and an Intent declared within the OnClickListener method redirects the user to the "**Battery_Info**" activity by initiating the **startActivity ( )** method.

```
public void Battery_Info_Button_Initiate() {
    Button_Battery_Info = (Button)findViewById(R.id.Button_Battery_Info);
    Button_Battery_Info.setOnClickListener((v) -> {
            Intent Battery_Info = new Intent(PowerConsumption.this, com.network.ad_hoc.fidelity_fi.ResourceMonitoring.Battery_Info.class);
            startActivity(Battery_Info);
    });
}
```

Figure 83 - Battery_Info_Button_Initiate ( ) method code

8.2.4.3.1 Battery_Info ( )

The battery details are called with the aid of "**BatteryInfo**" BroadcastReceiver registered in the **onCreate ( )** method using registerReceiver and an IntentFilter declared with Intent **ACTION_BATTERY_CHANGED**. The different functions are called within the same BroadcastReceiver to reduce the load on the system.

Remaining battery capacity – calculated by calling the "**value**" variable declared in **getIntExtra ( )** method with the intent "**ACTION_BATTERY_CHANGED**" and assigned to the integer variable "**level**". The battery level variable is assigned to a progress bar and the integer value is displayed as a percentage.

```
int level = intent.getIntExtra("level", 0);
ProgressBar progressBar_Remaining_Capacity = (ProgressBar) findViewById(R.id.progressBar_Remaining_Capacity);
progressBar_Remaining_Capacity.setProgress(level);
TextView Battery_Percentage_textView = (TextView) findViewById(R.id.Battery_Percentage_textView);
Battery_Percentage_textView.setText(Integer.toString(level) + "%");
```

Figure 84 - Battery_Info ( ) method code

Voltage – the current remaining battery voltage is calculated by calling the "**voltage**" variable within the **ACTION_BATTERY_CHANGED** intent. The return value is converted to a String data type and is displayed to the user in "**millivolts (mV)**".

```
TextView Voltage_Display = (TextView) findViewById(R.id.Voltage_Display);
if (intent.getAction().equals(Intent.ACTION_BATTERY_CHANGED)) {
    Voltage_Display.setText(String.valueOf(intent.getIntExtra("voltage", 0) + "mV"));
}
```

Figure 85 - Voltage display method code

Plugged in source – the method returns the device's plugged in source together with **ACTION_BATTERY_CHANGED** and intent **EXTRA_PLUGGED** of BatteryManager API. The method reads the BatteryManager API state (USB, AC and WIRELESS) and display the value accordingly.

```
TextView Plugged_in_Source = (TextView) findViewById(R.id.Plugged_in_Source);
if (intent.getAction().equals(Intent.ACTION_BATTERY_CHANGED)) {
    int source = intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
    String Source;
    switch (source) {
        case BatteryManager.BATTERY_PLUGGED_USB:
            Source = "USB";
            break;
        case BatteryManager.BATTERY_PLUGGED_AC:
            Source = "AC";
            break;
        case BatteryManager.BATTERY_PLUGGED_WIRELESS:
            Source = "Wireless";
            break;
        default:
            Source = "Unknown";
    }
    Plugged_in_Source.setText(Source);
}
```

Figure 86 - Plugged-in-Source method code

Temperature – the battery temperature is calculated by calling the method **EXTRA_TEMPERATURE** of BatteryManager API and **ACTION_BATTERY_CHANGED** intent. The variable "**Temperature**" is read and displayed to the user in degrees Celsius (°C).

```
TextView Operating_Temperature = (TextView) findViewById(R.id.Operating_Temperature);
if (intent.getAction().equals(Intent.ACTION_BATTERY_CHANGED)) ;
{
    int Temperature = intent.getIntExtra(BatteryManager.EXTRA_TEMPERATURE, 0);
    Operating_Temperature.setText(String.valueOf(intent.getIntExtra("Temperature", Temperature / 10) + "C"));
}
```

Figure 87 - Temperature method code

Battery health – the battery health is calculated by calling the method **BATTERY_HEALTH_UNKNOWN** of BatteryManager API and **ACTION_BATTERY_CHANGED** intent. The method reads the battery conditions (GOOD, DEAD, COLD, OVER_VOLTAGE and OVERHEAT) and returns the value accordingly.

```
TextView Battery_Health = (TextView) findViewById(R.id.Battery_Health);
if (intent.getAction().equals(Intent.ACTION_BATTERY_CHANGED)) ;
{
    int health = intent.getIntExtra("health", BatteryManager.BATTERY_HEALTH_UNKNOWN);
    String Health;
    if (health == BatteryManager.BATTERY_HEALTH_GOOD) {
        Health = "Good";
    } else if (health == BatteryManager.BATTERY_HEALTH_DEAD) {
        Health = "Dead";
    } else if (health == BatteryManager.BATTERY_HEALTH_COLD) {
        Health = "Cold";
    } else if (health == BatteryManager.BATTERY_HEALTH_OVER_VOLTAGE) {
        Health = "Over Voltage";
    } else if (health == BatteryManager.BATTERY_HEALTH_OVERHEAT) {
        Health = "Over Heat";
    } else {
        Health = "Unknown";
    }
    Battery_Health.setText(Health);
}
```

Figure 88 - Battery Health method code

Battery technology – the battery cell technology is obtained by calling **EXTRA_TECHNOLOGY** method of BatteryManager API and intent **ACTION_BATTERY_CHANGED**. The method returns the battery cell technology in String variable data type.

```
TextView Battery_Technology = (TextView) findViewById(R.id.Battery_Technology);
if (intent.getAction().equals(Intent.ACTION_BATTERY_CHANGED)) ;
{
    String technology = intent.getExtras().getString(BatteryManager.EXTRA_TECHNOLOGY);
    Battery_Technology.setText(technology);
}
```

Figure 89 - "Battery Technology" method code

Power saving mode check – the **isPowerSaveMode ( )** method is called from the PowerManager API and Context **POWER_SERVICE**. If the method returns true, the user is notified the respective state of the power saving mode. The **isPowerSaveMode ( )** method supports from API level 21 and upwards. Therefore a version check is carried out before executing the method. If the version check returns true, the method is executed. Else, the user is notified the device SDK doesn't support the function.

```
PowerManager powerManager = (PowerManager) context.getSystemService(Context.POWER_SERVICE);
TextView Power_Saving_Mode_TextView = (TextView)findViewById(R.id.Power_Saving_Mode_TextView);
if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) {
    if (powerManager.isPowerSaveMode() == true) {
        Power_Saving_Mode_TextView.setText("Enabled");
    } else {
        Power_Saving_Mode_TextView.setText("Disabled");
    }
}

else {
    Power_Saving_Mode_TextView.setText("SDK Unsupported");
}
```

Figure 90 - "Power Saving mode" method code

8.2.4.4 Button_RAM_Utilization_Initiate ( )

The button "**Button_RAM_Utilization_Initiate**" is assigned a **SetOnClickListener** and an Intent declared within the OnClickListener method programmed to redirect to the "**RAM_Info**" activity by initiating the **startActivity ( )** method.

```
public void Button_RAM_Utilization_Initiate() {
    Button_RAM_Utilization = (Button)findViewById(R.id.Button_RAM_Utilization);
    Button_RAM_Utilization.setOnClickListener((v) -> {
            Intent RAM_Info = new Intent(PowerConsumption.this, com.network.ad_hoc.fidelity_fi.ResourceMonitoring.RAM_Info.class);
            startActivity(RAM_Info);
    });

}
```

Figure 91 - Button_RAM_Utilization_Initiate ( ) method code

8.2.4.4.1 getTotalRAM ( )

The system service "**ACTIVITY_SERVICE**" and **MemoryInfo ( )** methods in ActivityManager API are utilized to calculate the RAM used by the device. The method "**MemoryInfo.totalMem**" is called within a try block and assigned to the variable "**totalMemory**" declared under long data type.

As the memory used by applications are returned in byte format, the variable "**freeMemory**" containing the return value is divided by 1000 to convert the value to kilobytes (kB) and as the variables are displayed in megabytes (MB), the kB calculation is again divided by 1024. The exception, (Exception e) is caught within the catch block and the final value is returned to the main

method for display within the textbox "**TOTAL_RAM_Available**" and the return values are limited to 2 decimal places using the function "%.2f"

```
public String getTotalRAM () {
    ActivityManager activityManager = (ActivityManager)getSystemService(ACTIVITY_SERVICE);
    ActivityManager.MemoryInfo memoryInfo = new ActivityManager.MemoryInfo();
    activityManager.getMemoryInfo(memoryInfo);
    long totalMemory = 0;

    try {
        totalMemory= memoryInfo.totalMem;
    } catch (Exception e) {
    }
    double kB = totalMemory/1000;
    double MB = kB/ (1024.0);
    String output;
    output = String.format("%.2f",MB);
    Display = output + " MB";
    return Display;
}
```

Figure 92 - getTotalRAM ( ) method code

8.2.4.4.2 availableMemory ( )

The system service "**ACTIVITY_SERVICE"** and **MemoryInfo ( )** methods in ActivityManager API are utilized to calculate the RAM used by the device. The method "**MemoryInfo.availMem**" is called within a try block and assigned to the variable "**freeMemory**" declared under long data type.

As the total memory available are returned in byte format, the variable "**freeMemory**" containing the return value is divided by 1000 to convert the value to kilobytes (kB) and as the variables are displayed in megabytes (MB), the kB calculation is again divided by 1024. The exception (Exception e) is caught within the catch block and the final value is returned to the main method for display within the textbox "**Application_Free_Memory**" and the return values are limited to 2 decimal places using the function "%.2f"

```
public String availableMemory() {
    ActivityManager activityManager = (ActivityManager)getSystemService(ACTIVITY_SERVICE);
    ActivityManager.MemoryInfo memoryInfo = new ActivityManager.MemoryInfo();
    activityManager.getMemoryInfo(memoryInfo);
    long freeMemory = 0;

    try {
        freeMemory = memoryInfo.availMem;
    } catch (Exception e) {
    }
    double kB = freeMemory/1000;
    double MB = kB/1024;
    String output;
    output = String.format("%.2f",MB);
    Display = output + " MB";
    return Display;
}
```

Figure 93 - availableMemory ( ) method code

8.2.4.4.3 applicationUsedMemory ( )

The application used memory is calculated by subtracting the free memory from the total available memory, rather than calculating the memory used by each application. The system service "**ACTIVITY_SERVICE**" and **MemoryInfo ( )** methods in ActivityManager API are utilized to calculate the RAM used by the device. The methods "**MemoryInfo.availMem**" and "MemoryInfo.totalMem" are called within a try block and the difference by subtracting the available memory from total memory will give the application used memory and the result is assigned to the variable "**usedMemory**" declared under long data type.

As the memory details are returned in byte format, the variable "**used Memory**" containing the return value is divided by 1000 to convert the value to kilobytes (kB) and as the variables are displayed in megabytes (MB), the kB calculation is again divided by 1024. The exception (Exception e) is caught within the catch block and the final value is returned to the main method for display within the textbox "**Application_Used_Memory**" and the return values are limited to 2 decimal places using the function "%.2f"

```
public String applicationUsedMemory() {

    ActivityManager activityManager = (ActivityManager)getSystemService(ACTIVITY_SERVICE);
    ActivityManager.MemoryInfo memoryInfo = new ActivityManager.MemoryInfo();
    activityManager.getMemoryInfo(memoryInfo);

    long usedMemory = 0;
    try {
        long freeMemory = memoryInfo.availMem;
        long totalMemory = memoryInfo.totalMem;
        usedMemory = totalMemory-freeMemory;
    } catch (Exception e) {
    }
    double kB = usedMemory/1000.0;
    double MB = kB / (1024.0);
    String output;
    output = String.format("%.2f",MB);
    Display = output + " MB";
    return Display;
}
```

Figure 94 - applicationUsedMemory ( ) method code

8.3 Android studio features used

8.3.1 Debugger

The functionality of the application can be tested by debugging the methods and functions on the Android Emulator or on a connected android device. The following features of the android debugger are used to test the implemented system.

- Device selection for debugging – application is installed and debugged on a variety of devise running on different android versions.
- Examination of expressions and methods evaluation at runtime. (Developer.android.com, 2016)

8.3.2 Android emulator

Android emulator allows to prototype test and develop android applications without the necessity of a hardware device running on android platform. The emulator uses AVD configurations to determine the functionality of the simulated device and each configured AVD functions as an independent device and runs on a full Android system stack including the kernel level. Android

emulator requires Android Studio 2.0 or higher and SDK tools 25.0.10 for activity. However, the android emulator doesn't support network related implementations including Wi-Fi, NFC and Bluetooth. Only the GUI of the implemented application is tested on the Android emulator as the implemented system is related to network status reading and modification. (Developer.android.com, 2016)

8.3.3 Android build system

The application resources, source codes and packages are compiled into APK files by the android build system which are deployable on android emulator/ hardware device. With the Gradle feature in-built in android studio, the build process is automated and managed while allowing the user to define and configure custom build configurations. The Build process involves multiple processes and tools converting the project into an android APK. The build process involves the following procedure putting up multiple components together. (Developer.android.com, 2016)

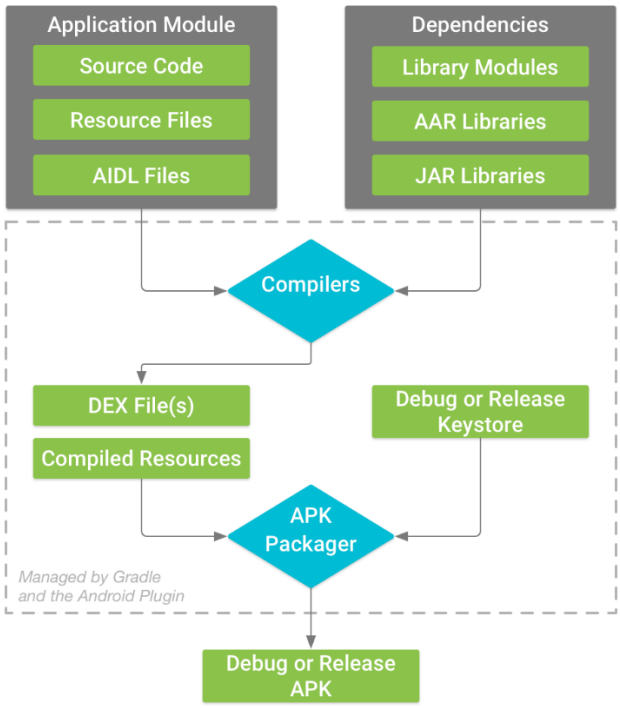

Figure 95 - android Gradle structure *(Developer.android.com, 2016)*

CHAPTER 9 – TESTING & PERFORMANCE EVAALUATION

To test the implemented application, several test procedures and methods are utilized. As the application development is done on android platform, the Android Studio built-in features are used to test the application. The testing includes JUnit testing running on the local JVM and instrumented testing procedure running on the test device and producing results accordingly.

9.1 White-box testing

The White-box testing procedure is carried out to check whether the implemented system meet the required criteria, where the unit functionality, performance, quality and accuracy of the output is monitored and recorded. White-box testing is carried out within the android studio itself by testing the essential methods and activities required for the application functionality.

9.1.1 Main menu activity

Test 01

- **Input** - onCreate ( ) method of the main menu activity
- **Result** –method passed with a total time of 2 milliseconds.

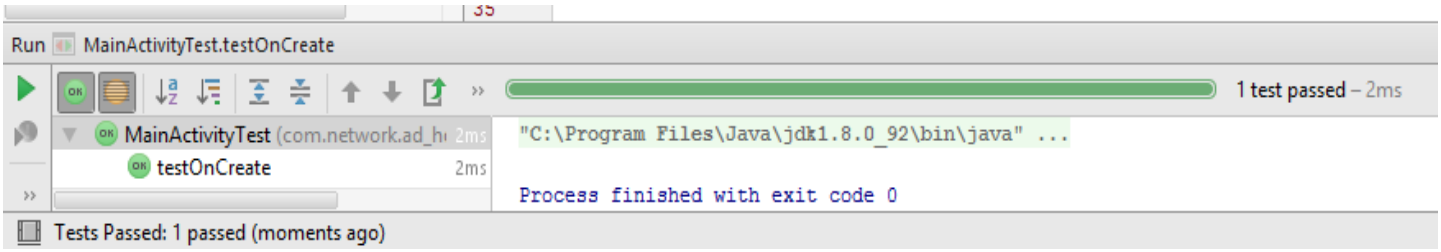

Figure 96 - Main menu activity onCreate ( ) method test results

Test 02

- **Input** - SignalStrengthButtonInitiate ( ) method
- **Result** – test passed with a total runtime of 1 millisecond.

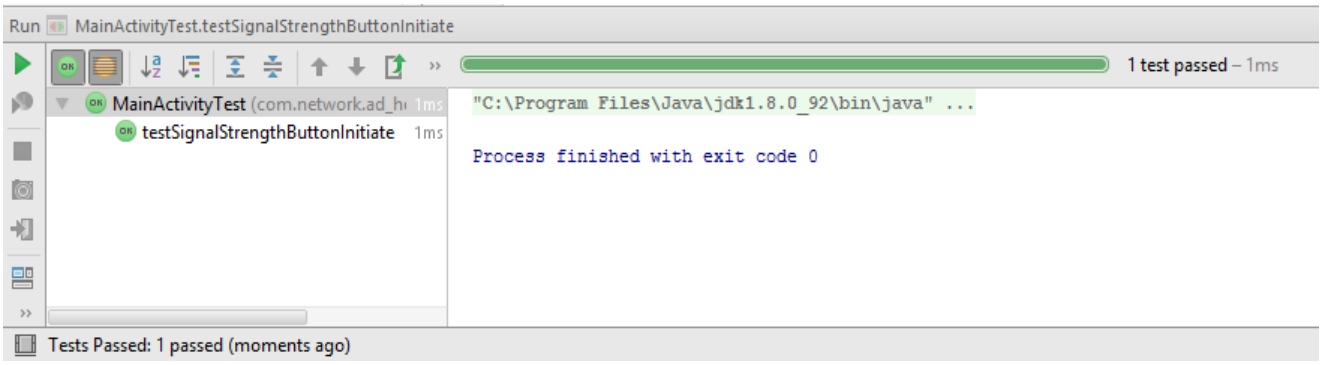

Figure 97 - Main menu activity SignalStrengthButtonInitiate ( ) method test results

Test 03

- **Input** - SignalStrengthButtonInitiate ( ) method
- **Result** – test passed with a total runtime of 1 millisecond.

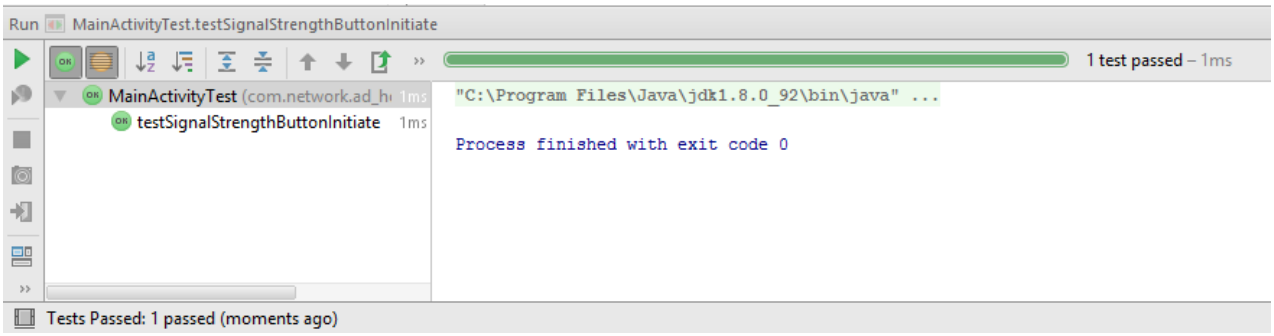

Figure 98 - Main menu activity SignalStrengthButtonInitiate ( ) method test results

Test 04

- **Input** – PowerConsumptionButtonInitiate ( ) method
- **Result** – test passed with a total runtime of 2 milliseconds

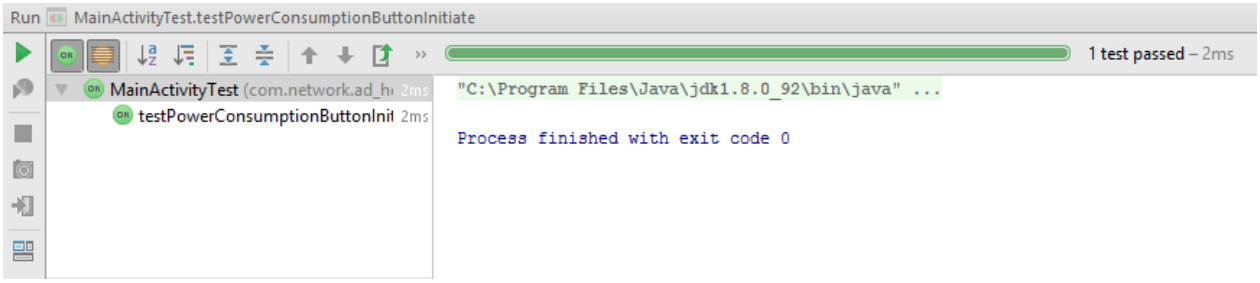

Figure 99 - Main menu activity PowerConsumptionButtonInitiate ( ) method test results

Test 05

- **Input** - OnCreateOptionsMenu ( ) method
- **Result** – test passed with a total runtime of 2 milliseconds.

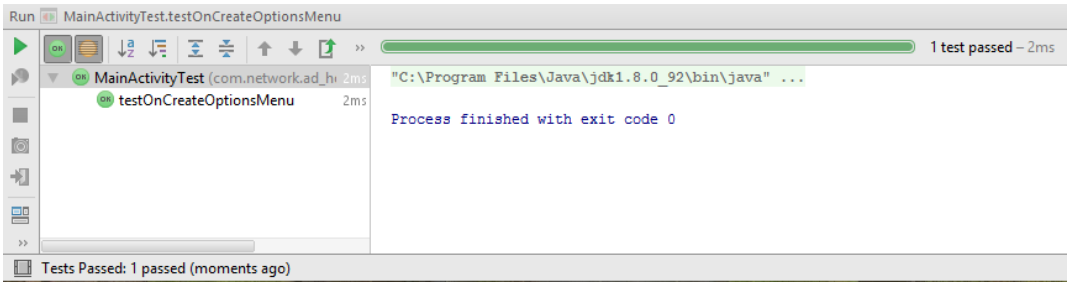

Figure 100 - Main menu activity OnCreateOptionsMenu ( ) method test results

Test 06

- **Input** - OnOptionsItemSelected ( ) method
- **Result** - test passed with a total runtime of 1 millisecond.

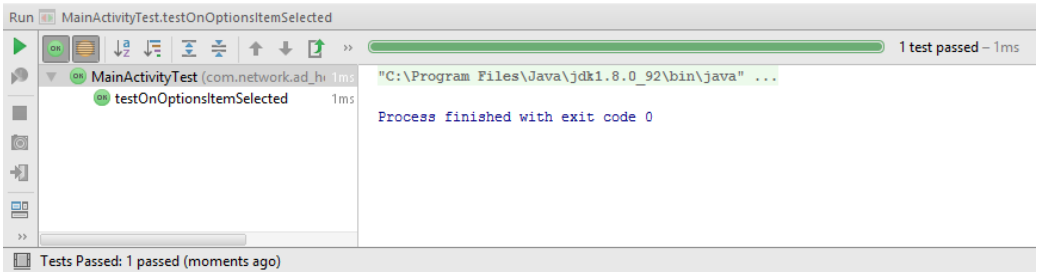

Figure 101 - Main menu activity OnOptionsItemSelected ( ) method test results

Test 07

- **Input** - OnBackPressed ( ) method
- **Result** - test passed with a total runtime of 2 milliseconds.

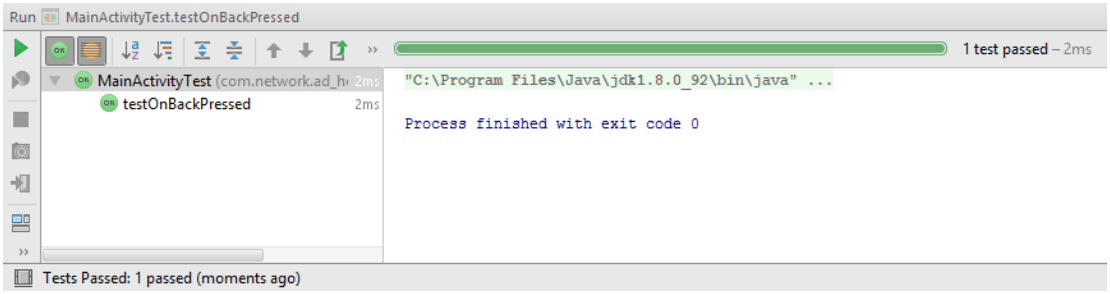

Figure 102 - Main menu activity OnBackPressed ( ) method test results

Test 08

- **Input** - TotalUploadDownload ( ) method
- **Result** - test passed with a total runtime of 2 milliseconds.

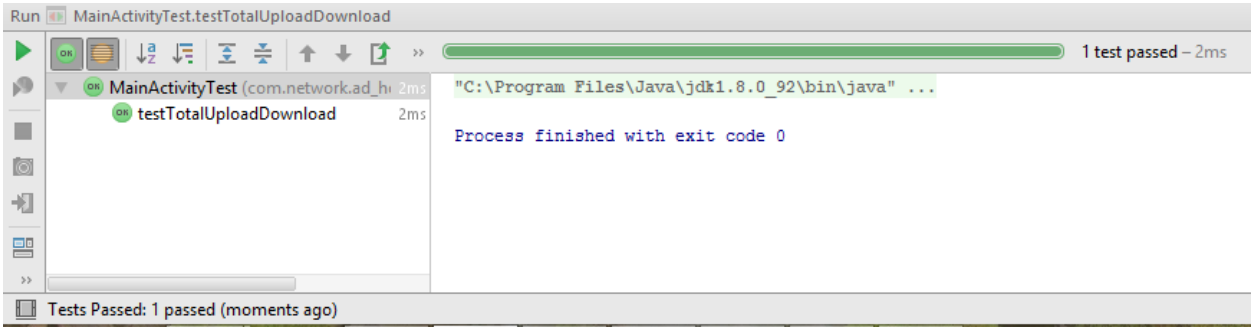

Figure 103 - Main menu activity TotalUploadDownload ( ) method test results

Test 09

- **Input** - UploadDownloadRates_Main ( ) method
- **Result** - test passed with a total runtime of 3 milliseconds.

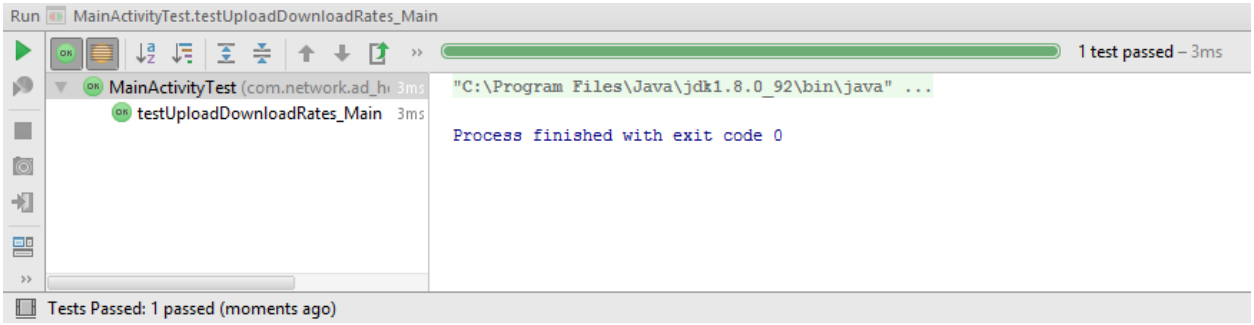

Figure 104 - Main menu activity UploadDownloadRates_Main ( ) method test results

Test 10

- **Input** - UploadDownloadPackets ( ) method
- **Result** - test passed with a total runtime of 2 milliseconds.

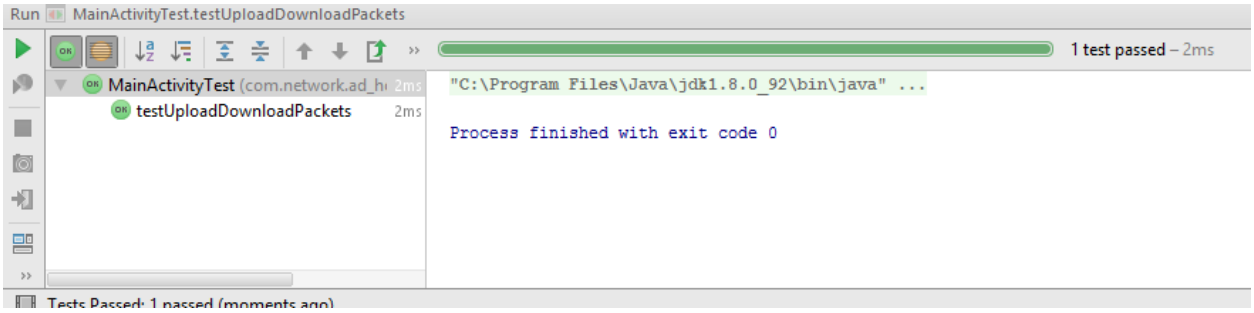

Figure 105 - Main menu activity UploadDownloadPackets ( ) method test results

Test 11

- **Input** - SetMobileDataOn ( ) method
- **Result** - test passed with a total runtime of 5 milliseconds.

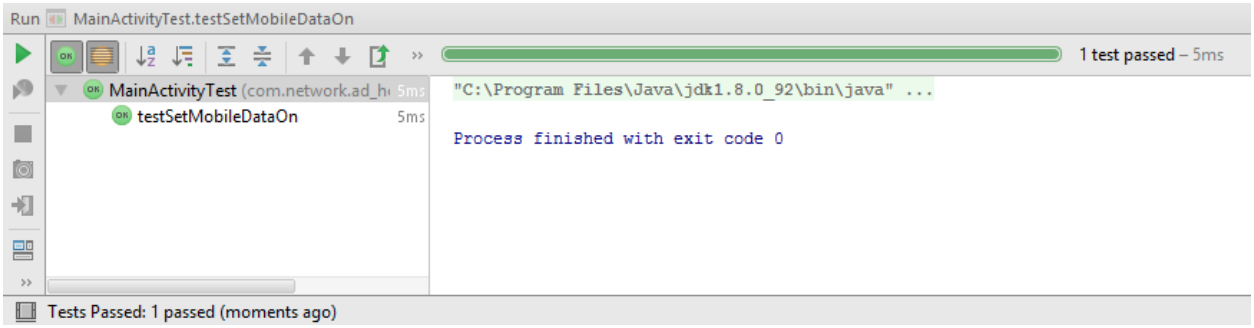

Figure 106 - Main menu activity SetMobileDataOn ( ) method test results

Test 12

- **Input** - SetMobileDataOff ( ) method
- **Result** - test passed with a total runtime of 2 milliseconds.

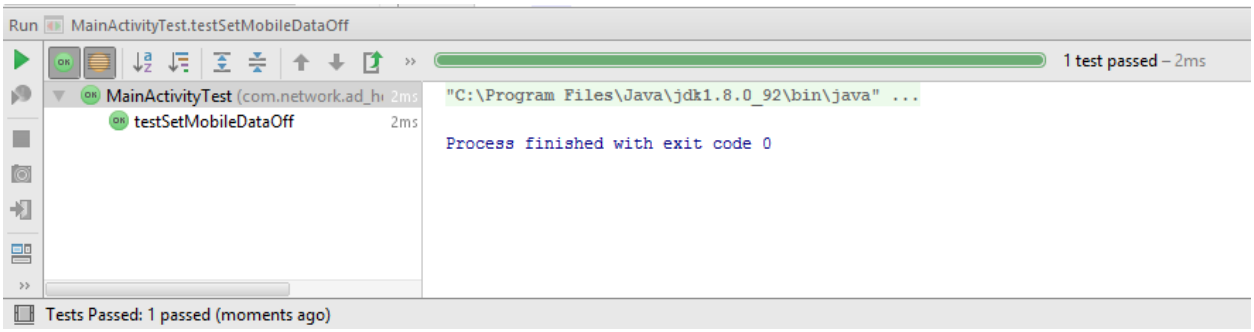

Figure 107 - Main menu activity SetMobileDataOff ( ) method test results

Test 13

- **Input** - AutoPilotMode ( ) method
- **Result** - test passed with a total runtime of 2 milliseconds.

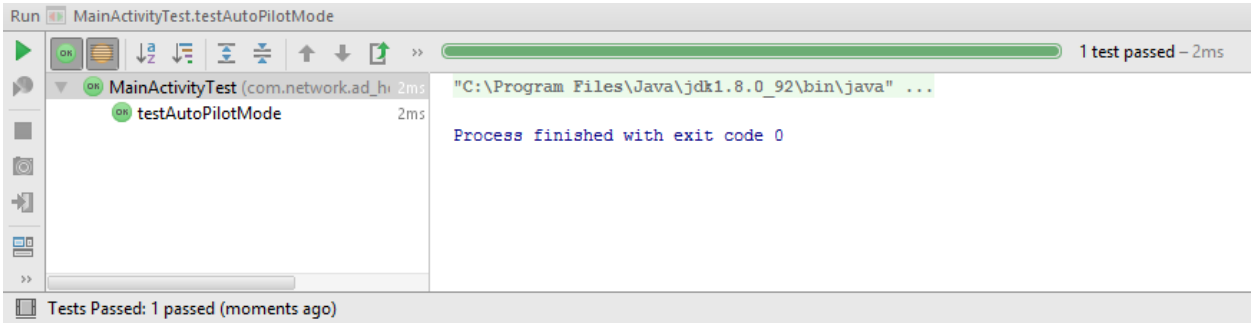

Figure 108 - Main menu activity AutoPilotMode ( ) method test results

Test 14

- **Input** - IsRooted ( ) method
- **Result** - test passed with a total runtime of 1 milliseconds.

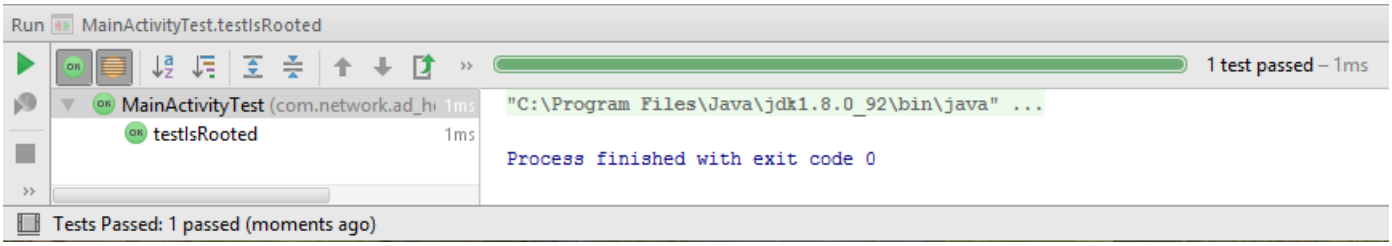

Figure 109 - Main menu activity isRooted ( ) method test results

9.1.2 Signal Strength activity

Test 15

- **Input** - onCreate ( ) method
- **Result** - test passed with a total runtime of 2 milliseconds.

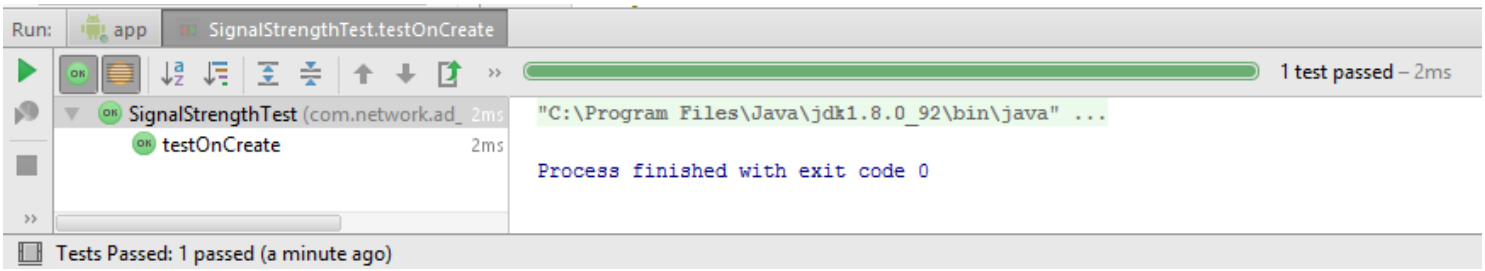

Figure 110 - Signal Strength Activity onCreate ( ) method test results

Test 16

- **Input** - GetWiFiMACAddress ( ) method
- **Result** - test passed with a total runtime of 1 millisecond.

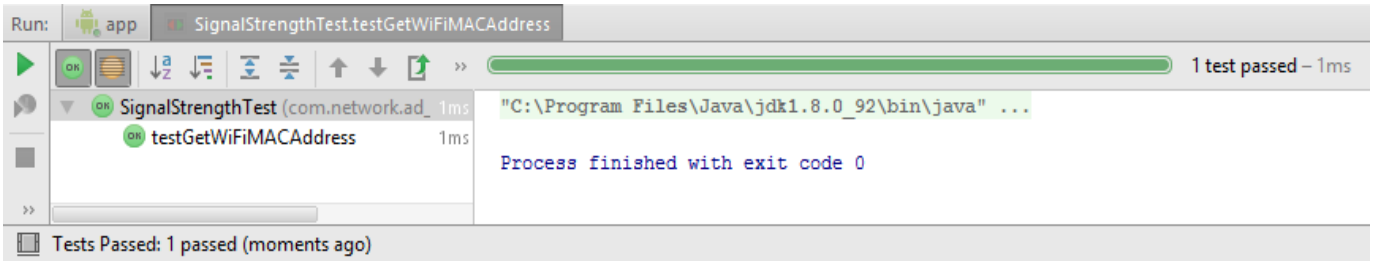

Figure 111 - Signal Strength Activity GetWiFiMACAddress ( ) method test results

Test 17

- **Input** - GetFrequency ( ) method
- **Result** - test passed with a total runtime of 1 millisecond.

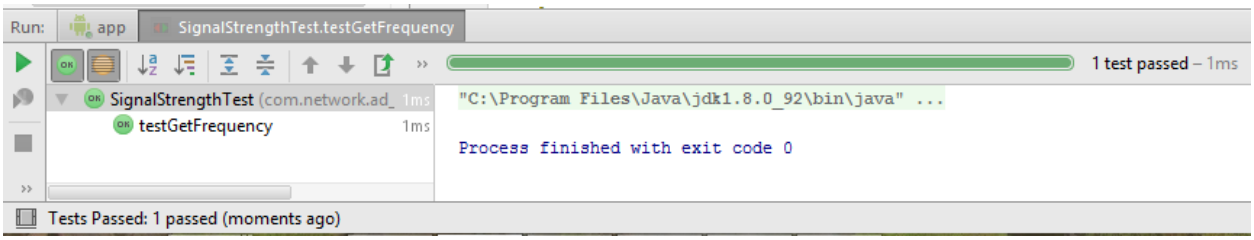

Figure 112 - Signal Strength Activity GetFrequency ( ) method test results

Test 18

- **Input** - GetBSSID ( ) method
- **Result** - test passed with a total runtime of 1 millisecond.

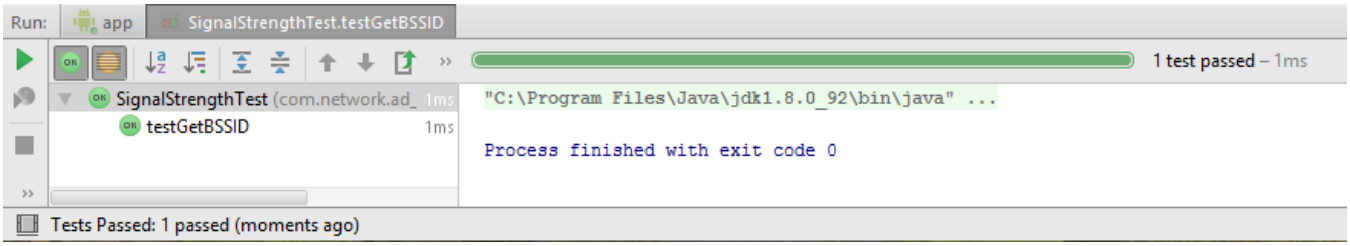

Figure 113 - Signal Strength Activity GetBSSID ( ) method test results

Test 19

- **Input** - Get_wifi_Speed ( ) method
- **Result** - test passed with a total runtime of 2 milliseconds.

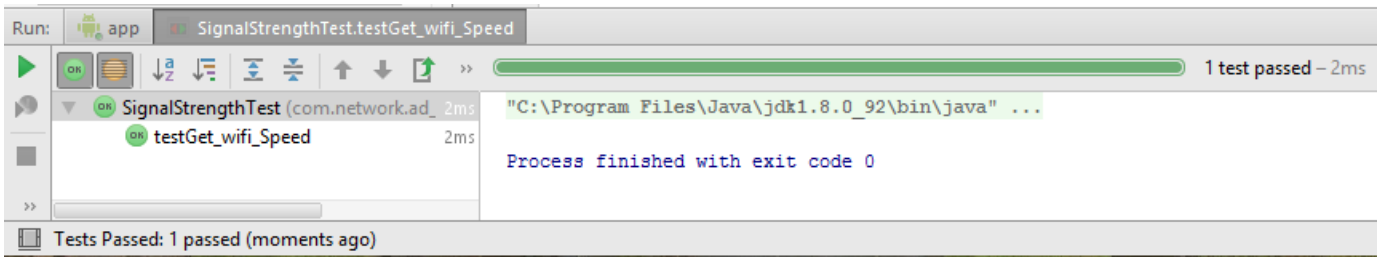

Figure 114 - Signal Strength Activity Get_wifi_Speed ( ) method test results

Test 20

- **Input** - Get_wifi_SSID ( ) method
- **Result** - test passed with a total runtime of 2 milliseconds.

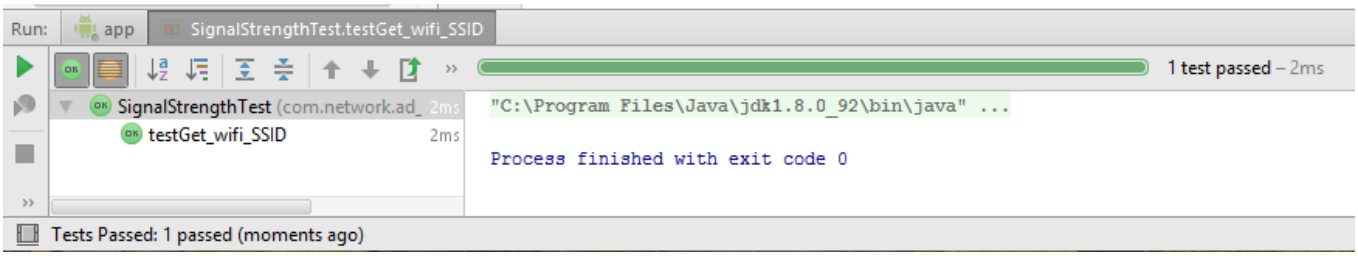

Figure 115 - Signal Strength Activity Get_wifi_SSID ( ) method test results

Test 21

- **Input** - GetWifiIpAddress ( ) method
- **Result** - test passed with a total runtime of 7 milliseconds.

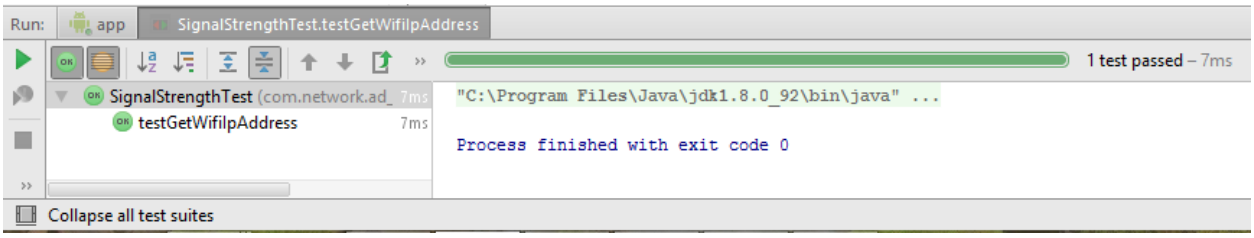

Figure 116 - Signal Strength Activity GetWifiIpAddress ( ) method test results

9.1.3 Power Consumption activity

Test 22

- **Input** - onCreate ( ) method
- **Result** - test passed with a total runtime of 1 milliseconds.

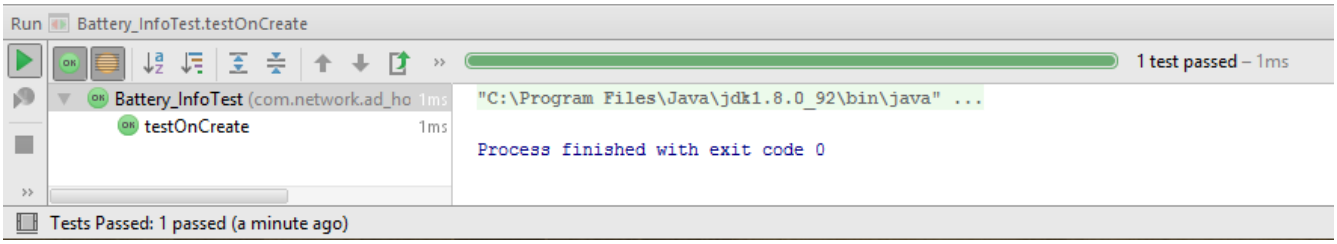

Figure 117 – Power Consumption Activity onCreate ( ) method test results

9.1.4 Functional testing

Test 23 – Power consumption activity

- Input - Battery_Info class
- Result - test passed with a total runtime of 2 milliseconds.

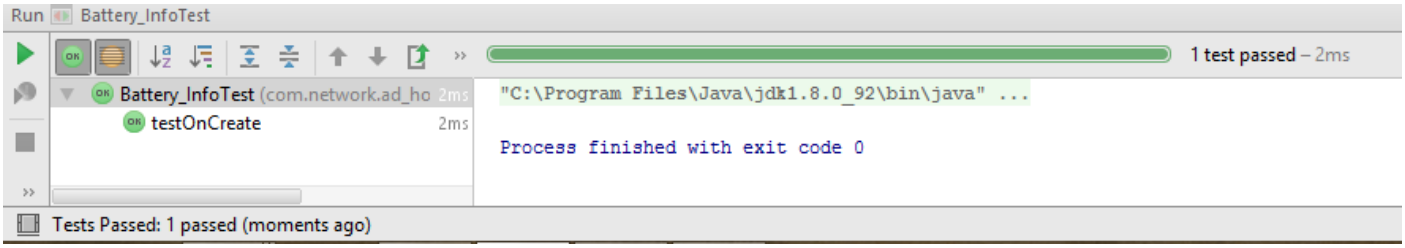

Figure 118 - Functional Testing Power Consumption Activity methods test results

Test 24 – Signal Strength activity

- **Input** - SignalStrength class
- **Result** – 7 tests passed with a total runtime of 1 millisecond(s).

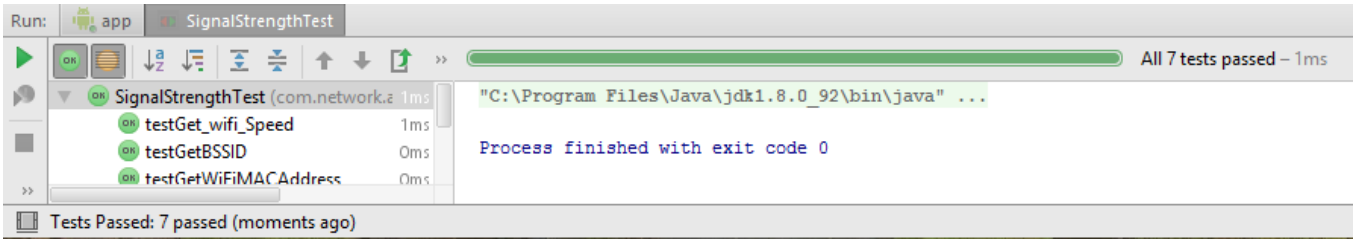

Figure 119 - Functional Testing Signal Strength Activity methods test results

Test 25 – Main menu activity

- **Input** - MainActivity class
- **Result** - test passed with a total runtime of 3 milliseconds.

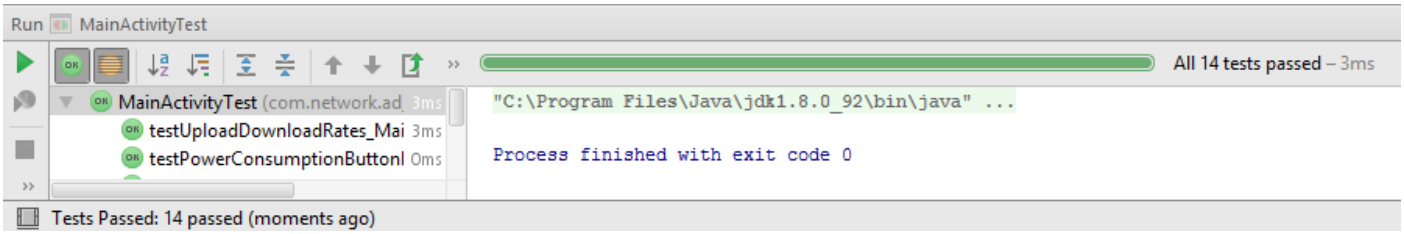

Figure 120 - Functional Testing Main menu Activity methods test results

9.1.5 Unit Test Summary

9.1.5.1 Main Menu Activity

Test number	Description	Test status	Total time
01	onCreate ( ) method	Pass	2 millisecond(s)
02	SignalStrengthButtonInitiate ( ) method	Pass	1 millisecond(s)
03	NodeMonitoringButtonInitiate ( ) method	Pass	1 millisecond(s)
04	PowerConsumptionButtonInitiate ( )	Pass	2 millisecond(s)
05	OnCreateOptionsMenu ( )	Pass	2 millisecond(s)
06	OnOptionsItemSelected ( )	Pass	1 millisecond(s)
07	OnBackPressed ( )	Pass	2 millisecond(s)
08	TotalUploadDownload ( )	Pass	2 millisecond(s)
09	UploadDownloadRates_Main ( ) method	Pass	3 millisecond(s)
10	UploadDownloadPackets ( ) method	Pass	2 millisecond(s)
11	SetMobileDataOn ( ) method	Pass	5 millisecond(s)
12	SetMobileDataOff ( ) method	Pass	2 millisecond(s)
13	AutoPilotMode ( ) method	Pass	2 millisecond(s)
14	IsRooted ( ) method	Pass	1 millisecond(s)

Table 5 - Main menu activity unit test summary

9.1.5.2 Signal Strength activity

Test number	Description	Test status	Total time
01	onCreate ( ) method	Pass	2 millisecond(s)
02	GetWiFiMACAddress ( ) method	Pass	1 millisecond(s)
03	GetFrequency ( ) method	Pass	1 millisecond(s)
04	GetBSSID ( ) method	Pass	1 millisecond(s)
05	Get_wifi_Speed ( ) method	Pass	2 millisecond(s)
06	Get_wifi_SSID ( ) method	Pass	2 millisecond(s)
07	GetWifiIpAddress ( ) method	Pass	7 millisecond(s)

Table 6 - Signal Strength activity unit test summary

9.1.5.3 Power Consumption activity

Test number	Description	Test status	Total time
01	onCreate ( ) method	Pass	1 millisecond(s)

Table 7 - Power Consumption activity unit test summary

9.1.6 Functional test summary

Test number	Description	Test status	Total time
23	Power Consumption activity	Pass	2 millisecond(s)
24	Signal Strength activity	Pass	1 millisecond(s)
25	MainActivity functional testing	Pass	3 millisecond(s)

Table 8 - Functional test summary

Considering the units tested in the Main menu activity, to execute each method, a total elapsed CPU time of 28 millisecond(s) is utilized. For the Signal Strength activity, a total CPU time of 16 millisecond(s) was elapsed. The Power Consumption activity has only one unit, consuming 1 millisecond(s) of CPU time. However, the integration testing results for the 3 activities consumed a total of 6 millisecond(s) from the CPU time, showing that component execution is more effective than executing unit by unit separately.

9.2 Black-box Testing

Black-box testing examines the functionality of the application regardless of its programming structure and checks the required criteria are met by carrying out a series of tests. A "unit" in a program is considered the smallest component in a system/application. Scaled at a rating of 1 to 5, multiple tests are carried out to confirm the unit functionality and accuracy of the output result. The major methods and functions required for optimum application functionality is displayed below. Other test case results are displayed in the Appendix A.

Test case 1: Indicate network connectivity status to the user

Scenario	If the hotspot is active, display "Enabled" within the Network Status text box. Else, display "Inactive". Notify the user with a toast message regarding the hotspot status.
Input	User input and interaction on HotSpot On/Off switch
Expected result	Display "Inactive" text if the hotspot is inactive; if the hotspot is active, display "Enabled" in the textbox Network Status. Display "HotSpot enabled" as a toast when the hotspot is active.
Actual result	Display "Inactive" text if the hotspot is inactive; if the hotspot is active, display "Enabled" in the textbox Network Status. Display "HotSpot enabled" as a toast when the hotspot is active.
Rating and criteria status	**Rating: 5** **Criteria: required status is achieved by the application.**

Table 9 - Black box Testing - Test case 01

Test case 2: Display the battery status and plugged in source

Scenario	Display the battery charging/discharging status and the plugged in source type
Input	Device battery sensor readings
Expected result	The user is notified “Charging” if the battery is at charging state, else displayed “Discharging” in the textbox. The plugged source is notified from the available types “AC”, “USB” and “WIRELESS”.
Actual result	The user is notified “Charging” if the battery is at charging state, else displayed “Discharging” in the textbox. The plugged source is notified from the available types “AC” and “USB”. A device with wireless charging feature was lacking during the application testing phase to check for “WIRELESS” state detection.
Rating and criteria status	**Rating: 4** **Criteria: required competencies are met by the application** **Constraints: lacks results from a device with wireless charging capability.**

Table 10 - Black box Testing - Test case 02

Test case 3: Display the battery temperature and remaining battery voltage

Scenario	Display the current battery temperature and voltage at the application launch
Input	Device battery sensor readings
Expected result	The user is notified the current battery temperature and battery voltage of the device.
Actual result	The user is notified the current battery temperature and battery voltage of the device.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.** **Constraints: assuming the device sensors are accurate as the readings solely depend on the device sensor readings.**

Table 11 - Black box Testing - Test case 03

Test case 4: Display the network upload and download rates

Scenario	Display the network activity of the device for the current active session.
Input	Device network activity readings
Expected result	Display the ongoing upload and download rates in kB/s or MB/s for the current active session.
Actual result	Display the ongoing upload and download rates in kB/s and MB/s for the current active session.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.**

Table 12 - Black box Testing - Test case 04

Test case: 5 Display the total uploaded and downloaded bytes and packet count

Scenario	Display the total bytes uploaded and downloaded by the device for the current active session.
Input	Device network activity readings
Expected result	Display the ongoing upload and download rates in kB, MB or GB for the current active session.
Actual result	Display the ongoing upload and download rates in kB, MB or GB for the current active session.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.** **Constraints: the application assumes network routing updates are negligible compared to the transmitted and received bytes.**

Table 13 - Black box Testing - Test case 05

Test case: 8 open the signal strength activity

Scenario	Open the signal strength activity on button press
Input	User interaction on signal strength button.
Expected result	Open the signal strength activity on user interaction of signal strength view button.
Actual result	Open the signal strength activity
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.**

Table 14 - Black box Testing - Test case 08

Test case: 9 open the power consumption activity

Scenario	Open the power consumption activity on button press
Input	User interaction on power consumption button.
Expected result	Open the power consumption activity on user interaction of signal strength view button.
Actual result	Open the power consumption activity
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.**

Table 15 - Black box Testing - Test case 09

Test case: 10 - Display the IP address, hardware type and MAC address and reachability status of the connected devices on opening node monitoring activity

Scenario	Open the power consumption activity on button press
Input	User interaction on power consumption button.
Expected result	Open the power consumption activity on user interaction of signal strength view button.
Actual result	Open the power consumption activity
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.**

Table 16 - Black box Testing - Test case 10

Test case: 11 - Display the SSID, BSSID and device hardware address on opening signal strength activity.

Scenario	Display the SSID, BSSID and device hardware address within the Signal Strength activity
Input	device network connectivity status.
Expected result	Display the SSID and BSSID if the device is connected, display the device hardware/MAC address
Actual result	SSID and BSSID of the connected network is displayed, hardware address of the device is displayed if the device is running on android API level 22 or below.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.** **Constraints: the hardware/MAC address of the device always return 02:00:00:00:00:00 from API level 23 upwards.**

Table 17 - Black box Testing - Test case 11

Test case 12: Display wireless network speed, signal strength and encryption type on opening the signal strength activity

Scenario	Display the Wi-Fi speed, signal strength and encryption type within the Signal Strength activity
Input	Device network connectivity status.
Expected result	Display the Wi-Fi speed, signal strength and encryption type if the device is connected to a network
Actual result	The Wi-Fi speed, signal strength and encryption type of the connected network are displayed to the user.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.**

Table 18 - Black box Testing - Test case 12

Test case: 13: Display the frequency and operating channel on opening the signal strength activity

Scenario	Display the details on operating frequency range of the connected network
Input	Device network connectivity status.
Expected result	Display the frequency range of thee connected network selected from the given categories
Actual result	Display the frequency range of thee connected network selected from the given categories
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.**

Table 19 - Black box Testing - Test case 13

Test case 14: Display IP address, Network mask and default gateway on opening the signal strength activity

Scenario	Display the network IP address, network mask and default gateway within the Signal Strength activity
Input	Device network connectivity status.
Expected result	Display the IP address, network mask and default-gateway assigned by the network.
Actual result	The assigned IP address, network mask and the default-gateway of the network are displayed.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.**

Table 20 - Black box Testing - Test case 14

Test case 15: Open the activity Battery on user click

Scenario	Open the "Battery_Info" activity on button press
Input	User interaction on power consumption button
Expected result	Open battery information activity on button click
Actual result	The battery information activity is opened on button click
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.**

Table 21 - Black box Testing - Test case 15

Test case 16: Display battery status, plugged-in source and battery technology on opening battery status activity

Scenario	Display the information on battery charging/discharging status, plugged-in source type and device battery technology.
Input	Read the device battery sensor readings
Expected result	The user is notified “Charging” if the battery is at charging state, else displayed “Discharging” in the textbox. The plugged source is notified from the available types “AC”, “USB” and “WIRELESS”.
Actual result	The user is notified “Charging” if the battery is at charging state, else displayed “Discharging” in the textbox. The plugged source is notified from the available types “AC” and “USB”. A device with wireless charging feature was lacking during the application testing phase to check for “WIRELESS” state detection.
Rating and criteria status	**Rating: 4** **Criteria: required competencies are met by the application** **Constraints: lacks results from a device with wireless charging capability.**

Table 22 - Black box Testing - Test case 16

Test case 17: Display the battery remaining voltage, current temperature and battery health on opening battery status activity

Scenario	Display the information on battery remaining voltage, current temperature and device battery health.
Input	Read the device battery sensor readings
Expected result	The user is notified with the remaining voltage of the battery displayed in millivolts and the battery temperature at the given instance displayed in degrees Celsius (^{o}C) alongside the battery health of the device.
Actual result	The battery remaining voltage is displayed in millivolts with the battery temperature of the device and the battery health status of the device is displayed
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 23 - Black box Testing - Test case 17

Test case 18: Display the power saving mode status and remaining battery power on a progress bar.

Scenario	Display the information on device power saving mode, and the remaining battery power on a progress bar and as a percentage display adjacent to the progress bar.
Input	Read device battery sensor readings and power saving mode details.
Expected result	The remaining power of the device is displayed in a progress bar and as a percentage value at the given instance. The power saving mode status is accessed and the results are displayed to the user.
Actual result	The remaining power of the device is displayed in a progress bar and as a percentage value at the given instance. The power saving mode status is accessed and the results are displayed to the user if the device is running on android API level 21.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application** **Constraints: the power saving mode supports API level 21 and above**

Table 24 - Black box Testing - Test case 18

Test case 24: Display the device activity according to the threshold values set

Scenario	Display device activity depending on the battery sensor readings
Input	Device battery sensor readings.
Expected result	Display “Low”, “High” and “Inactive” depending on the battery sensor readings.
Actual result	The user is notified as “High” when the battery temperature exceeds threshold values, “Inactive” when the battery charge drops below 20% and “Low” if the battery sensor readings are below the threshold values.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.**

Table 25 - Black box Testing - Test case 24

Test case 25: Check the root status of the device when auto-pilot mode is enabled and display the user if the device is not rooted through a toast

Scenario	Check for the root status of the device and notify the user if the device is rooted or not.
Input	Read the System files for buildTags, SU path and SU xbin files, user interaction with the auto-pilot ode on/off switch
Expected result	Display the root status of the device in a toast message.
Actual result	The program will notify the user if the device is rooted. Else, the device is not rooted is displayed.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 26 - Black box Testing - Test case 25

Test case 26: if the device passes the root status check, the auto-pilot mode is enabled.

Scenario	Enable the auto-pilot mode “mobile data enable/disable feature” if the device is rooted.
Input	Read the System files for buildTags, SU path and SU xbin files, user interaction with auto-pilot mode on/off switch, device battery sensor readings.
Expected result	Enable the auto-pilot mode and control the mobile network connection if the device is rooted.
Actual result	Automatically enable and disable the mobile data connection if the device is rooted.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 27 - Black box Testing - Test case 26

Test case 27: the mobile data connection is enabled and disabled according to the readings from the device battery temperature, remaining charge and plugged in source.

Scenario	Enable the auto-pilot mode "mobile data enable/disable feature" according to the device battery sensor readings.
Input	Read the System files for buildTags, SU path and SU xbin files, user interaction with auto-pilot mode on/off switch, device battery sensor readings.
Expected result	Enable and disable the mobile data connection of the device depending on the device battery sensor readings.
Actual result	Automatically enable and disable the mobile data connection depending on the battery sensor readings if the device is rooted.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 28 - Black box Testing - Test case 27

Test case 28: prompt non-rooted devices to manually control the mobile data connection.

Scenario	If the device is not rooted, prompt the user to enable/disable the mobile data manually.
Input	Read the System files for buildTags, SU path and SU xbin files, user interaction with auto-pilot mode on/off switch.
Expected result	Prompt the user to enable/disable the mobile data connection depending on battery sensor readings.
Actual result	Prompt the user to enable and disable the mobile data connection depending on the battery sensor readings if the device is not rooted.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 29 - Black box Testing - Test case 28

Test case 29: prompt the user to enable and disable the power saving mode

Scenario	Prompt the user to enable/disable the power saving mode of the device regardless of the root status.
Input	Read the System for power saving mode status, user interaction with auto-pilot mode on/off switch.
Expected result	Prompt the user to enable and disable the power saving mode regardless of the root status of the device.
Actual result	The user is prompted to enable and disable the power saving mode of the device regardless of the root status of the device.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 30 - Black box Testing - Test case 29

9.3 User testing

The application implemented was handed over to the same participants who took part in the user analysis survey and asked to rate the application depending on their experience on the implemented system. The questionnaire carries a series of questions gathering the user experience on the implemented system. The survey questions on the usability and the accessibility of the application, features supported and their rating on major components monitored including device power consumption, network activity, signal strength and resource utilization. The user is asked to rate the application based on their experience and if the requirements are fulfilled. The user testing results are displayed in the pie-charts given below.

9.3.1 Features supported

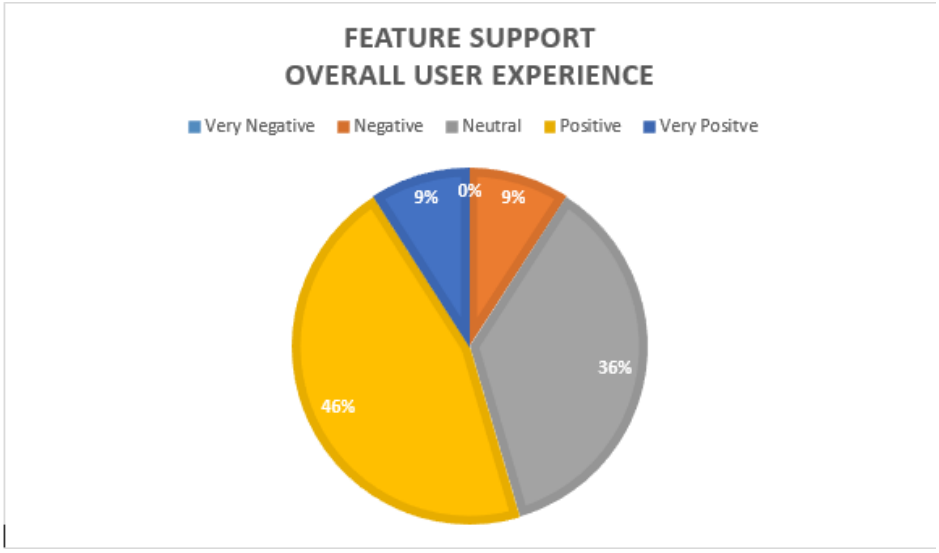

Figure 121 - Feature support user rating

The feature support of the overall user experience returned 9% of rating level 5, 46% positive of rating 4, 36% of rating level 3 and 9% of rating level 2. The overall user experience is over 91% user positive feedback.

9.3.2 Accessibility

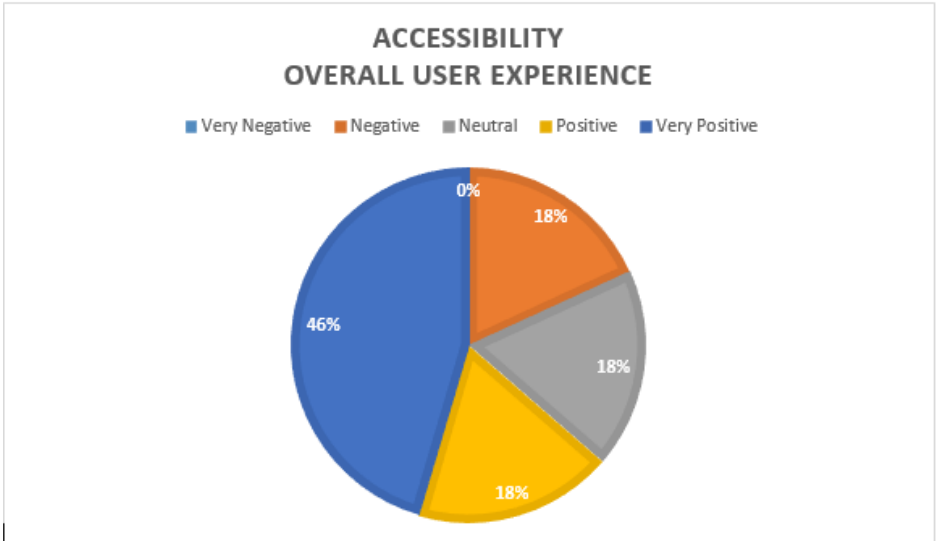

Figure 122- Application accessibility user rating

The accessibility of the overall user experience returned 46% of rating level 5, 18% positive of rating 4, 18% of rating level 3 and 18% of rating level 2. The overall user experience is over 82% user positive feedback.

9.3.3 Power consumption

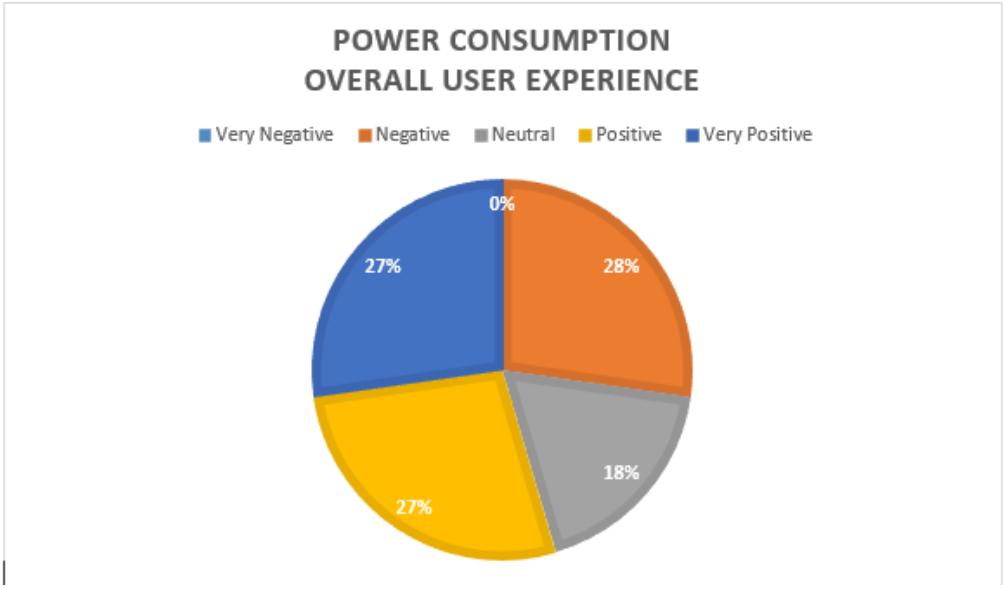

Figure 123 – Power Consumption user rating

The survey carried out on the power consumption activity experience returned 27% of rating level 5, 27% positive of rating 4, 18% of rating level 3 and 28% of rating level 2. The overall user experience is over 72% user positive feedback.

9.3.4 Resource utilization

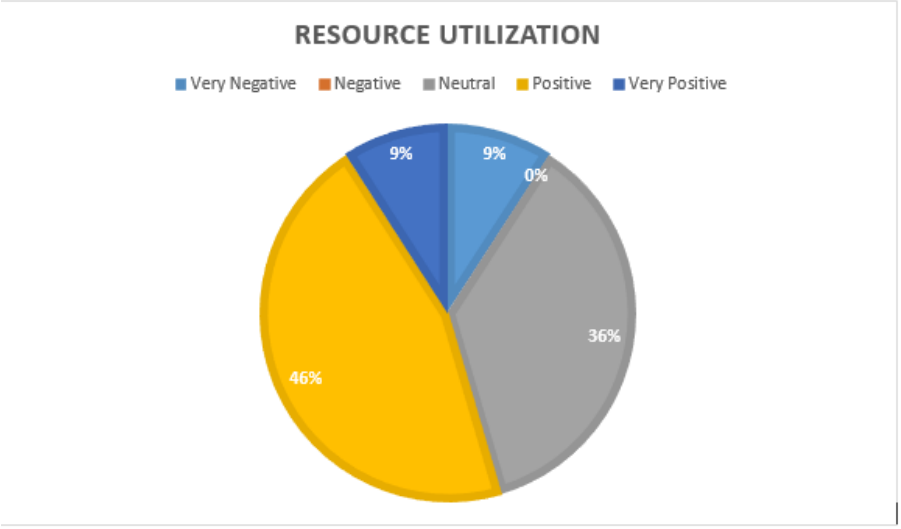

Figure 124 – Resource Utilization user rating

The survey carried out on the overall user experience returned 9% of rating level 5, 46% positive of rating 4, 36% of rating level 3 and 9% of rating level 2. The overall user experience is over 91% user positive feedback.

9.3.5 Signal strength

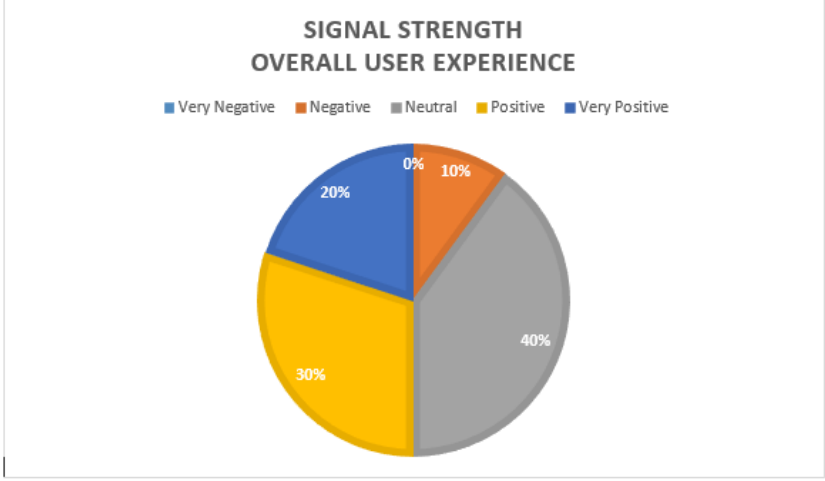

Figure 125 - Signal Strength user rating

The survey carried out on the overall user experience returned 20% of rating level 5, 30% positive of rating 4, 40% of rating level 3 and 10% of rating level 2. The overall user experience is over 90% user positive feedback.

9.3.6 Upload download

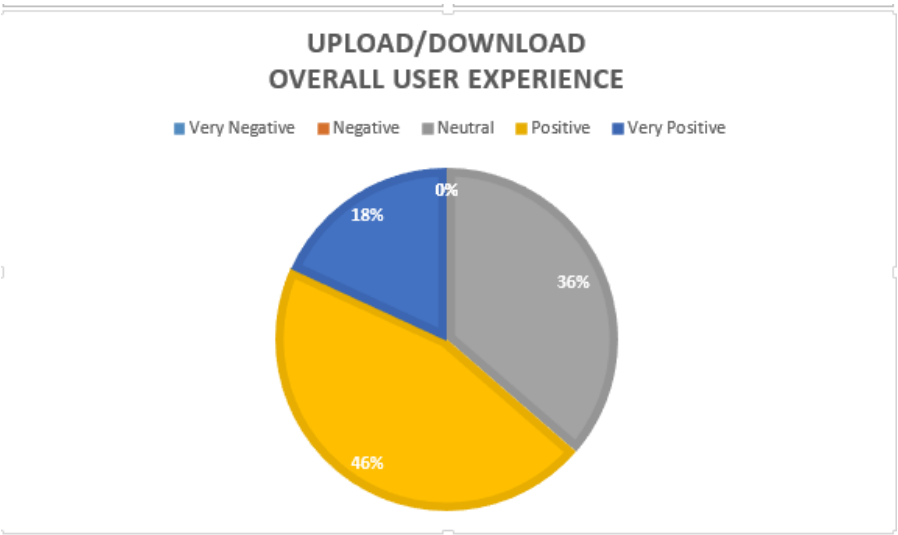

Figure 126 - Upload/Download user rating

The survey carried out on upload and download returned 18% of rating level 5, 46% positive of rating 4, 36% of rating level 3 and no rating on level 1 and level 2. The overall user experience is a 100% user positive feedback.

9.3.7 Usability

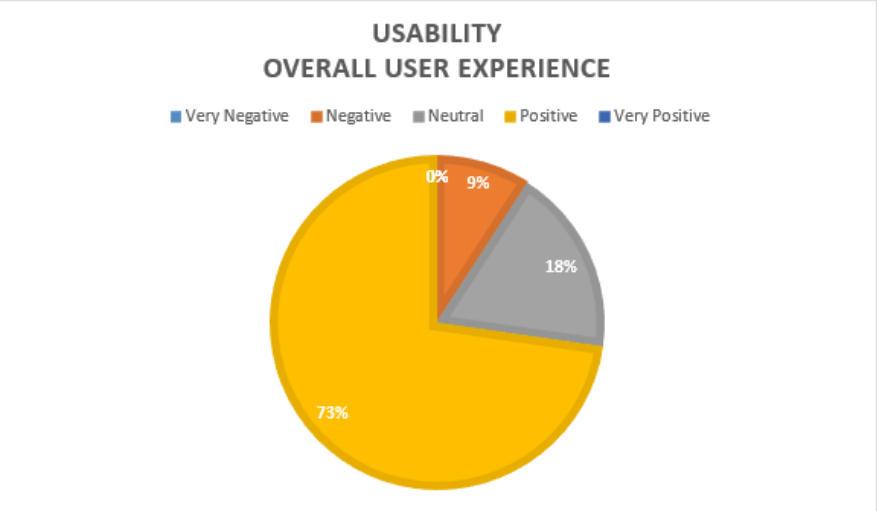

Figure 127 - Usability user rating

The accessibility of the overall user experience returned 0% of rating level 5, 73% positive of rating 4, 18% of rating level 3 and 9% of rating level 2. The overall user experience is over 91% user positive feedback.

9.3.8 Overall rating

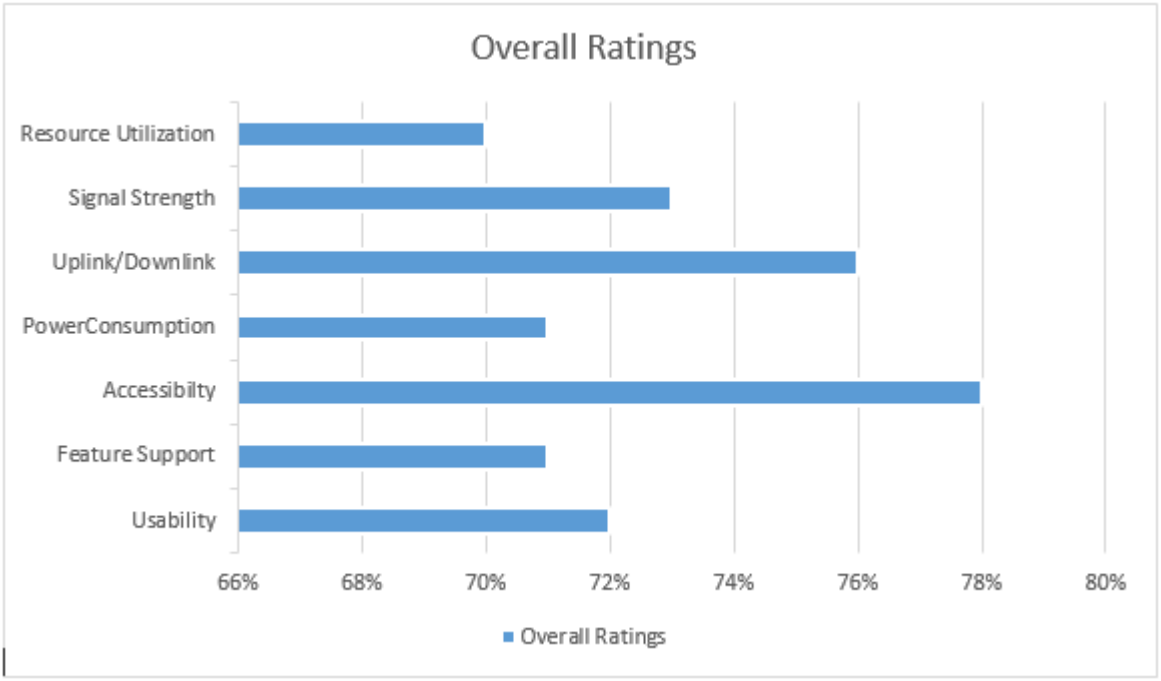

Figure 128 - Overall application ratings

The overall rating on the user feedback includes a rating of 70% on resource utilization monitoring, 73% on Signal Strength monitoring, and 76% on upload and download monitoring, 71% on power consumption and 78% on accessibility of the application. Features provided by the application is rated at 71% and a 72% rating on usability of the application.

9.4 Testing the Effectiveness Of The Implemented System

Test case	Expected result	Actual outcome	Comments
Auto-pilot mode enable/disable feature	Automatically enable/disable the mobile data connection according to the parameters set depending on battery sensor readings	The mobile data will be automatically enabled and disabled only on rooted devices. The devices running on factory-default OS has no ability to modify mobile data connection.	To modify a system settings, permission is required by the application. Devices running on stock- OS don't have permission granted to modify system settings as the device security settings can be compromised.
Power consumption monitoring feature	Monitor the battery charging/discharging status, plugged in source, battery technology, temperature and voltage, battery health and power saving mode status	The battery charging/discharging status, plugged in source, battery technology, temperature and voltage, battery health details are displayed. The power saving mode status is only	The method utilized to read the power saving mode status supports API level 21 and above. The devices running below API level 21 has no functionality to support the API; therefore SDK

		displayed on android devices running on API level 21 and above.	Unsupported is returned.
Upload and download monitor	Calculate the upload and download rates per second, total uploaded and downloaded bytes and packets for the current session	The upload and download rates are displayed in kB/s or MB/s. The total uploaded and downloaded values are displayed in kB, MB or GB accordingly. The total packets uploaded and downloaded are displayed.	The application doesn't provide the functionality to capture the uploaded and downloaded packets for analysis purposes. The packets are counted and the resulting values are displayed to the user.
Display the network details	Display the SSID, BSSID, MAC address, Wi-Fi signal strength and speed, network encryption type, operating frequency and channel, IP address, network mask and the default-gateway of the connected network.	The SSID, BSSID, Wi-Fi signal strength and speed, network encryption type, network frequency and channel number, assigned IP address, subnet mask and default-gateway of the connected network are also displayed. The hardware address of the device returns	The method declared to read the hardware address is deprecated from API level 23 considering the security reasons. The returned value will always be 02:00:00:00:00:00 regardless of the operating device type.

		02:00:00:00:00:00 in devices running on android API level 23 and above considering the security reasons. Devices running on API level 22 and below will return the device hardware address.	
Connected node monitoring feature	Provide the details of the IP address, hardware address, and type and reachability status of the connected node/host.	The IP address, hardware address, reachability status and hardware type of the connected host/node are displayed. The node devices display details on all connected nodes and the host devices display the details of the node connected to.	The node/host network activity (upload and download rates) are not monitored as the process is resource intensive and heavily dependent on system resources, significantly affecting the power consumption of the device.

Table 31 - effectiveness testing of the implemented system

9.5 Evaluating Effectiveness of The System Compared To Available Tools

	MANET Manager	FoxFi	Fidelity-Fi
Calculate network activity and bandwidth use	✖	✖	✓
Monitor power consumption of the device	✖	✖	✓
Monitor network for connected nodes	✖	✓	✓
Monitor system resource utilization	✖	✖	✓
Network signal strength detection	✓	✓	✓
Control device network	✓	✓	✓
Usable on devices with factory-default OS	✖	✓	✓

Table 32 - evaluating effectiveness of the system compared to available tools

CHAPTER 10 – CRITICAL EVALUATION

The research was carried out under two main categories; as the domain research and technical research. The domain research includes a thorough analysis carried out on existing technologies and a review on their strengths and weaknesses. A user analysis is carried out following multiple interview methods focusing on their experience with the existing systems and the drawbacks as seen by the users. The requirement specifications are elected and core components of the system are implemented according to the data provided in the user analysis. Depending on the user requirement analysis, the application criteria are categorized under functional and non-functional requirements. After the application implementation, another survey is carried out on the same participants of the initial survey and their feedback are summarized and displayed in pie-charts for easy analysis and overview. After the user requirement analysis, components and the adjacent APIs required for implementation are gathered. As the application is developed to be compatible with android platform, Android studio is used as the IDE for application implementation, development and test purposes.

A deep technical research is carried out on data communication architectures, SNMP protocol and its components, user authentication algorithms deployed and factors directly affecting the required and monitored components. After gathering information on required components, a software architecture is designed for the proposed system. The software architecture includes a class diagram, use case diagram and activity diagrams briefing the application.

Testing is carried out simultaneously with implementation, where every unit is tested before moving into a new unit. Once all units pass the tests, the unit integrity is checked by testing the method containing all units. The above procedure is carried out on all available methods. After implementation, certain changes were made to the application to increase the efficiency. Network activity methods were moved to the main menu activity to increase the usability and reduce system resource utilization. Android studio debugger, emulator and android build system features are used for efficient application implementation and testing. The GUI of the application is tested on android emulator, the unit testing is carried out on two devices, one running on android API level 23 (version 6.0.1) without root; while the other device runs on android API level 19 (version 4.4.2)

with rooted OS. As the application is tested on both rooted and non-rooted devices, the application functionality is ensured on both rooted and non-rooted devices. Other than white-box and black-box testing methods, user testing phase is carried out involving the effectiveness testing and an evaluation of effectiveness of the implemented system over other systems available.

CHAPTER 11 – CONTRIBUTION

By implementing an application to monitor the device resource utilization and network activity during ad-hoc network session, the users are provided with functionality on monitoring a step ahead. With the existing applications, the user is only given features to implement and disable ad-hoc networks, select the encryption type, SSID name and a few functionality on the hardware resource control, with no functions provided to monitor device resource utilization or network utilization. Considering the functions available within existing applications and their limitations, the new application is implemented in such a way that the limitations are minimized and the user expected criteria are met.

The application is designed and implemented a step forward, by embedding the functionality to monitor the connected nodes and clients to the network. This is achieved by reading the ARP table values in the file "/proc/net/arp" available within the system available for both rooted and non-rooted devices. Other than node monitoring, the system is monitored for power consumption, network activity and system resource utilization. The above features are available as separate applications. Unifying the monitored features will enable the user to focus on his task rather than worrying over the device.

Conserving the remaining power and enhancing the usable time of the mobile device is the main goal of the application, while enabling the user to view the ongoing network activity and details on connections of the device.

CHAPTER 12 – SUGGESTED FUTURE ENHANCEMENTS

The implemented application has the functionality to monitor ongoing network activity, resource utilization and view details of the connected devices. However, the application lacks features to capture ongoing network activity and provide the details for analysis purposes. This feature was skipped as the allocated time frame was not adequate to meet the criteria.

For future enhancements, the application can be further developed to capture network activity in PCAP format, decode the captured packets and monitor for malicious network behavior. Taking a step further ahead, by decoding the network packets real-time, the network activity of the connected nodes can be identified. By following this procedure, the connected node users can be warned of high resource utilization.

The application can also be developed into a file-sharing monitoring system, by embedding the functionality to transfer files between the connected nodes of the network. The node device can be configured as the file-sharing server by enabling both connectivity and monitoring options.

CHAPTER 13 - CONCLUSION

After the domain analysis, the common problems faced by users due to the limitations on existing systems are identified and an application has proposed with suitable options to overcome the limitations and weaknesses on the existing systems. Mainly, the unavailability of a centralized solution to monitor the upload and download, identify the connected nodes and monitor device resource utilization during the given session is identified as the main problem existing in the selected domain.

Depending on the domain analysis, a solution with network connectivity monitoring application is provided with added facility to identify the connected nodes and monitor device resource utilization; specially the battery power consumption, as the mobile devices pack a limited amount of charge when unplugged from the power source. The implementation of an application with added facility to monitor the required components is identified as the solution for the existing problem. The developed solution provides added functionality to monitor the device resource utilization, including device power consumption, signal strength monitoring, identify the connected nodes; regardless of its mode of operation, as a node or as a host. To complete the technical research category, the author has used multiple research papers published under the domain. The techniques and data gathered from those trusted sources are documented prior to the application implementation phase.

Researching in the domain of the problem has provided the author with information beneficiary for the implementation and development phases of the proposed solution. As a result, an application with capabilities of controlling the device resources depending on device battery sensor readings have been implemented. The developed application is capable of displaying accurate outputs of the device regardless of its rooted status; thus addressing the main problem existing in the selected domain.

CHAPTER 14 - REFERENCES

Bakalis, P., Bello, L., Rapajic, P. & Anang, K. A., 2013 . *Power Consumption Analysis in Mobile Ad Hoc Networks.* London, IEEE.

Bolding, K. & Snyder, L., 1994. *Parallel Computing Routing and Communication.* 1st ed. Washington: Springer - Verlag.

Bordim, J. L., Barbosa, A. V., Caetano, M. F. & Barreto, P. S., 2010. *IEEE802.11b/g Standard: Theoretical Maximum Throughput.* s.l., IEEE.

Buchmann, J. A., 2000. *Introduction to Cryptography.* 2nd ed. Germany: Springer Verlac NewYork, Inc..

Carmack, C., 2005. *How BitTorrent Works.* [Online]
Available at: http://computer.howstuffworks.com/bittorrent.htm
[Accessed 31 January 2016].

CCM, 2016. *Networking - 3-Tier Client/Server Architecture.* [Online]
Available at: http://ccm.net/contents/151-networking-3-tier-client-server-architecture
[Accessed 31 January 2016].

Chang, D., 2016. *WEP.* [Online]
Available at: http://www.math.ucsd.edu/~crypto/Projects/DavidChang/WEP.htm
[Accessed 31 January 2016].

Chou, H. & Kang, M., 2010. *CCNA Wireless Exam 640-721.* 1st edition ed. New Delhi: Tata McGraw Hill Education Pvt Ltd. .

Code.google.com, 2015. *Google Code Archive - Long-term storage for Google Code Project Hosting..* [Online]
Available at: https://code.google.com/archive/p/android-wifi-tether/
[Accessed 24 December 2015].

Coleman, D. D. & Westcott, D. A., 2009. *CWNA Certified Wireless Network Administrator.* 1st Edition ed. Indiana: Wiley Publishing Inc..

Cope, J., 2002. *Peer-to-Peer Network.* [Online]
Available at: http://www.computerworld.com/article/2588287/networking/peer-to-peer-network.html
[Accessed 31 January 2016].

Cozirok, C. M., 2005. *TCP/IP Guide A COMPREHENSIVE ILLUSTRATED INTERNET PROTOCOLS REFERENCE.* 1st ed. San Francisco: No Starch Press Inc..

Developer.android.com, 2016. *<uses-sdk> | Android Developers.* [Online]
Available at: https://developer.android.com/guide/topics/manifest/uses-sdk-element.html
[Accessed 21 February 2016].

Developer.android.com, 2016. *ActivityManager.MemoryInfo | Android Developers.* [Online]
Available at:
https://developer.android.com/reference/android/app/ActivityManager.MemoryInfo.html
[Accessed 28 April 2016].

Developer.android.com, 2016. *Android 6.0 Changes | Android Developers.* [Online]
Available at: https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html
[Accessed 20 January 2016].

Developer.android.com, 2016. *Android Debug Bridge | Android Studio.* [Online]
Available at: https://developer.android.com/studio/command-line/adb.html
[Accessed 26 April 2016].

Developer.android.com, 2016. *BroadcastReceiver | Android Developers.* [Online]
Available at: https://developer.android.com/reference/android/content/BroadcastReceiver.html
[Accessed 27 February 2016].

Developer.android.com, 2016. *Configure Your Build | Android Studio.* [Online]
Available at: https://developer.android.com/studio/build/index.html
[Accessed 26 April 2016].

Developer.android.com, 2016. *ConnectivityManager | Android Developers.* [Online]
Available at: https://developer.android.com/reference/android/net/ConnectivityManager.html
[Accessed 22 February 2016].

Developer.android.com, 2016. *Debug Your App | Android Studio.* [Online]
Available at: https://developer.android.com/studio/debug/index.html
[Accessed 26 April 2016].

Developer.android.com, 2016. *Handler | Android Developers.* [Online]
Available at: https://developer.android.com/reference/android/os/Handler.html
[Accessed 12 March 2016].

Developer.android.com, 2016. *Package Index | Android Developers.* [Online]
Available at: https://developer.android.com/reference/packages.html
[Accessed 21 February 2016].

Developer.android.com, 2016. *PowerManager | Android Developers.* [Online]
Available at:
https://developer.android.com/reference/android/os/PowerManager.html#isPowerSaveMode()
[Accessed 23 April 2016].

Developer.android.com, 2016. *Run Apps on the Android Emulator | Android Studio.* [Online]
Available at: https://developer.android.com/studio/run/emulator.html
[Accessed 26 April 2016].

Developer.android.com, 2016. *Update the IDE and Tools | Android Studio.* [Online]
Available at: https://developer.android.com/studio/intro/update.html
[Accessed 26 April 2016].

Developer.android.com, 2016. *WifiInfo | Android Developers.* [Online]
Available at: https://developer.android.com/reference/android/net/wifi/WifiInfo.html
[Accessed 32 February 2016].

Developer.android.com, 2016. *WifiManager | Android Developers.* [Online]
Available at: https://developer.android.com/reference/android/net/wifi/WifiManager.html
[Accessed 22 February 2016].

Developer.android.com, 2016. *Developers, TrafficStats | Android.* [Online]
Available at: http://developer.android.com/reference/android/net/TrafficStats.html
[Accessed 28 February 2016].

Developer.android.com, 2016. *TrafficStats | Android Developers.* [Online]
Available at: http://developer.android.com/reference/android/net/TrafficStats.html
[Accessed 01 March 2016].

Developer.xamarin.com, 2016. *Understanding Android API Levels - Xamarin.* [Online]
Available at:
https://developer.xamarin.com/guides/android/application_fundamentals/understanding_android_api_levels/
[Accessed 21 February 2016].

Docwiki.cisco.com, 2012. *Simple Network Management Protocol - DocWiki.* [Online]
Available at: http://docwiki.cisco.com/wiki/Simple_Network_Management_Protocol
[Accessed 04 February 2016].

ElBatt, T. A., Krishnamurthy, S. V., Connors, D. & Dao, S., 2000. *Power Management for Throughput Enhancement in Wireless Ad-Hoc networks.* California, IEEE.

Ellingwood, J., 2014. *An Introduction to SNMP (Simple Network Management Protocol) | DigitalOcean.* [Online]
Available at: https://www.digitalocean.com/community/tutorials/an-introduction-to-snmp-simple-network-management-protocol
[Accessed 05 February 2016].

Forouzan, B. A., 2007. *Data Communications and Networking.* 4th ed. New York: McGraw Hill.

FoxFi, 2012. *FoxFi - Turn Android into free WiFi Hotspot (no root).* [Online]
Available at: http://foxfi.com/?src=m
[Accessed 26 December 2015].

Godwin, M., 2016. *Freenet - About.* [Online]
Available at: https://freenetproject.org/about.html
[Accessed 31 January 2016].

Gralla, P., 2005. *Windows XP hacks.* 2nd edition ed. United States of America: O'Reilly.

Greenfield, J., 2003. *What the TKIP protocol is all about - TechRepublic.* [Online]
Available at: http://www.techrepublic.com/article/what-the-tkip-protocol-is-all-about/
[Accessed 31 January 2016].

Gupta, P., Saxena, P., Ramani, A. & Mittal, R., 2010. *Optimizeed use of battery power in Wireless ad-hoc networks.* Korea, IEEE.

Gust, M. S., 2013. *802.11n A Survival Guide.* 3rd ed. Sebastopol: O'Reilly Media, Inc. .

Hofman, M., 2009. *Cyber Security Awareness Month - Day 12 Ports 161/162 Simple Network Management Protocol (SNMP).* [Online]
Available at: InfoSec Handlers Diary Blog
[Accessed 12 February 2016].

Johns, A., 2015. *Mastering Wireless Penetration Testing for Highy Secured Environments.* 1st edition ed. Birmingham: Packt Publishing Ltd..

Jun, J., Peddabachagari, P. & Sichitiu, M., 2003. *Theoretical Maximum Throughput of IEEE 802.11 and its Applications.* s.l., IEEE.

Knowlin, K., 2016. *Client Server Architecture Types | eHow.* [Online]
Available at: http://www.ehow.com/list_6706525_client-server-architecture-types.html
[Accessed 31 January 2016].

Lee, H. C., 2007. *Theoretical Maximum and Saturation Throughput in The 802.11a Wireless LAN.* s.l., IEEE.

Li, P., Fang, Y. & Li, J., 2010. *Throughput, Delay, and Mobility in Wireless Ad-hoc Networks.* San Diego, IEEE INFOCOM.

Murphy, G. B., 2015. *Systems Security Certified Practitioner (SSCP).* 1st edition ed. Canada: John Wiley & Sons, Inc..

OpManager, M., 2016. *Network Monitoring Software by ManageEngine OpManager.* [Online]
Available at: https://www.manageengine.com/network-monitoring/what-is-snmp.html#snmp-manager
[Accessed 05 February 2016].

OpManager, M., 2016. *Network Monitoring Software by ManageEngine OpManager.* [Online]
Available at: https://www.manageengine.com/network-monitoring/what-is-snmp.html#mib
[Accessed 05 February 2016].

Perkins, D. D., Hughes, H. D. & Owen, C. B., 2002. *Factors Affecting the Performance of Ad Hoc Networks,* East Lansing: IEEE.

Play, G., 2015. [Online]
Available at: https://play.google.com/store/apps/details?id=org.span
[Accessed 24 December 2015].

Play, G., 2015. *Google Play*. [Online]
Available at: https://play.google.com/store/apps/details?id=og.android.tether
[Accessed 26 December 2015].

Play, G., 2015. *Google Play*. [Online]
Available at: https://play.google.com/store/apps/details?id=com.foxfi
[Accessed 27 December 2015].

Posey, B. et al., 2008. *MCTS/MCITP Windows Server 2008 Configuring Network Infrastructure.* 1st Edition ed. Burlington: Syngress Publishing Inc..

Schemeh, K., 2001. *Cryptography and Public Key Infrastructure on the Internet.* 1st ed. Germany: John Wiley & Sond Ltd..

Schollmeier, R., 2002. *A Definition of Peer-to-Peer Networking for the Classification of Peer-toPeer Architectures and Applications*, Munchen: Institute of Communication Networks, Technische University, .

Singh, P., Mishra, M. & Barwal, P. N., 2014. *Analysis of security issues and their solutions in wireless LAN.* Chennai, IEEE, p. 6.

Tcpipguide.com, 2005. *The TCP/IP Guide - SNMP Version 2 (SNMPv2) Message Formats.* [Online]
Available at: http://www.tcpipguide.com/free/t_SNMPVersion1SNMPv1MessageFormat.htm
[Accessed 04 February 2016].

Tcpipguide.com, 2005. *The TCP/IP Guide - SNMP Version 2 (SNMPv2) Message Formats.* [Online]
Available at: http://www.tcpipguide.com/free/t_SNMPVersion2SNMPv2MessageFormats-5.htm
[Accessed 04 February 2016].

Tcpipguide.com, 2005. *The TCP/IP Guide - SNMP Version 3 (SNMPv3) Message Format.* [Online]
Available at: http://www.tcpipguide.com/free/t_SNMPVersion3SNMPv3MessageFormat.htm
[Accessed 04 February 2016].

Technet.microsoft.com, 2016. *How SNMP Works: Simple Network Management Protocol (SNMP); Services for Macintosh.* [Online]
Available at: https://technet.microsoft.com/en-us/library/cc783142(v=ws.10).aspx
[Accessed 05 February 2016].

Technet.microsoft.com, 2016. *How SNMP Works: Simple Network Management Protocol (SNMP); Services for Macintosh.* [Online]

Available at: https://technet.microsoft.com/en-us/library/cc783142(v=ws.10).aspx
[Accessed 05 Feebruary 2016].

Technet.microsoft.com, 2016. *port Assignments and Protocol numbers.* [Online]
Available at: https://technet.microsoft.com/en-us/library/cc959834.aspx
[Accessed 16 February 2016].

CHAPTER 15 – APPENDICES

Appendix A – Black Box testing test results

Test case: 6 Enable auto-pilot mode and display the status

Scenario	Display the operating mode of the device when auto-pilot mode is enabled.
Input	Device battery sensor readings
Expected result	Display the device status from different modes “Suspended” or “active” if the auto-pilot mode is enabled and the device is rooted. else display “Unsupported”
Actual result	Depending on battery sensor readings, the device activity is displayed to the user. If the device is rooted and the auto-pilot mode is enabled, the user is displayed “suspended” or “active depending on the device operating mode. If the device is not rooted, “Unsupported” is returned.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.** **Constraints: the application assumes network routing updates are negligible compared to the transmitted and received bytes.**

Table 33 - Black Box testing Test case 06

Test case 19: Display the total RAM, used RAM and free RAM on opening RAM manager activity

Scenario	Display the information on device power saving mode, and the remaining battery power on a progress bar and as a percentage display adjacent to the progress bar.
Input	Read device battery sensor readings and power saving mode details.
Expected result	The remaining power of the device is displayed in a progress bar and as a percentage value at the given instance. The power saving mode status is accessed and the results are displayed to the user.
Actual result	The remaining power of the device is displayed in a progress bar and as a percentage value at the given instance. The power saving mode status is accessed and the results are displayed to the user if the device is running on android API level 21.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application** **Constraints: the power saving mode supports API level 21 and above**

Table 34 - Black Box testing Test case 19

Test case 20: Display the CPU details on opening CPU info activity

Scenario	Display the information on device CPU
Input	Read device "/proc/cpuinfo" file and display the details
Expected result	A detailed version of the device CPU details is displayed in a scrollable view text box.
Actual result	Details of the device CPU information including the processor type, number of cores, CPU architecture, and hardware details are displayed.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 35 - Black Box testing Test case 20

Text case 7: Open node monitoring activity

Scenario	Open the node monitoring activity on button press
Input	User interaction on node monitoring button.
Expected result	Open the node monitoring activity on user interaction of node monitoring view button.
Actual result	Open the node monitoring activity
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application.**

Table 36 - Black Box testing Test case 07

Test case 21: Display the OS version, API level and device manufacturer name on opening the device info activity

Scenario	Display the information on device OS, API level and device manufacturer name
Input	Read the System files and device configuration files.
Expected result	Display the OS version, API level and manufacturer ID given for the device by the manufacturer
Actual result	Display details on OS version, API level and devic1e manufacturer ID
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 37 - Black Box testing Test case 21

Test case 22: Display the device model, Product and OS version release on opening the device info activity.

Scenario	Display the information on device model, product release region and OS version release.
Input	Read the System files and device configuration files.
Expected result	Display the information on device model name, product release region and OS version of the device.
Actual result	Displays the device model name information, product release region and OS version of the current device.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 38 - Black Box testing Test case 22

Test case 23: Display the device brand, Build ID, manufacturer and the device serial ID on opening device info activity.

Scenario	Display the information on device brand, manufacturer details and serial ID of the device.
Input	Read the System files and device configuration files.
Expected result	Display device brand information, details on manufacturer and serial of the device.
Actual result	Display information on device brand, manufacturer details and serial ID of the device.
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 39 - Black Box testing Test case 23

Test case 30: prompt user by a popup prompting the application

Scenario	Pop-up an alert dialog prompting to close the application when the user press the return key.
Input	User interaction on the back press key, current application open activity
Expected result	Prompt the user to exit the application on pressing the return key.
Actual result	The program will return to the previously opened activity on return key press, until the main menu activity is reached. Once on the main menu activity, the program will prompt the user to exit the application on return key press
Rating and criteria status	**Rating: 5** **Criteria: required competencies are met by the application**

Table 40 - Black Box testing Test case 30