On-the-Fly Establishment of Multi-hop D2D Communication based on Android Smartphones and Embedded Platforms:

Implementation and Real-Life Experiments

By

Harald Gramstad Lie and Michael Stensrud

Supervisor

Prof. Frank Y. Li

University of Agder
Faculty of Engineering and Science
Department of Information and Communication Technology

Grimstad, May 26, 2015

Abstract

In the last decade, we have experienced natural disasters and crisis situations happening more rapidly than before. This fact has led to an increasing demand for developing systems which offer communication capability where conventional telecom infrastructure is either inaccessible or congested. With regard to this challenge, D2D communication has been envisaged as a promising technology for P2P communication in such situations. This technology enables *easy to deploy* services among devices by allowing wireless communication without infrastructure, in which information is relayed from source to destination in a multi-hop fashion.

In our Master's thesis we have developed a communication system intended for rescue responders and victims that can take the advantage of D2D communication without relying on regular network infrastructure. We have designed and implemented a prototype system which is composed of a portable battery powered embedded system, multiple WiFi routers and Android smartphones. The system supports multi-services such as voice, SMS, MMS and video. The D2D communication is based on ad hoc mode running the OLSR protocol. We demonstrate that the developed prototype system functions properly in real-life, then validated and tested the implemented system through extensive real-life experiments. The numerical results demonstrate that the performance parameters such as delay and packet loss meet the QoS requirements for D2D communication. Furthermore, the system is also capable of providing connection to the Internet if required, through cellular networks, WiFi or fixed connection. In addition, the system we developed is able to provide hybrid and dynamic IP address allocation for end users.

Keywords: *Device-to-Device Communication, OLSR, Multi-hop Communication, Android Smartphones, Multi-services, Design and Implementation, Real-Life Experiments*

Preface

This report was written as the final result in IKT590 - Master's thesis (30 ECTS credits), at the Faculty of Engineering and Science, University of Agder (UiA) in Grimstad, Norway. The work was performed from the beginning of January 2015 to May 26, 2015.

We would like to thank our supervisor Prof. Frank Y. Li for his assistance in giving us constructive and helpful feedback, on both technical and practical contents of the report throughout this thesis period. We would also like to thank the Department of Information and Communication Technology at UiA for providing us with funds to buy the equipment needed to complete our Master's thesis. Finally we would like to thank our respective families for supporting us through this thesis period.

Grimstad
May 26, 2015

[illegible]
Harald Gramstad Lie

[illegible]
Michael Stensrud

Contents

List of Figures

List of Tables

Listings

Acronyms

3GPP	3rd Generation Partnership Project
ADB	Android Debug Bridge
ADS	Alert Dissemination Strategy
AODV	Ad hoc On-Demand Distance Vector Routing
AP	Access Point
APK	Android Application Package
ASP	Active Server Page
BATMAN	Better Approach To Mobile Adhoc Networking
BS	Base Station
CPU	Central Processing Unit
D2D	Device-to-Device
DHCP	Dynamic Host Configuration Protocol
DID	Direct-Inward Dialing Numbers
DNA	Distributed Numbering Architecture
DNS	Domain Name System
DTN	Delay/Disruption-Tolerant Network
GUI	Graphical User Interface

HDMI	High-Definition Multimedia Interface
HNA	Host Network Address
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol
IPsec	IP Security
ISP	Internet Service Provider
LAMP	Linux, Apache, MySQL and PHP
LAN	Local Area Network
MANET	Mobile Ad hoc Network
MMS	Multimedia Messaging Service
OEMAN	On-the-Fly Establishment of Multi-hop Wireless Access Network
OLSR	Optimized Link State Routing
OS	Operating System
PHP	Personal Home Page
PSTN	Public Switched Telephone Network
RTP	Real-time Transport Protocol
RTT	Round Trip Time
SDK	Software Development Kit
SID	Subscriber ID

SIP	Session Initiation Protocol
SMS	Short Message Service
SQL	Structured Query Language
SSH	Secure Shell
SSID	Service Set Identifier
TCP	Transmission Control Protocol
TTL	Time-To-Live
UDP	User Datagram Protocol
UI	User Interface
VAP	Virtual AP
WiFi	Wireless Fidelity

Chapter 1

Introduction

This chapter introduces the general organization of the thesis, where we first introduce the basis of the performed thesis work, explain our motivation and the expected goal for the project. Furthermore we define our problem statement and draw some key limitations before we elaborate on our work methodology. Finally we incorporate a guideline of what is to be presented in the following chapters.

1.1 Background and Motivation

Device-to-Device (D2D) communication has been envisaged as part of the 3rd Generation Partnership Project (3GPP) 5G communication paradigm during the past five years. D2D is a upcoming standard that allows devices to communicate directly without any infrastructure. Currently research and standardization activities within the communication paradigm is conducted on the basis of D2D communication, and the technology is intended to provide offloading of unnecessary traffic and to extend coverage to handsets that are outside the coverage area of 5G enabled Base Station (BS). Here we talk about three types of D2D communication. First we have infrastructured D2D where handsets are controlled by the BS to extend coverage, and provide service to other devices. The second is infrastructureless where clients choose to provide access and communicate directly with other handsets without the control of any BS. The last method is an hybrid of the two previous methods where devices and BS cooperate in order to provide offloading and services to other devices.

As part of UiA's contribution towards this hot topic the solution presented in this thesis is triggered from three previous projects (one Master's thesis and two semester projects), and has been an ongoing development for the past two years. It first started with an application to support D2D voice conversations between Optimized Link State Routing (OLSR) enabled Android phones, without the need of any infrastructure. This work was done by two earlier students Magnus Wennberg and Nils Erik Skjønsberg as their Master's thesis in the spring of 2013 [40]. Further improvements were conducted by Michael Stensrud and Henrik Nergaard in their project in the spring of 2014. They improved the Graphical User Interface (GUI) of the application and employed a self-developed Domain Name System (DNS) server, that to some extent supported dynamically allocation of Internet Protocol (IP) addresses [36]. In the spring of 2014 we redeveloped the system to move more against a commercializable system. We further improved the Android application to support a newly developed user framework employed on a Linux server. As well as making the system less dependent of the OLSR protocol, by allowing clients to connect through regular Wireless Fidelity (WiFi) [35].

Even though lots of work has been performed as explained above, an overall solution which includes both user devices and servers, capable of provisioning of multi-services, is still lacking. This observation triggered our motivation to develop such a system as presented in this thesis.

1.2 Project Goal

The main goal for this thesis is to develop a system which produces a solution for D2D communication without traditional telecom infrastructure. This network should also be operational by temporarily deployed infrastructure, which is battery powered. The platform should be developed to be a portable system that could enable multi-services including Short Message Service (SMS), Multimedia Messaging Service (MMS), video, and voice messaging in crisis situations. It should also provide opportunities to further extend the available services. More specifically this thesis should cover the following tasks:

- Design and implement a D2D communication platform.
- Implement an embedded system to provide service and user information in the network.
- Enable automatic IP assignment in the network.
- Enable multi-service including SMS, MMS, video and voice.
- Design and produce a workable prototype.
- Validate and test the performance of the implemented solution in *real world*.

1.3 Problem Statement

In addition to the project goal which is to design, improve and develop a communication system for rescue responders that enables voice communication and exchange messages with or without any infrastructure, the system should enable automatic IP address assignment for new devices. Furthermore the system should be designed to enable mobile battery powered Access Points (APs), that provide network connectivity, IP resolution and opportunity for gateway through either Ethernet or cellular connection. Finally we should design and integrate two portable prototype cases, that enable us to do real-life experiments.

1.3.1 Scope

- Identify one application scenario, with the goal of providing a viable D2D communication platform for crisis situations.
- Design a system solution and implement it both on Android smartphones and OpenWrt platform for communicating with Raspberry Pi -based embedded server.
- Develop a viable prototype and validate by real-life experiments.

1.3.2 Limitations

While accomplishing In order to achieve the project goal within the time schedule, the accomplishment of this thesis project has the following limitations.

- Create an application to the Android platform. IOS and Windows Phone is not considered.
- Android application must have *ROOT* privileges.
- Android devices is limited to three types of model which are Samsung Galaxy S3, Nexus I9250 and Nexus Tablet.
- Perform small- and medium-scale experiments, due to number of available devices and participants to test the system.

1.4 Methodology

1.4.1 General description

This thesis mainly consists of software implementation. Therefore the agile methodology is quite suitable for our purpose, which means our requirements and solutions evolves along our progress of the thesis. Since this thesis consists of two collaborators we will have daily and weekly meetings to evaluate our progress. On the daily meetings, the members can discuss problems and possible approaches to make progress. The weekly meetings consist of the members and the supervisor, where an overview of the progress is explained to the supervisor. In this meeting the supervisor has the opportunity to make suggestions to the members. By using this methodology we are able to keep track of our progress in regards to the timeline.

By utilizing the methodology we are able to adapt to the current development of the system, since development of applications and features most likely introduces a variety of different problems. This is imminent and illustrates the importance of having a solid project plan with a properly working methodology.

1.4.2 Detailed description

As mentioned in the previous subsection we have used the agile methodology. To provide more insight on how to solve both new implementation features and problems, we propose a four step plan that will work in all scenarios. For simplicity we have chosen to use the four step plan when we implement a new feature into the system. Figure 1.1 illustrates the four step plan consist of planning, conduction, processing and finally an evaluation of the new feature. The evaluation is done on behalf of the previously three steps and is validated if it goes towards our desired goal.

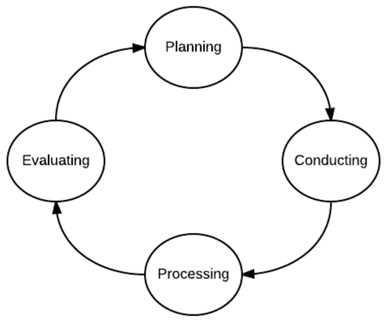


Figure 1.1: Four step process loop.

Planning

When we want to implement a new feature we need to have some kind of theory behind it. The new feature is implemented to further reach our desired goal. With the use of our knowledge and research in certain Internet societies and forums we will be able to grasp the theory behind each implemented feature. As problems arrive, we will use more effort researching in how the feature is supposed to work. In the beginning of solving a problem we will gather more information using theory to further enhance our knowledge. We also rely on having daily meetings with discussion to see if there is any problem that needs attention. The requirement and approach for the new task is discussed. The requirement is chosen either by our own knowledge or by researching. First we isolate the new feature as much as possible to make it work. Then we scale out to the whole system to see that our approach also works as intended. Finally, we evaluate and discuss if the new solution satisfy our requirement.

Conducting

To verify that a new feature work we need to do some conducting. Since this project mainly relies on practical implementation we need to deploy the new feature and see that it works. When the new feature is implemented we should first try to isolate it as much from the system. This is to make it easier for us to verify that the basics of the feature works as intended. If it works as intended we should scale the feature and test it up against the complete system. The experiment will also mainly consist of practical testing where we can clearly discover if the feature works as planned or not.

Processing

To verify that our feature works we need to gather information about the new feature. This is accomplished by how we collect the data and gather information dependent on what kind of feature we have conducted. On the Android phones we will use a debugging tool called *Logcat* to gather information. For the server scripts we will use own test scripts and sniffing tools to verify that the scripts receive and replies as intended. For the router, we will gather and compare logs after major changes to verify that e.g. firewall blocks or allows our defined rules.

Evaluation

The final step includes an evaluation of the previous steps, where we research how the possible solution works in regards to the initial development plan, and how the solution enhances or depreciates to the desired project plan. The evaluation will be debugged and verified. With the use of the collected data using several tools from the processing step we will be able to make more consistent evaluations. With this in mind the project plan should be updated in regards to the new feature, whilst keeping in mind that we need to clearly define how and why we came to edit the plan. With the updated plan we should move back to step one, and repeat the process for any new implemented feature.

1.5 Report Outline

This report follows a top-down approach, where it first gives a general introduction on the stated problems, what should be done, and what our desired outcome is. The rest of the thesis is organized as follows:

- Chapter 2 includes several short summaries of different related network solutions, and what the similarities are. Further on it elaborates on different technologies and tools that are essential in order to reach the desired goals.
- Chapter 3 includes the initial planning, and the design of the current system and prototype solution.
- Chapter 4 incorporates the implementation efforts done by us, with strong connection to the appendices.
- Chapter 5 covers the real-life experiments with results and evaluation.
- Chapter 6 consists of discussion on our performed work and outcome.
- Chapter 7 includes the final conclusions, contributions and further work.

Chapter 2

Related Work, Enabling Technologies and Tools

In order to accomplish the designed goals we have used different technologies and tools. This chapter aims at providing an overview into the requirements needed in this project, as well as giving a general introduction into some related network solutions.

2.1 Related Work

To provide grounds for comparison we will elaborate on different related projects that aim at providing communication in crisis situations. First we explain in general how the different approaches work, before we draw some comparisons towards our developed system. The three final subsections provides a general introduction into UiA's efforts to provide viable results in the D2D communication field.

2.1.1 On-the-fly establishment of multi-hop communication

To provide Internet access to victims in a disaster area where network infrastructure is broken, the paper [22] introduce a method for On-the-Fly Establishment of Multi-hop Wireless Access Network (OEMAN) for disaster recovery. The method involves utilizing an AP which is not affected by the disaster in surrounding areas, and extending the coverage by making nearby devices download software that transform them into Virtual APs (VAPs). The VAPs then extend the coverage to other devices which again are transformed into VAPs, building up a tree-based topology of all the connected VAPs. This would mean that the APs that are not affected by a disaster must still have a designated power source which could be problematic affected areas, and the lifetime of the network therefore could variate. Figure 2.1 illustrates the three-based multihop OEMAN.

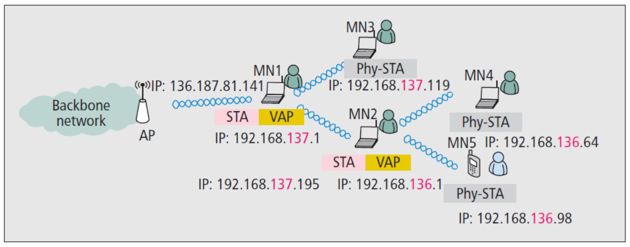


Figure 2.1: Demonstration of a three-based multi-hop OEMAN [22].

The difference between our system is that we aim at providing rescuers with a system that enable them to communicate with each other based on a simple ad hoc network utilizing the OLSR protocol. Our system is preconfigured and should be easy to deploy in crisis situations by using batteries as the designated power source. The system is designed to be portable and rescue workers could easily deploy the network in designated areas.

2.1.2 Realizing multi-hop D2D communications

In order to provide a network for message exchange, the paper [28] introduce a hybrid method between Mobile Ad hoc Network (MANET) and Delay/Disruption-Tolerant Network (DTN) enabled nodes. Even though all nodes run the OLSR protocol they switch between the

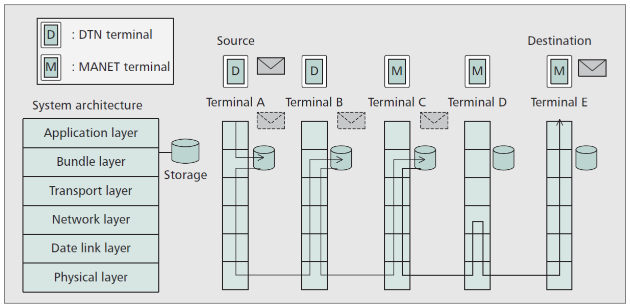


Figure 2.2: Message transmission in the architecture of DTN over MANET [28].

two modes. The reason for enabling this hybrid mode is to ensure that the network is as connectable as possible for both static and highly mobile nodes. MANET operated nodes will be beneficial when they are static, have multiple neighbors, and when the remaining battery power is high. Whereas the use of DTN nodes are favorable when nodes are highly mobile, have less battery power and don't have many neighboring nodes. In this paper they prove the networks feasibility with 30 operating nodes as well as the use of gateways to relay messages further distances. Figure 2.2 illustrate at which layer messages are routed in the implemented system.

There are two main differences between our system and what the publisher of this paper has done. Our system relies on hybrid infrastructure to set up the nodes, whereas the paper require manually configured nodes. Our system automatically assigns IP addresses to new nodes as long as they are covered by our backbone network. After assignment, nodes are free to move wherever they need to go, and as long as they are able to connect to any other node information is relayed throughout the network. We are able to connect D2D multi-service including voice, SMS, picture and video, whereas the published network only are able to exchange text messages. They have enabled possibility to extend the network by introducing gateways, as we also have done. Their gateways are only used to send messages to foreign networks whereas our system is able to provide Internet access for the entire network. The last thing we would like to point out is that our system manages users and provides a backbone embedded server which is part of our battery powered prototype case, which should be able to operate without AC power for several hours.

2.1.3 Serval project

The Serval Project is a communication system that aims at providing telecom free of charge to under-developed countries and in crisis situations, where telecom infrastructure is possibly broken and services are unavailable. The project [10] is based on an Android application that requires *ROOT* access for operation in ad hoc mode, and make it possible to connect smartphones through different operation modes that does not require *ROOT* access. In contrast to our system, it therefore enables more devices to connect to the system. They use the Better Approach To Mobile Adhoc Networking (BATMAN) protocol as an underlying ad hoc routing protocol, whereas we utilize OLSR. The performance difference between BATMAN and OLSR show that the OLSR protocol is a better approach, since OLSR rapidly builds up routing tables for the entire network. This results in it being a better protocol to utilize in a more dynamic network topology, even though it introduce more protocol overhead [32].

The Serval Distributed Numbering Architecture (DNA) [10] provides a framework for identification and call establishment in the network. It consist of a random 256-bit Subscriber ID (SID) address which corresponds to a Direct-Inward Dialing Numbers (DID) which could be a phone number. The voice conversations are interconnected using Session Initiation Protocol (SIP) messages, thus enabling the use of a SIP trunk in the network to handle calls to regular Public Switched Telephone Network (PSTN) network. There has also been done work to provide encryption of the voice conversation by using the Diffie-Helleman key-exchange.

We have currently not made it possible to interconnect calls between our system and the regular PSTN network. Our system is dependent on pure Real-time Transport Protocol (RTP)/User Datagram Protocol (UDP) exchange, and does not include SIP messaging that would allow us to interconnect calls to PSTN. Where their system utilize SID and DID identification, our system automatically assigns IP addresses to clients, and this address is used to route voice and messages to the desired endpoint. By using a router and Structured Query Language (SQL) database on a Raspberry Pi we are able to store information on each user, and make this information available throughout the network.

2.1.4 Resilient D2D communication in emergency situations

The paper [17] introduce a viable alternative networking solution to increase network life-time, and devices energy consumption. This is done by proposing a strategy called Alert Dissemination Strategy (ADS) which enables the device to enter a sleeping state often, and only wake up in determined periods. By introducing two metric variables they are able to rank the individual devices based on neighboring connectability and remaining battery power. To achieve a more viable network they introduce beaconing and broadcasting in selected timeframes. The beaconing include the metric values, and are the base for computing which nodes are used to broadcast alert messages. A node with a high number of neighboring nodes and sufficient battery power will be selected over a node with less neighboring nodes. In the defined wake times, each node will listen for a broadcasted message before they go back to sleep, which reduce the possibility of all nodes receiving the message. This is a calculated risk, that is fully elucidated in the paper. Extensive testing of the system [17] has been done, which indicate that the introduction of the ADS result in a network that does not consume a lot of power, and does not congest the entire network with fully flooded messages.

The main difference between their research and ours lies in the network overhead produced in our system. Our system introduces much more overhead and consumes more energy resulting in a shorter lifespan of the entire network. But in contrast, our system enables multi-service including voice, SMS, picture and video, between connected devices. Further on each node in the ADS solution [17], need to be assigned a manually configured IP address, where our system is fully automated, and does only require that each client to enter their credentials before the application takes over and enables multi-service between all connected devices.

2.1.5 D2D communication with static IP address allocation

The first major research and implementation at UiA's of D2D using the OLSR protocol was done in a Master's thesis by two earlier students [40]. By using the existing Manet Manager application [34], they added support for RTP audio stream between the nodes in the D2D network within their application. An illustration of the application is showed in Figure 2.3. The application relied on manually mapping the nodes using static IP addresses so the nodes could reach each other. The drawback with this application is that the IP address is the only identification of the nodes, which does not make it user friendly. This has resulted in an GUI which is not very appealing, in regards of getting an overview of available nodes in the network. Still, their achievement worked as a proof of concept that D2D using ad hoc with OLSR on Android phones was possible with RTP audio transmission.

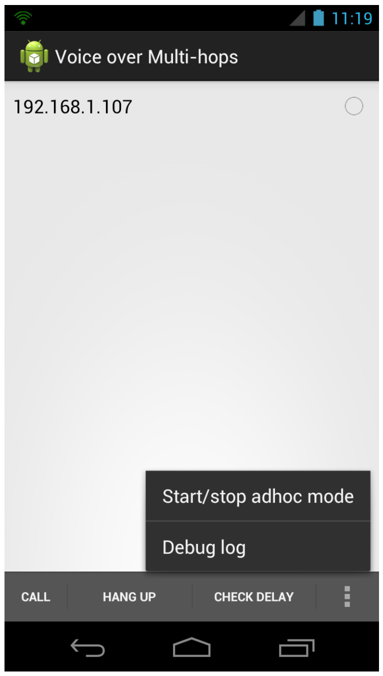


Figure 2.3: Main overview of previous Android application [40].

The students performed several tests such as max range for RTP conversations with single and multi-hop. They also made a separate application for performance evaluation that checked delay between the nodes. Our application has used some of the fundamentals of this Android application, but most features have been rewritten, mainly to provide more stability and better performance.

2.1.6 D2D communication with dynamic IP address allocation

After the previous Master's thesis project by the two earlier students, this project was done by one of the current author of this Master's thesis and another student. The main task of this project was to develop a dynamic IP address allocation to avoid manually mapping static IP address on every node. The solution relied on implementing a DNS server on the network as well as writing support on the client side to communicate with the DNS server. The DNS server was built from scratch using the following RFC papers [5, 7, 9, 21, 23, 24, 25, 26, 39]. By using the DNS server it was possible to identify the nodes with firstname and lastname by letting the DNS server contain the information on every node. In addition, the Android application got enhancements on the GUI to make it more appealing to use.

After testing of the system it worked to some extent. Still, there was several issues that was not taken into consideration, which afterwards resulted in conclusion that this implementations was not ideally to use in later implementation. One of the main reason was that primarily DNS servers are weak on security and could easily be misused [4].

Since we as students do not have any qualifications on security within DNS servers we decided that this implementation could lead to great misuse that could bring down the whole network if it was misused. Even though the implementation of the DNS was not continued, it provided us with useful knowledge that the D2D system could be enhanced in other ways, and that the system is generally quite capable to handle other types of enhancements.

2.1.7 D2D communication with enhanced secure services

This project [35] was performed by both authors in the fall of 2014. The main goal was to further extend the previous work done at UiA, where the Android application was redeveloped and employed with a user framework on a Linux computer. The system was developed to be scalable in order to support a huge number of users, both in the local backbone network and in foreign networks. This was achieved by employing a secure router entity in the network, which enabled clients to connect through a secure IP Security (IPsec) tunnel, providing integrity. Further work was done in the backbone network where we enabled clients to connect either through a pure OLSR network or through regular WiFi connection. The main focus here was to further extend the clients that would be able to connect to the network by doing so. The system was designed to be more commercializable than the previous system was, and thus the application should not be dependent of third party software that is reliant of $ROOT$ privileges on the respective device.

2.2 D2D Communication in Cellular Networks

D2D communication in general can enable devices to communicate directly with each other without any infrastructure. This would mean that any device is able to directly reach other devices in either direct contact or through multiple hops without the need of APs or BSs. In the cellular communication paradigm this feature is seen as promising and would enable large networks to offload unnecessary traffic through direct communication or to extend coverage to devices outside coverage of the BS. We talk about three types of D2D communication in general, infrastructured, infrastructureless and hybrid. Figure 2.4 illustrate the three types of operation, where "Inband" illustrates infrastructured and hybrid communication in the cellular band, while "Outband" shows infrastructureless in the Industrial, Scientific and Medical (ISM) radio bands. Infrastructured D2D communication would mean that the serving BS would allocate resources on connected devices to other connected devices, enabling D2D communication to offload traffic to the serving BS. In the infrastructureless method, devices would share their resources with other devices on a self-determined basis, without the control of serving an BS. For the last method, both the device and an BS would cooperate to serve other devices with bandwidth, either through D2D communication or in direct contact with an BS.

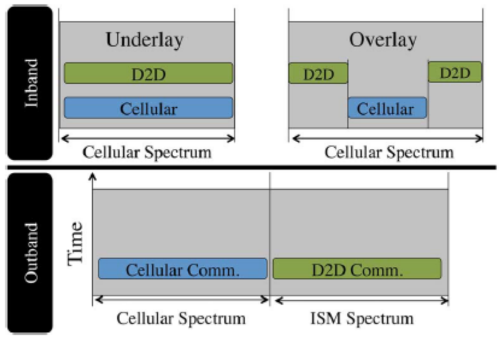


Figure 2.4: Schematic representation of inband and outband D2D [1].

2.3 Android Smartphones

The Android platform has been owned and developed by Google from the year of 2005 [2]. The mobile platform is currently the most popular and growing platform on world basis [11]. The platform has in the past few years also expanded to different platforms such as TV, Internet of Things (IoT) and embedded devices. One reason why Android has become so popular, is the large market of third party applications that is being developed. Google has done a tremendous amount of work by offering wide support for application development. Tools such as Eclipse with Android Software Development Kit (SDK) and Android Studio is often used for application development. In addition, the Android platform is easy to provide *ROOT* privileges, which for many developers is very useful. This means that one can access *ROOT* folders, and *ROOT* privileged hardware which is normally not accessible. Still, this has both positive and negative impact. If a phone has *ROOT* privileges and is infected with some kind of malware, the attacker could have complete control over the phone and can basically do what it wants. Yet, by giving *ROOT* to the platform it offers and provides the user and developer almost full functionality to do what it wants with the phone. This has been some of the reasons why people chose the Android platform over IOS platform which is quite restricted.

2.4 Wireless Ad Hoc Networks and OLSR Daemon

A wireless ad hoc network is a network that does not rely on infrastructure to operate. Each node has equal status in the network, and data is forwarded by relying to and from other nodes. The network itself does not have a dependency of routing entities like routers and APs. In addition to regular routing, the network can utilize flooding in order to reach each core of the interconnected network. In wireless ad hoc network we have different routing protocols that could be used. One of the most commonly used is the OLSR protocol, where each node build up a routing table of the topology in the connected network, and regularly sends out information that enables each node to update their routing tables. A different protocol that could be used is the Ad hoc On-Demand Distance Vector Routing (AODV) protocol, where each node does not have complete knowledge of the topology of the network, and floods out messages to establish routes when they are needed.

The OLSR daemon is an implementation of the ad hoc routing protocol OLSR, enabling ad hoc routing of a various number of system. It is available through the `olsr.org` web page, and is currently supporting devices running Android, FreeBSD, Windows, Linux and more. The daemon provides clients the opportunity to route traffic directly between clients or through multiple hops, as clients are acting as relays of messages. OLSR is a pro-active routing protocol, which means that each device running OLSR, need to send out messages to build up routing tables, before messages can be sent. This makes the power consumption on devices running this protocol higher than for other protocols, but it also provides a robust backbone network that is able to handle moving nodes quite well.

2.5 Tools

In this section we present the most important tools that have been used for development and experiments.

2.5.1 OpenWRT

OpenWRT is an alternative open source Operating System (OS) for routers, built from the ground based on Linux. It enables multiple features through a package management system, and offers a fully writable filesystem. This allows for a complete configurable router, which can be tailored for the need of the system it provides service to. Thus making the OS highly flexible and fully customizable.

2.5.2 LAMP

The Linux, Apache, MySQL and PHP (LAMP) package is a collection of open source packages that enable web interactiveness on Linux machines. In our system we use a Debian Wheezy port for Raspberry Pi, which is customized and optimized for the embedded system. The LAMP package collection consists of three main entities, namely an Apache web server, a MySQL database server, and packages to enable use of the server side scripting language Personal Home Page (PHP).

Apache web server

The Apache web server package for Linux is one of the most commonly used web servers [3, 27]. It offers a robust, commercial-grade, feature-full and open source web server, that can be fully customized for each use, and since it is part of the Apache Software Foundation there are numerous of users which is contributing ideas, code and documentation to the project. In our project the Apache server acts as the basis for communication between each device and the MySQL database, with corresponding PHP scripts that enable access to data.

MySQL

The MySQL database package for Linux is one of the most popular relational databases, and it offers superior speed, reliability and ease of use. In order to make it easier to handle the databases, one could install second hand software like phpMyAdmin, since the basic MySQL server does not provide a graphical interface. It is fully accessible through the terminal, with the use of SQL syntax.

PHP

The server side scripting language PHP is designed to provide web development on the server, but it could also be used as a general programming language. The basis of the language is to provide capabilities to clients in the form of generating web pages, and interacting through Hypertext Transfer Protocol (HTTP) forms. When using PHP the entire code is hidden from the user, and only code that the developer want the client to access is available. This make it ideal when talking to databases as the developer can fully control what the client is able to retrieve. Other languages like Active Server Page (ASP) could of course be used, and might also be more beneficial in large systems, different from ours. The main reason for choosing PHP over other languages is that it is easy to use, and performs fast and reliable.

2.5.3 phpMyAdmin

phpMyAdmin is a freeware software that is able to provide an intuitive web interface of different databases. It currently support MySQL, MariaDB and Drizzel, and is written in PHP. The software makes it easy to handle most database management operations in a easy and intuitive manner. Operations like creating databases, tables, columns, relations, indexes, users, permissions, etc. are available through the web interface with great descriptions.

2.5.4 Eclipse

Eclipse is a free and open source Integrated Development Environment (IDE) that offers the possibility to program in a various number of languages. Some of the available language packages are, XML, Java, PHP, Perl, JavaScript, Ruby, etc. This is accomplished by the native support for multiple plugins and SDKs. In our situation it is used to develop the Android application through the Android SDK provided by Google. In addition, Eclipse enables support for debugging which is crucial when developing. This makes the developer able to trace the program and also provides the opportunity to check if the application behaves as intended.

2.5.5 ADB

In order to aid development of Android application, Google has made available a Android Debug Bridge (ADB) tool which is very useful. It allows the developer to get shell access to either an emulated or connected Android phone, where numerous actions can be performed. One of the most popular feature is to check how the private folder stores its *sharedPerferences* and *SQLite* database files on the Android platform. By utilizing the ADB tool, the developer can provide itself *ROOT* access and perform database lookups with regular Linux commands. One other useful feature is that the developer can push Android Application Package (APK) files, and install them on the phone remotely, without having to go through the Google Play Store.

2.5.6 Logcat

The most efficient way for an Android developer to check and verify that the code works as intended is by debugging. By debugging the developer is able to optimize performance, stability and behaviour of the application. To debug Android applications a tool called *logcat* is often used. To populate logs to *logcat* there are two main methods called *system.out.print* and *log*. The *log* function has possibilities to log as Error, Warning, Verbose, Info and Debug type, whereas the *system.out.print* function prints out log just Information to Logcat.

2.5.7 Wireshark

Wireshark is a powerful network protocol analyzer that is capable of capturing all kinds of network traffic. This tool is very useful in order to validate and verify where traffic is being requested and received. It can read the content in every packet, which makes it extremely useful for testing and validation. It supports reading *PCAP* formats, and works perfectly with *Tcpdump* [37]. Wireshark offers easy and quick filters so one can e.g. filter on only HTTP traffic.

Chapter 3

Design of the Current Solution

This chapter include our work performed on how the system should operate and what requirements we have. In addition we provide an overview of the design of current network solution with the three components in our system. Further we explain the efforts to design these constituent components, from the enabling embedded system, Android application and finally the design of the prototype case with accessories.

3.1 Initial Plan

To provide grounds for improvement we began to evaluate the previous work done on the system, where we outlined the main parts we saw as vital for creating a network platform capable of providing seamless communication in crisis situations. With this information we needed to evaluate what we would be able to accomplish within the time schedule.

The system should be stable, autonomous and provide network coverage outside the bounds of enabling APs, and design a platform that would make it easy to implement new future services. We decided to adopt the features of implementing a framework for handling of users, and needed a way of assigning IPs to connected clients. The system should also be portable so we needed to keep the energy consumption low, resulting in finding an embedded system that would provide us with a platform for doing so. In addition we needed a power source capable of providing enough energy to power the system for several hours. The system should provide voice and text messaging service, but it should not be hard to implement services like MMS and video in the future. On the client side there should just be one Android application, not two like the previous project required, this alone meant that the application needed to be totally redesigned. In the following list we provide a clear overview of the system requirement for the designed solution.

- One single Android application which include both ad hoc mode initialization and services
- User friendly interface which automatically updates connected nodes
- Provide multi-services including voice, text, picture and video
- Employ an embedded system to provide automatically assignment of IP address, and user maintenance
- Design and implement a prototype system including Android smartphones, wireless routers and Raspberry Pi which is operational without the need for AC power

3.2 System Design Overview

In order to provide a light overview of the current network solution, we created a design overview illustrated in Figure 3.1. Here we introduce the main parts of the system. The core network consist of two OLSR enabled APs which are meant to extend the coverage area of the network. The Raspberry Pi provides the connected clients with information that would enable them to automatically be assigned an IP address, as well as all the necessary information needed to enable SMS, MMS, video and voice communication. Clients connected to the network should be able to extend the coverage of the network to neighbouring clients, by utilizing the OLSR protocol, and all traffic should be able to be relayed to the designated source. The main idea of the network is to form a solid backbone network that would enable clients to move freely and still be able to make voice calls and send SMS to each other. The number of OLSR routers could be increased, and by placing them at strategic places around the desired working area, provide seamless connectability to the clients.

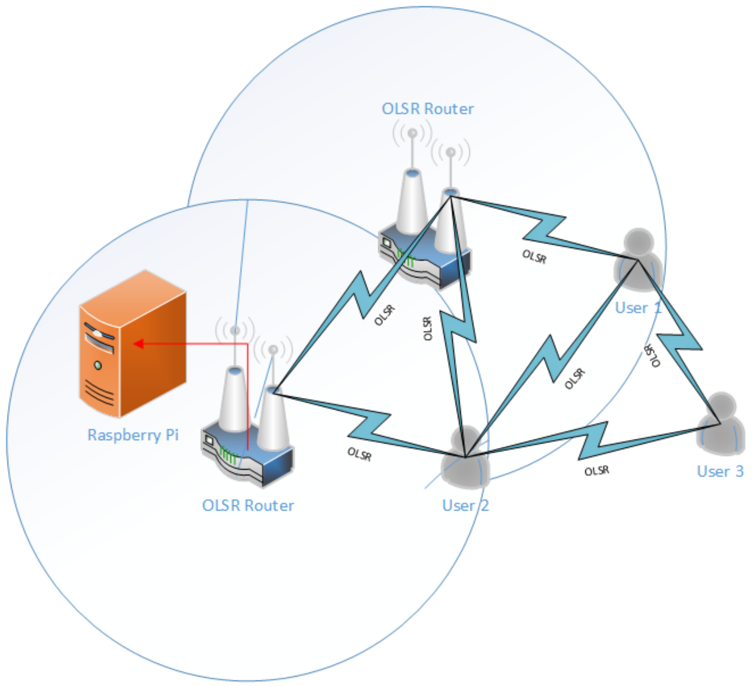


Figure 3.1: System design, illustrating that clients could be connected to the local backbone network via one or multiple hops to extend the network connectivity.

3.3 Design of the Constituent Components

3.3.1 Embedded system

In the network we needed an embedded system that were able to provide all the necessary services, and still be lightweight enough to be operational on battery power. After doing some research we decided to go for the Raspberry Pi model B+ that offers a fully configurable Linux distribution and power consumptions as low as about 250 mA in ordinary operation. By utilizing the Raspberry Pi we are able to set up a web server that could provide access to different scripts that would enable us to provide hybrid IP address allocation and other services like user handling.

3.3.2 Android smartphones

To reach a broader audience that can easily understand and manage the D2D application, we have focused on making the application have a user friendly GUI. We have tried to make the application as self-understanding as possible by using elements such as *listview, navigation-drawer, dialogs, notifications*, etc. In this section, several illustrations from the application User Interface (UI) will be presented with a brief explanation.

Core design evolution

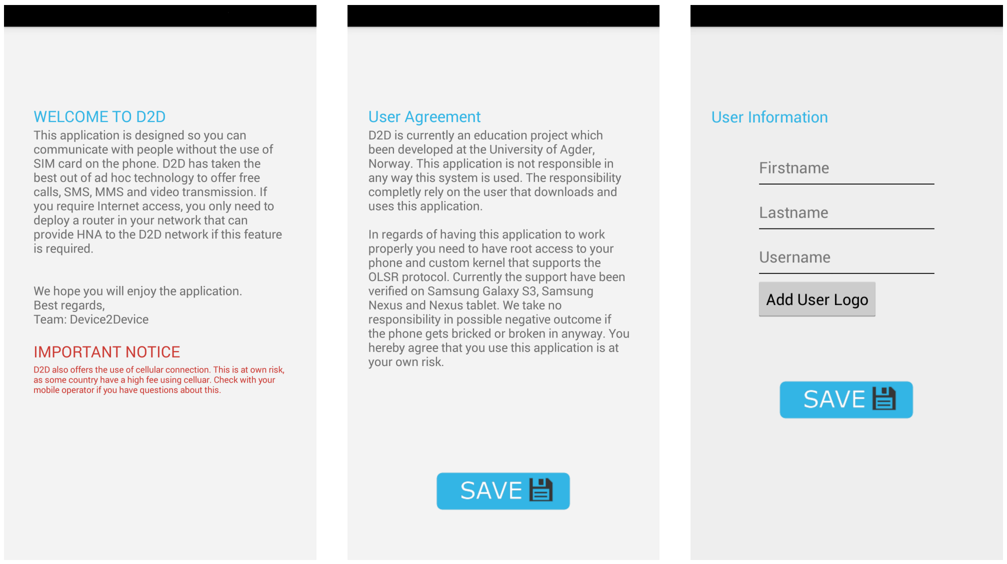
Since this project is based on continuously work that has been made from three previously applications, we were to some extent limited to have completely control of the design. One design limitation is the core functionality of the OLSR. Since our application uses core features from the *Manet Manager* application [34], we have to use the existent functionality which the *Manet Manager* provides. One of the main issues with the design of the OLSR service, is whenever changes are made to the service, it must be restarted. In addition there is no good method to know when the service is again in *ready state*. In Section 4.4, we will explain a workaround that we have discovered and is currently using to make the application work as intended.

One of the major design improvement that we have accomplished in this project is by only having one application that is implemented with the OLSR service, UI, media services and registration to server. From the previously Master's thesis, the application was dependent on the *Manet Manager* as an own application on the devices. Our application is currently not dependent on having this separate application, since we have integrated most of it into our own application. There are both positive and negative effects with this merging. The positive is that we have everything in one application, and we have more control over the OLSR service. The negative effect is that the complexity of the application has increased quite much. The general workload for the application has also increased, since it now handles everything. This has led us configure the Android application to extend the Random Access Memory (RAM) allocation, known as grow heap [12].

It should also be mentioned that the *call* functionality is not a part of our design and implementation. This feature has only been modified and tweaked to work in the new application. Even though we are dependent on some feature from the *Manet Manager* and the previous Master's thesis, we have implemented several new features that has allowed us to come up with a design which we believe is quite unique and has not yet been done. As of today we have not seen any other application that offers the same friendly UI as well as the nthatumber of possible media opportunities to communicate with other nodes on the D2D network.

User Interface

The illustrations when the application is started for the first time, and enters the setup is shown in Figure 3.2. The figure shows a welcome screen, user agreement and user registration. After the user selects the button *Save* from the last image shown in Figure 3.2 the application will start by initializing the OLSR service and register to the server. In Figure 3.3, we can see these illustrations. The application goes through three main steps before entering the main class. The Figure are quite self explanatory on what the application is doing in order to go to the next step.



WELCOME TO D2D

This application is designed so you can communicate with people without the use of SIM card on the phone. D2D has taken the best out of ad hoc technology to offer free calls, SMS, MMS and video transmission. If you require Internet access, you only need to deploy a router in your network that can provide HNA to the D2D network if this feature is required.

We hope you will enjoy the application.
Best regards,
Team: Device2Device

IMPORTANT NOTICE

D2D also offers the use of cellular connection. This is at own risk, as some country have a high fee using celluar. Check with your mobile operator if you have questions about this.

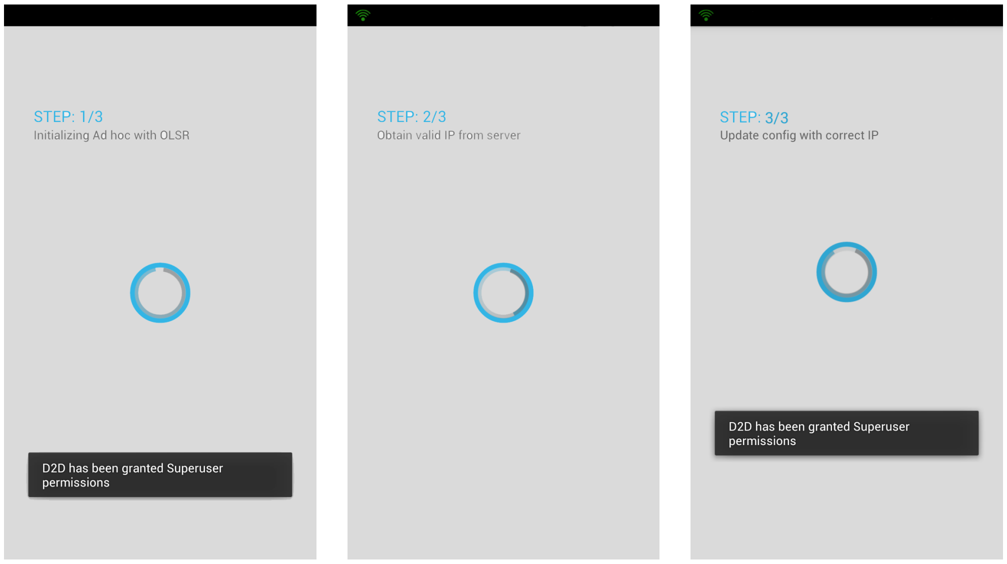
User Agreement

D2D is currently an education project which been developed at the University of Agder, Norway. This application is not responsible in any way this system is used. The responsibility completly rely on the user that downloads and uses this application.

In regards of having this application to work properly you need to have root access to your phone and custom kernel that supports the OLSR protocol. Currently the support have been verified on Samsung Galaxy S3, Samsung Nexus and Nexus tablet. We take no responsibility in possible negative outcome if the phone gets bricked or broken in anyway. You hereby agree that you use this application is at your own risk.

User Information

Figure 3.2: Android setup part 1.



STEP: 1/3
Initializing Ad hoc with OLSR

D2D has been granted Superuser permissions

STEP: 2/3
Obtain valid IP from server

STEP: 3/3
Update config with correct IP

D2D has been granted Superuser permissions

Figure 3.3: Android setup part 2.

When the application has reached step *3/3* from the Figures 3.3 it enters the main class. The main class UI is illustrated in Figures 3.4. The illustrations shows the *navigation-drawer* with different options such as start, stop adhoc, routing info, about, settings and exit. The second figure shows several users that is currently active in the D2D network. The last illustration shows when a user is selected from the listview. The user can then decide to send message, video or call the other person.

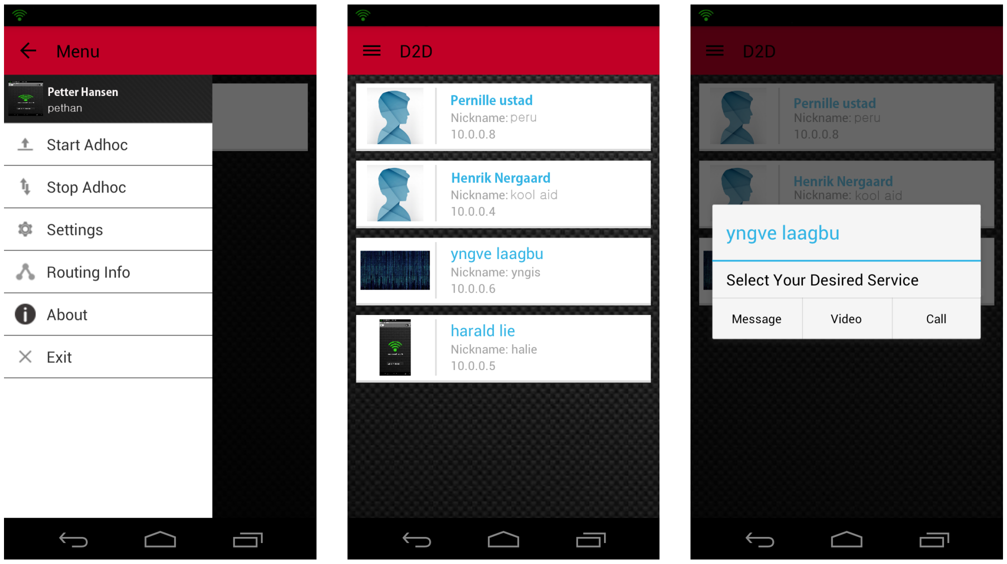


Figure 3.4: Android main class overview.

3.3.3 System prototype

In order to have a workable prototype we designed two cases which include a battery pack, a TP-Link router and an enabling Raspberry Pi. This section describe the efforts done to design the two prototype cases. As well as including sketches of the designed hardware boxes.

Water proof case for assembly

To hold all the required hardware, we required two cases that were robust and could withstand some ruff treatment. We found suitable cases at Clas Ohlson [6]. The cases are illustrated in Figure 3.5. They should be able to protect the hardware and battery pack from the environment, and they should also be waterproof which is a nice feature for crisis situations where the weather could change.

Figure 3.5: Water proof prototype case, 33x28x12 cm [6].

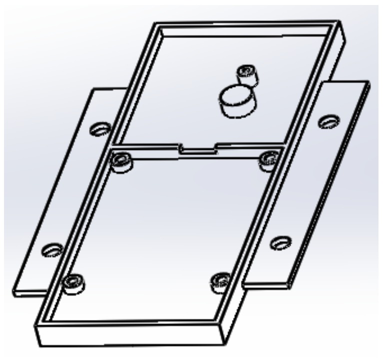
Battery power

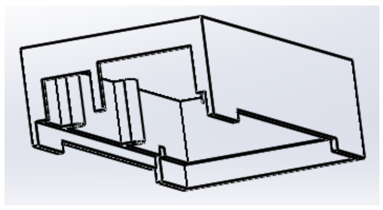
To operate the backbone system we need a battery pack that is able to power the system for several hours. By reading the specification sheet on the main entities of the system cases, we found out that the Raspberry Pi drain about 0.25 A [31] whilst the router drain about 1.5 A [38]. This result in a total of 1.75 A for the complete backbone system. The power pack we have chosen are equipped with 26 Ah lithium battery pack. With these numbers provided we calculated that the battery pack should be able to operate the system for about 15 hours. The chosen battery pack is illustrated in Figure 3.6.

Figure 3.6: Intocircuit 26000mAh battery pack [15].

Design of hardware boxes

In order to assemble the hardware in the prototype cases, we designed custom hardware boxes. The Raspberry Pi does not come with any protecting box, and we therefore designed a box that would enable us to mount the Raspberry Pi. Figure 3.7 illustrate the case that was designed, and from it we see the two parts that need to be assembled around the Raspberry Pi, as well as an illustration of the assembled box. Furthermore we have the TP-Link router that we dismounted from its original box. This is done to save space, since the original case takes up more space than the hardware board itself. Figure 3.8 illustrates the designed TP-Link box, from the two separate parts to the assembled case. Both boxes were designed to fit in the assembly case and therefore they are designed to be mountable, as illustrated in Figure 3.9.


(a) Bottom


(b) Top

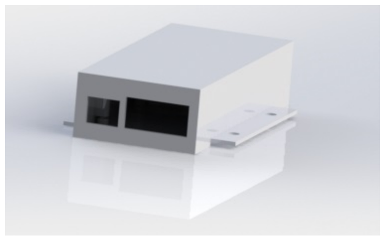
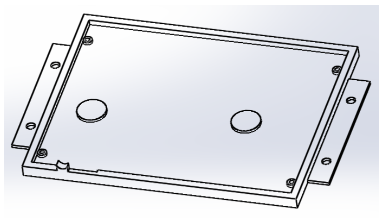
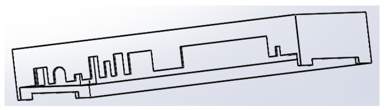

(c) Assembled

Figure 3.7: Design of box for Raspberry Pi.


(a) Bottom


(b) Top

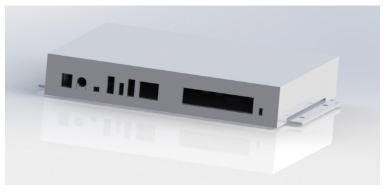

(c) Assembled

Figure 3.8: Design of box for TP-Link router.

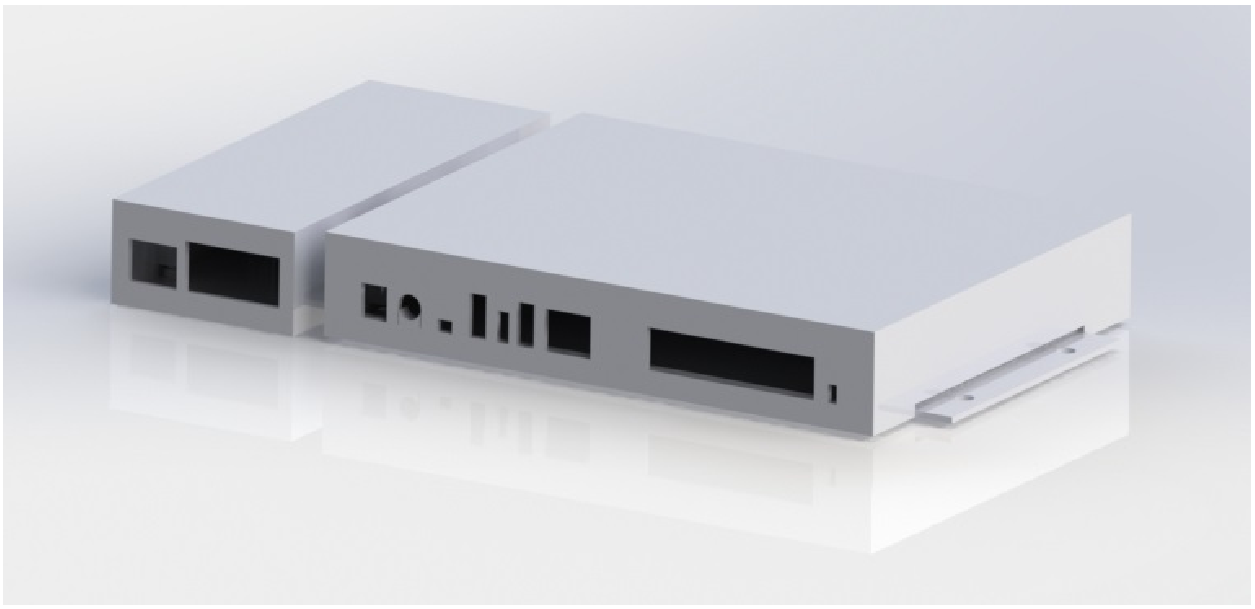


Figure 3.9: Assembled box of both TP-link and Raspberry Pi.

3.4 Chapter Summary

In this chapter we have presented our initial planning that include the requirements that we have set for our system. As well as an overview on the design of the current network solution that emphasises that we have three main entities in our system. From the enabling embedded system, extending TP-Link routers and the Android smartphones running our developed application. Further on we have explained the design of these constituent entities, from the selection of a embedded system, to the design of the Android application and finally the design of the prototype cases with accessories.

Chapter 4

Implementation

This chapter include what has been developed, implemented and initially provides a overview of the network topology with possible extension. In Section 4.2 we describe how the OLSR routers are implemented into the system. Further in Section 4.3 we elaborate on how the Raspberry Pi and corresponding services are developed and implemented. Next in Section 4.4 we explain how the Android application is developed and how it interact in correspondence with the user framework described in Section 4.3. Finally in Section 4.5 we illustrate the prototype system. Furthermore the different implementation aspects are described with strong relation to the appendices.

4.1 Network Topology

The network topology is shown in Figure 4.1, where we illustrate the three main entities of the network. First we have the OLSR routers that provide a stable backbone network for the clients. Connected to one of the routers we have the Raspberry Pi, which act as a centralized database of connected clients. Here we make it possible for the connected clients to retrieve all the information needed in order to connect calls and exchange messages, as well as providing the clients to be automatically assigned with an IP address. This will be further explained in the following sections. The network consists of these three main entities, but the Raspberry Pi makes it possible to connect a fourth entity. This could for instance be a 3G/4G network adapter that would provide the network with Internet connectivity. This feature is not adapted by us at this stage and is purely meant as a possible extension of the topology. Since this project involves having a portable network, the OLSR routers would be placed at strategic places around the area of operations, enabling the clients to move freely within the designated area, and further extend access to other clients moving outside of the backbone coverage area through D2D communication.

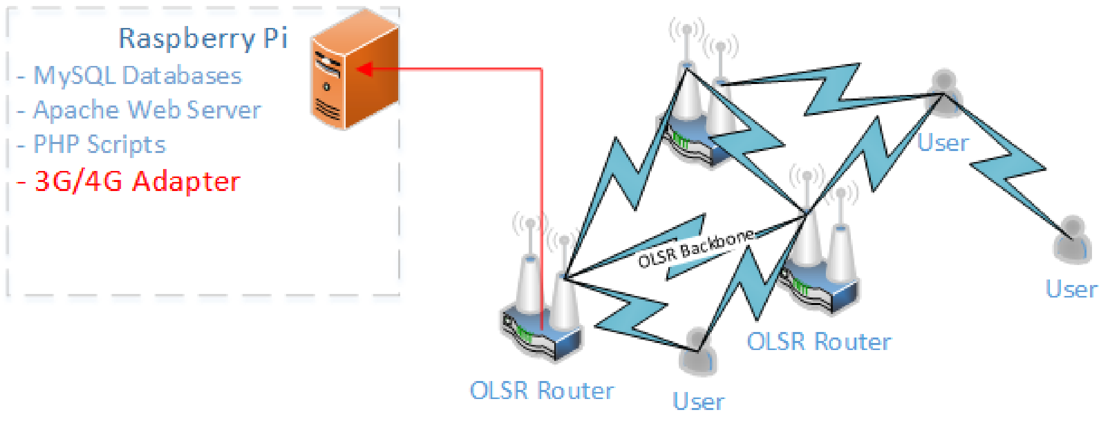


Figure 4.1: Network topology illustrating the different aspects of the local backbone network, where clients are able to connect through the OLSR backbone or through multiple hops.

4.2 OLSR Routers

Figure 4.2: TP-Link WDR3600 [20].

The OLSR enabled backbone routers are set up using a custom router OS named OpenWRT. This is done due to the flexibility using such custom firmwares on the routers. In order to provide the OLSR routing capabilities on the routers, several packages needed to be installed on the routers. By installing these packages the routers are further configured to act as extenders of the network, and to provide a stable backbone network for connected clients. To provide a portable router setup, we designed two cases that include battery power that would enable the routers to be operational for up to 15 hours. The design of these cases are further explained in Section 4.5

The setup are done utilizing two TP Link routers viewed in Figure 4.2. These routers are capable of providing multiple Service Set Identifier (SSID) on one interface, and operation on multiple antennas simultaneously. Due to the power consumptions utilizing this feature, the routers are manually configured to only provide access on one antenna with one SSID. Connected to one of the routers we have a Raspberry Pi that supplement the routers with access to vital information. In order to know the IP of the Raspberry Pi we configured a static IP lease on the designated Dynamic Host Configuration Protocol (DHCP) server running on one of the routers. The setup process of the routers are more detaily explained in Appendix A.2.

4.3 Raspberry Pi based Hybrid IP Allocation Server

Figure 4.3: Raspberry Pi B+ [16].

In order to provide services through the local network we set up a Raspberry PI B+, running a Linux Debian Wheezy port especially compiled for the respective device. The choice of using the model B+ over other models is due to the power consumption and available Local Area Network (LAN) interface. We set the OS up without any GUI as we don't need it and to keep the power consumption as low as possible. Furthermore we configured the OS to enable Secure Shell (SSH) and installed the LAMP package collection. To decrease the power consumption even more we configured the Raspberry Pi to automatically obtain an IP address from a DHCP server and disabled the High-Definition Multimedia Interface (HDMI) on the device. This should improve the overall power consumption and make the installation of the Raspberry Pi more suitable for battery power. The installation of the Raspberry Pi is further explained in Appendix A.1.

4.3.1 MySQL user database

In order to store information about users in the system, we set up a MySQL database included in the LAMP collection. We could also have chosen to use a SQLite database, but to provide more flexibility and ease we decided to use MySQL instead. The database is only accessible through the local interface on the Raspberry Pi. For ease when configuring the database we utilized the phpMyAdmin tool described in Chapter 2. The database has been deployed to use the following table-fields: *name, lastname, image, IP and expiration of the IP*. To handle all traffic to and from the database we have written several PHP scripts which make it easier when developing the framework. This enable us to make all information available through simple HTTP requests. The setup process of the database is further explained in Appendix A.1.3

4.3.2 Apache web server

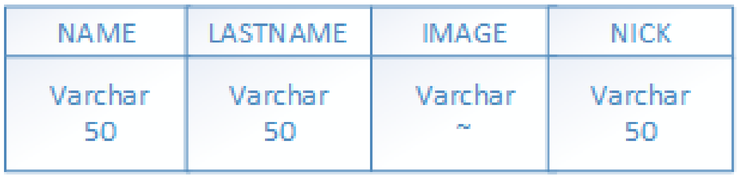
To provide access to the PHP user framework, we utilized the Apache web server included in the LAMP package collection. When installing the Apache server, there are standardized configuration available, but for our purpose we needed to modify some configuration ourselves. As we neglect the security of the connection between the client and server, we did not configure Hypertext Transfer Protocol Secure (HTTPS). We only allow access to the server in regular HTTP requests, which would mean that all information is transferred in clear text. Since the information available on the server is not sensitive, we do not see this as an vital issue. The configurations and complete install process is further explained in Appendix A.1.4

4.3.3 PHP user framework

In order to provide an easy user framework we developed multiple PHP scripts, that enables interconnection to the MySQL database. Here we enable access to all information stored in the database by processing requests to the respective scripts. All scripts are included in Appendix C.1. In the following paragraphs the scripts are explained.

Add a new user

In order to provide a framework for registry of new user in the system, we developed a PHP script that takes the input as illustrated in Figure 4.4. The script collects all the necessary information to complete the registry process and require that all fields is defined. The script first check that the supplied nickname does not already exist in the database. If it does not, it moves on to store the respective image with the users nickname in a designated image directory. Furthermore, putting the hyperlink to the image, along with all the other information in the MySQL database. If all succeeds the scripts return 200 OK, to indicate that all is done correctly. Response codes are further explained in the last paragraph.



NAME	LASTNAME	IMAGE	NICK
Varchar 50	Varchar 50	Varchar ~	Varchar 50

Figure 4.4: The add new user framework: Registering new user to the database.

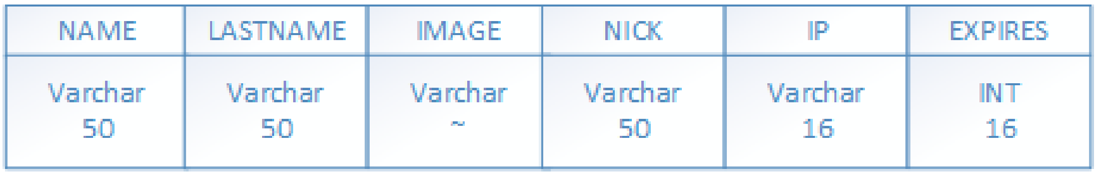
Update IP address

The update IP address script is the most important script we have developed. It is the basis of the hybrid IP allocation mechanism in the system, as it provides the client with a leased IP address. It only require one input, which is the clients current IP address. The script either responds with a 200 OK message which indicate that the client can keep using the same address. Otherwise it returns a new IP address which the client should switch to. This is done in the following way:

- Check whether the supplied IP address ends with 254.
    - If true: Check the database which IP address can be provided to the client, and update the database.
    - Return the respective IP address to the client.
- If the IP address does not end with 254.
    - Check whether the supplied IP address is still valid.
        * If true: Update database.
        * Return 200 OK, letting the client know that it can still be used.
    - If IP address is not valid.
        * Check which IP address can be provided to the client, and update the database.
        * Return the respective IP address to the client.

Get user, by IP or nickname

The system needed a way to retrieve all the information of a respective user, so we wrote two scripts that are able to get this data. The first script takes an IP address as input, then searches the database for the row that contains the specific IP and returns the entire row to the user. The same goes for the script that takes a nickname as input, only here we search for the row that contains the respective nickname. If the requested user is not found or an unexpected error occur, the script will return an error code which is further explained in the last paragraph. The script produce the output as illustrated in Figure 4.5, which is the entire dataset of a user.



NAME	LASTNAME	IMAGE	NICK	IP	EXPIRES
Varchar 50	Varchar 50	Varchar ~	Varchar 50	Varchar 16	INT 16

Figure 4.5: The get user framework response: An array of all the available data.

Update Image

After the first setup process there should be a method to change the specific thumbnail picture of any user. Thus we created a script that takes input as illustrated in Figure 4.6, where we collect the respective image and assigns it to the specified nickname. The old image will be replaced, but we do not verify that the picture is uploaded by a validated user. This can in worst case be exploited, but since the system at the current state is defined as a prototype we accept this flaw.

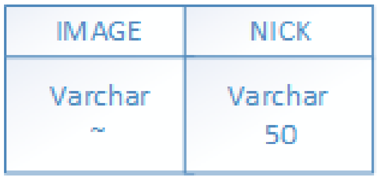


IMAGE	NICK
Varchar ~	Varchar 50

Figure 4.6: The update image framework: For pushing up a new user image to the database.

Check if nick exists

Before the initial setup process the user specify a nickname it want to assign it's respective device to. To avoid problems in this process due to a nickname already being used, we wrote a simple script where the application may check the validity of the entered nickname before the initial add user script are being executed. The script takes a nickname as input and checks the database for a similar nickname, if there is not found a similar nickname there will be a 404 response, else a 409 response as described in the next subsubsection.

Error code handling

Most of the PHP scripts do not require much feedback after the request or update is completed. Therefore we used the standardization of HTML error codes into our scripts. In order to provide ease we have used the following error codes:

- 200 OK
- 404 Not found
- 409 Conflict
- 500 Internal server error

4.3.4 Testing and clean-up scripts

Clean up expired IP addresses

When IP addresses expire we need a swift way of removing them from the database, so we implemented a script that runs in the background of the Raspberry Pi OS once every hour. This removes the IP address entries and sets the expire field to zero. This is done to optimize the IP allocation process as we only allocate addresses that are not present in the database already.

Calculate Raspberry Pi uptime

The Raspberry Pi is powered by a battery pack, and this introduces a problem of determining how long the Raspberry Pi is able to operate along side with the OLSR router. Therefore we developed a simple bash script that writes down how long the system has been operating to a file. For testing purposes we run the system until the battery is empty, then power it up again and check the log file. This is further explained in Section C.2.

Stress test add user

In order to validate that the system is scaleable we developed a script to generate randomized users, and calculate the latency of each iteration. The script can be configured to add as many users as we see fit, and the result of the latency are further explained in Section 5.5.

Stress test get user by nickname

After generating a lot of users, we implemented a script that calculates how fast we are able to get information of a specific user. This script collects data of a pre-configured user and calculate the latency. In Section 5.5 we utilize this script to see how fast we are able to collect data when the database is large.

Asynchron stress tests

The two previous tests are synchronous and would not stress the system enough. Therefore we developed a different script that spawns many instances. The scripts task is simply to execute separate instances of the previous scripts, resulting in a multiple instance of each script running in parallel and competing for resources, thus slowing down the system and putting a lot of stress on it.

4.4 Android Application

To explain the functionality and implementation on the Android application will be explained. Due to the large amount of features that this application uses, this section will explain the major key features which will be divided into topics to make it easier to comprehend. In Appendix D we have gathered some general information about the Android application code. Information such as *total lines of codes, number of Java classes* etc. is presented.

In this section we will first start with an explanation where the setup of the application using the hybrid IP allocation feature will take part. Further we go to explaining the *listview* capabilities and functionalities. Furthermore we will show how the application is able to send SMS, MMS, video and make call conversations between each other. Every topic will be explained using activity diagrams. The diagrams are simplified but shows the overall actions that the application performs.

4.4.1 Android initialization

In order to explain the Android setup phase we have divided the activity diagram illustrations into two parts. The first part mainly shows the initialization and general actions that is performed on the application. The second part is a combination where it will register the user and obtain a valid IP before entering the *MainActivity* class.

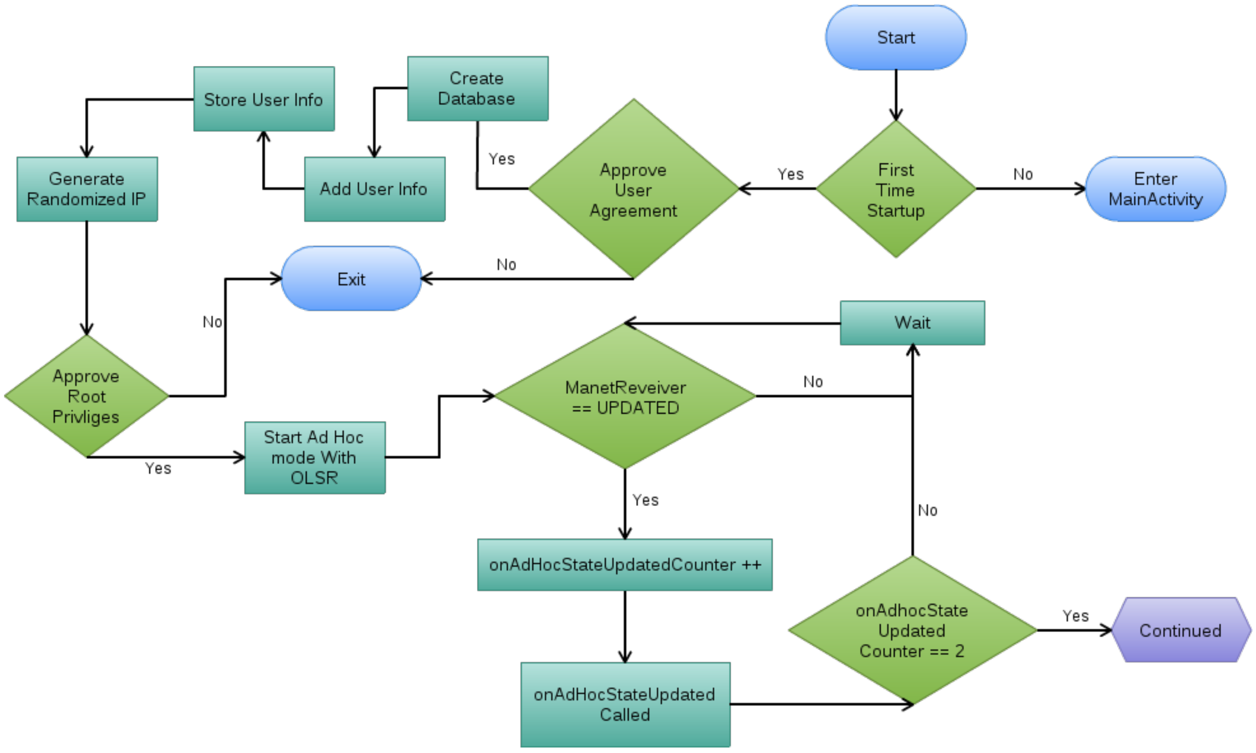


Figure 4.7: Activity diagram: Part 1 overview of the application setup process.

From Figure 4.7, it checks if the user has installed the Android D2D application. This is measured by using a boolean that is set to *true* if the application has been started. If not, the boolean is default set to *false*, and the user will be sent to the *MainActivity* class. Since the user is running the application for the first time, it will be prompted with a welcome menu, where the user also must agree and acknowledge the user-agreement of the application. Once this is done, the application creates a database with two tables. The first table is called *User*, where the user of the application will be stored. The second table is called *Friends*, where the users of the D2D network will be stored. After these two tables has been created, the user must enter its firstname, lastname, nickname and select a logo from its storage. If the user do not want to select any logo, the application is set to use a default logo that will be used. The default logo is the same for every user that do not want to use its own.

When the user information is added and stored into the *User* table, the application will generate the temporary randomized IP. Now the application has enough information about the user and is ready to initialize the OLSR. In order to install the OLSR service to the application, we need to have *ROOT* privileges. This must be approved, otherwise the OLSR will not be installed and the application will not perform as intended. When the user has accepted, the OLSR initialization will start.

Since there is no function that tells us when the OLSR service is ready, we have found a method around this by listening for the *onAdhocStateUpdated*. We figured out that *onAdhocStateUpdated* is initialized two times when the OLSR is started or stopped. Since we know that the *onAdhocStateUpdated* must be runned twice we use a counter to verify when it is equal to 2. When this state is true, we know that the OLSR state is ready and we can go on to the next state in our application.

In the second part we can see that the Figure 4.8 continues from the first Figure 4.7. Since the first Figure showed that the OLSR was in ready state we now insert the randomized IP into the OLSR configuration. In order for the OLSR service to use the randomized IP, we need to restart it. Before we can go further we must know when the OLSR is in ready state again. As explained from the first part we must wait for the *onAdhocStateUpdated* . Since we appended the counter to 2 in the first part, we must now wait for the counter to be equivalent to 4.

When the *onAdhocStateUpdateCounter* is equal to 4, we know that the OLSR is in ready state and using the randomized IP. Now the application will start by collecting the user info from the database and send an HTTP POST request to the server. If the server received the data and approved our request it will reply back with an *200OK*. If the server did not approve the request it will reply back with an error code. These code has been implemented in the Android application so it is easy for us to debug and find what kind of error the server had. The Android management of HTTP responses is demonstrated in Listing B.2.

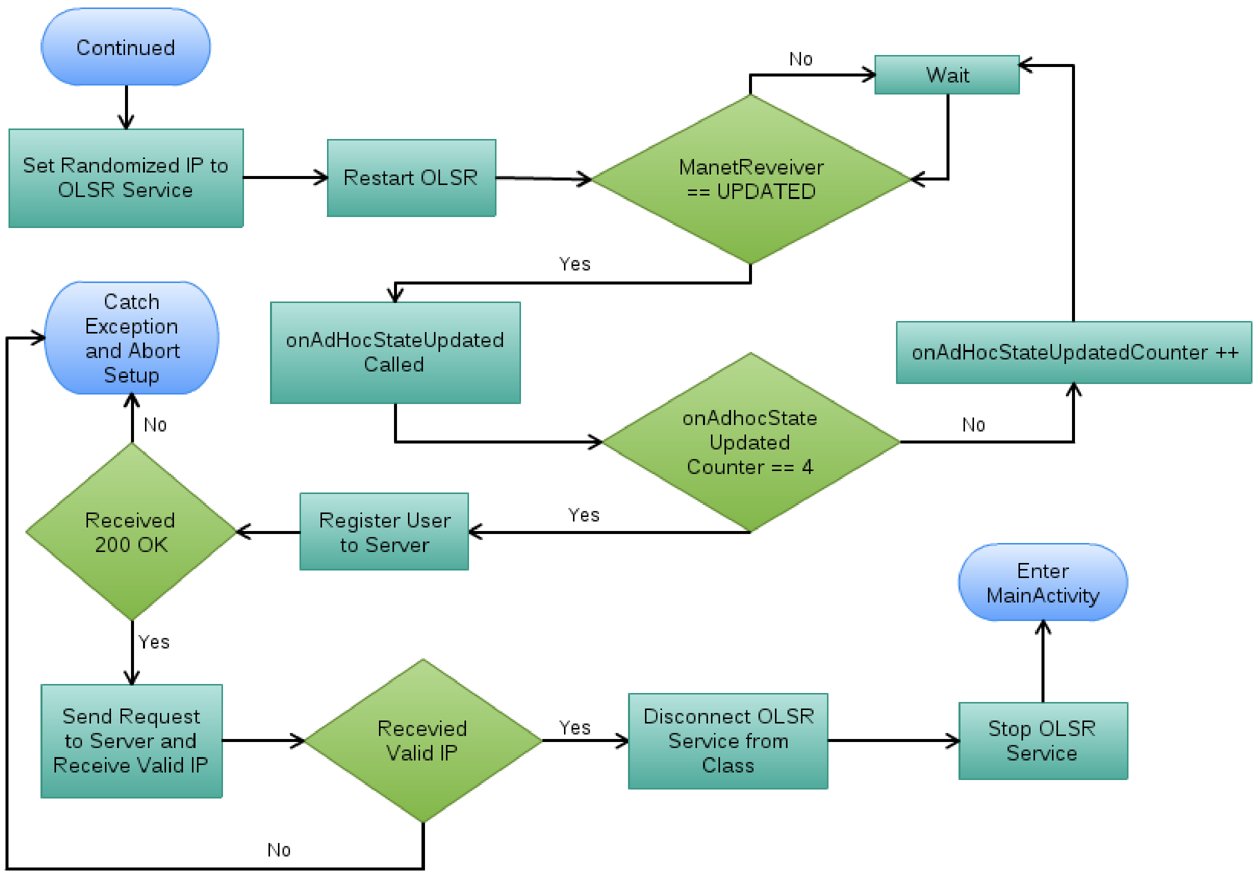

Continued
Set Randomized IP to OLSR Service
Restart OLSR
ManetReveiver == UPDATED
No
Wait
Yes
onAdHocStateUpdated Called
onAdhocState Updated Counter == 4
No
onAdHocStateUpdatedCounter ++
Yes
Register User to Server
Received 200 OK
No
Catch Exception and Abort Setup
Yes
Send Request to Server and Receive Valid IP
Recevied Valid IP
Yes
No
Disconnect OLSR Service from Class
Stop OLSR Service
Enter MainActivity

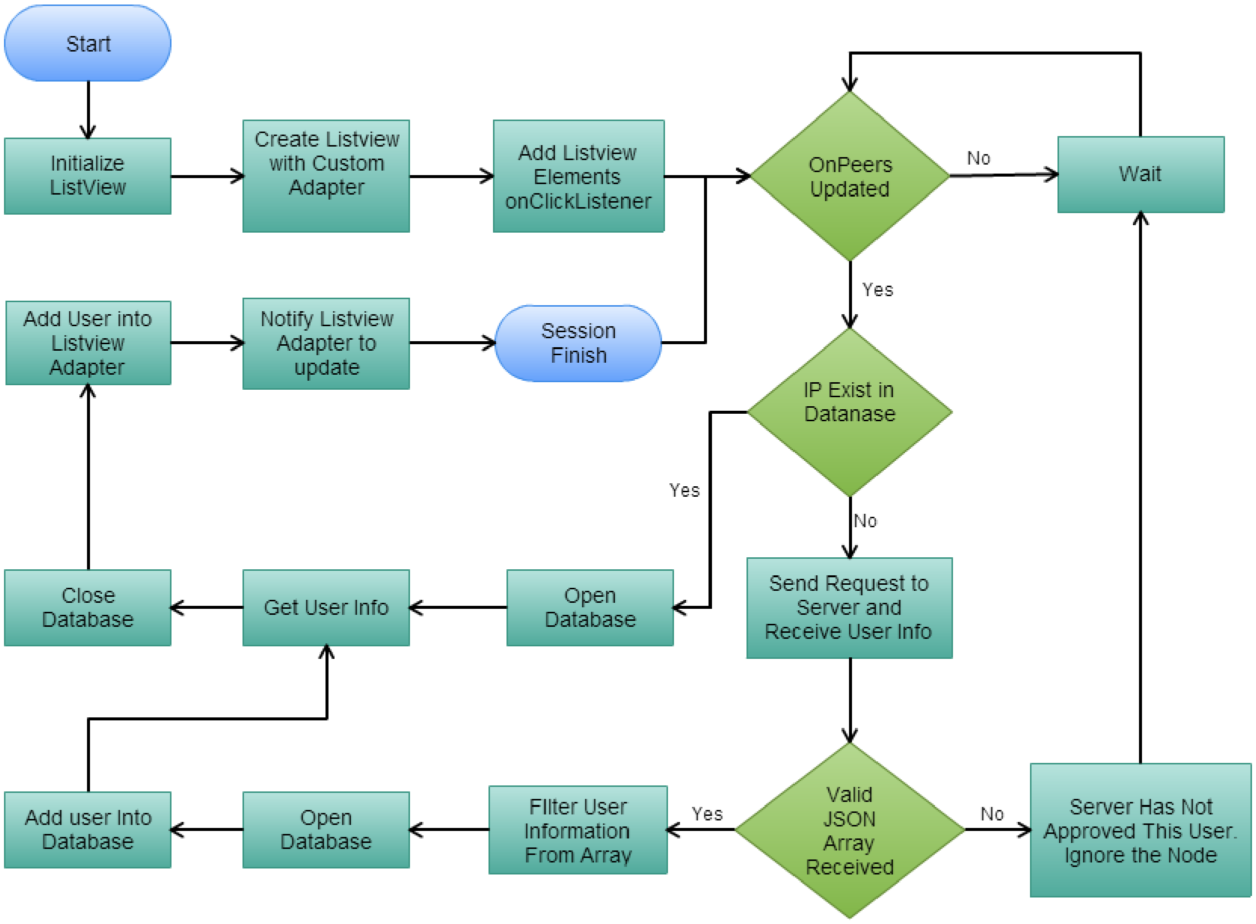
Figure 4.8: Activity diagram: Part 2 overview of the application setup process.

Once the application received *200OK*, it will send a new request to obtain a valid IP address which the server allocates. Again, the application checks if the reply is as expected or not. If it obtained an valid IP address the application stores the IP and disconnects and stops the OLSR service. This must be done since the service is currently bound to an *activity* class. Since we are currently in the setup and not in the *MainActivity* class, we need to disconnect and stop it in the setup, and reinitialize it in the *MainActivity* class. In the *MainActivity* the OLSR is started and the new IP address is used.

From this setup the application has gathered the user information, used the hybrid IP allocation to contact the server, registered to the server and obtained a valid IP address that it can use in the D2D network.

4.4.2 Listview functionality

Since the Listview is the main feature of the *MainActivity* class and its influence on the UI is of great importance we have illustrated the main functionallity.



Start
Initialize ListView
Create Listview with Custom Adapter
Add Listview Elements onClickListener
OnPeers Updated
No
Wait
Yes
Add User into Listview Adapter
Notify Listview Adapter to update
Session Finish
IP Exist in Datanase
Yes
No
Close Database
Get User Info
Open Database
Send Request to Server and Receive User Info
Add user Into Database
Open Database
FIlter User Information From Array
Yes
Valid JSON Array Received
No
Server Has Not Approved This User. Ignore the Node

Figure 4.9: Activity diagram: Listview and onPeersUpdated overview of the application.

From the Figure 4.9 we begin in *Start*. This indicated that the user has entered the *MainActivity* class and reached the point where the listview is initialized.

Since we want to customize and design our own listview in each element of the list, we need to create a custom adapter. The adapter is bounded to the listview, and the adapter can define how the element inside the listview should look like. From Figure 4.10 the illustration of the adapter is showed. From each element that the adapter creates it appends it into the listview to be populated. Once the listview and adapter has been created and initialized, we bind *onClickListener* to the listview.

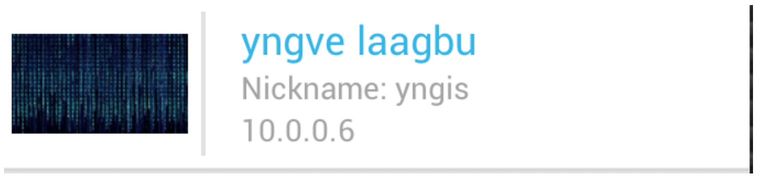


Figure 4.10: Illustration of one single element in listview using custom adapter.

The application now uses the listener *OnPeersUpdated* to check if any new nodes has entered the network. Once this occur, we check if the IP address of the node exist in our *Friend* table. If the user already exist in the Android database, it sends an query to the database and populate the user information into the adapter and notify it for changes so the listview can populate the user in the network.

If the IP address of the new node does not exist in the *Friend* table, the application sends a request to the server and asks for user information about this IP address. The server then replies back with a JSON array. The application then parses out the array and filters the user that it wants. Then the application opens the database and insert the user, where it will retrieve the user information and insert it into the adapter and notify the listview for changes to be populated.

In this topic we have explained and illustrated how the listview is bounded to a custom adapter, and how it works when a new node enters the D2D network.

4.4.3 HTTP media transmissions

In this section we will present how the application establish its socket ports for both receiving and sending data through the network. This section is divided into two parts, where the first part will evolve around when the application transmit data. In the second part we will show how the application create its own predefined socket ports for listening and how it reacts when data is sent to these ports.

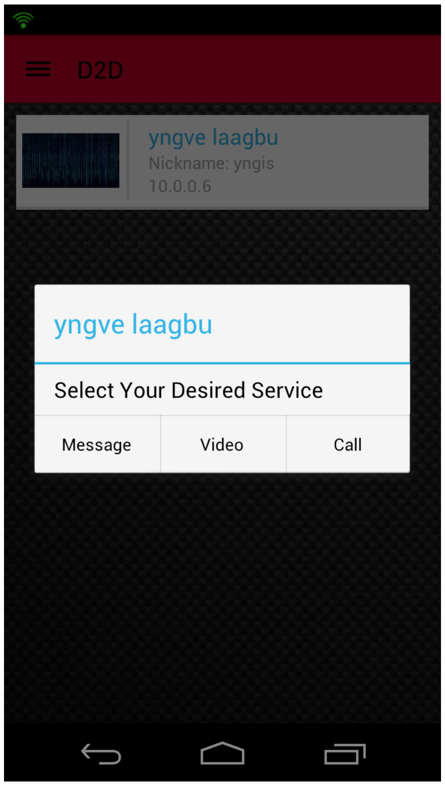


Figure 4.11: Dialog when user is selected from the listview.

When a user wants to call someone which is online in the network, this user will be showed in the listview of the application. When the desired user is selected, a dialog will be populated where the user has three options. From Figure 4.11 we can see that these are *Message*, *Video* and *Call*.

From Figure 4.12 we can see the general overview when the user selects a user from the listview. In this illustration a user has been selected and is prompted with the dialog. If the user selects *Message*, it will have the opportunities to add a picture. If an picture is added, the application checks if any text is added. If not, the application will generate a predefined message which the recipient will receive in its *notification-bar*. This functionality is also predefined if a user sends a video. For the *Call* feature, the application will use an separate thread when initializing a call. When initiating a call, the application checks if the phone is in *Idle* state. If this is true, then the application can initialize the call. Finally we can see from the Figure 4.12 that *Message*, *Video* and *Call* sends on its own port.

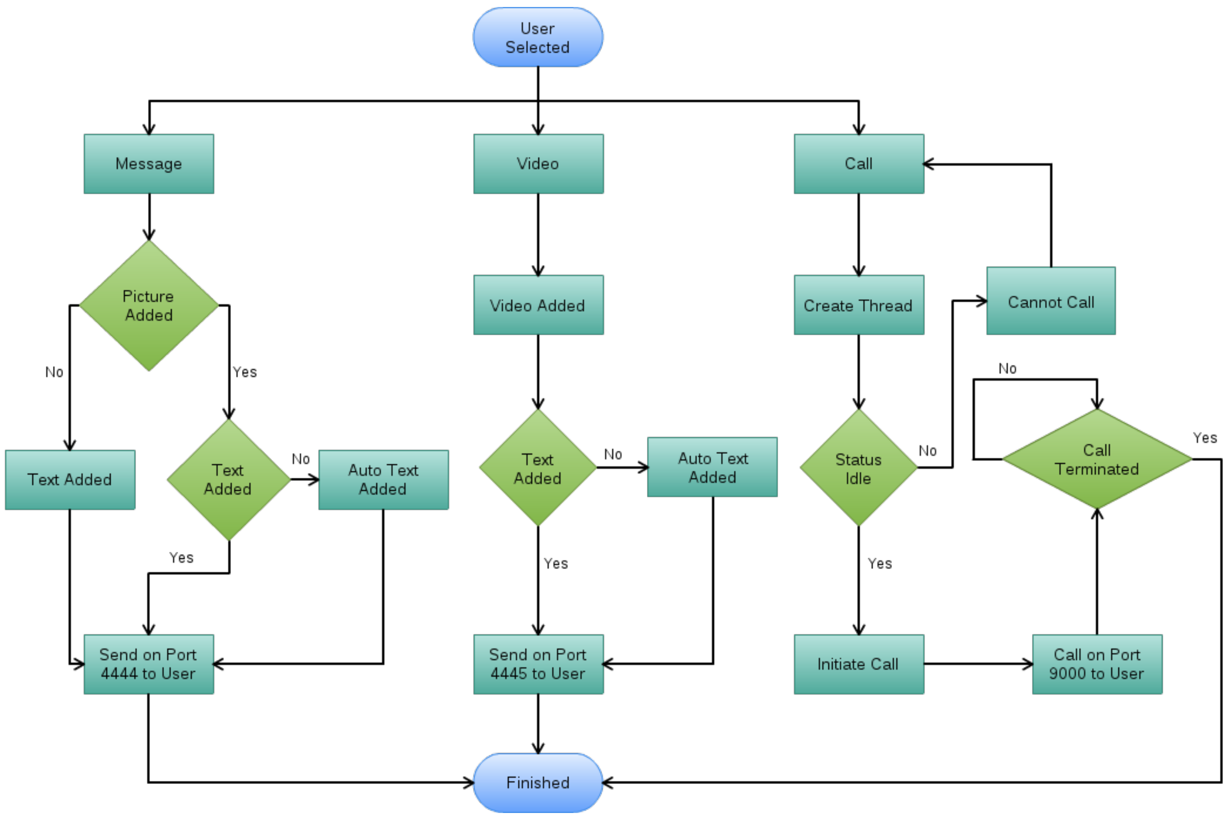


Figure 4.12: Activity diagram: Transmitter overview of message, video and call processes.

When the Android application is started and enters the *MainActivity* class, we start three separate socket port listeners. These listeners will trigger and alert the application if data is sent to any of these ports. From Figure 4.13 we can see an overview of how the application reacts if one of the ports gets incoming data. If data is received on port *4444* or *4445* the application will start to read the byte stream. Since this transmission consist of text message, picture and video, we have used Transmission Control Protocol (TCP) to ensure that the data is completely transmitted from sender to receiver. In both these two cases the application will populate an message in the *notification-bar* so the user is aware that it has received something from another user. Once this is clicked an UI is populate with the data that was transmitted.

If the user receives data on port *9000*, the application knows that someone wants to initiate a call session. The audio stream is sent using UDP transmission. Once port *9000* trigger, it will tell the application to start a call sound and populate a dialog telling the user that another user wish to initiate a call. The recipient can then decide to accept or decline the call.

In this scenario we have showed a brief overview of what happens when a user selects a desired medium to send to a recipient. The Figures 4.12 and 4.13 shows how the application reacts when data is sent and received on the different socket ports.

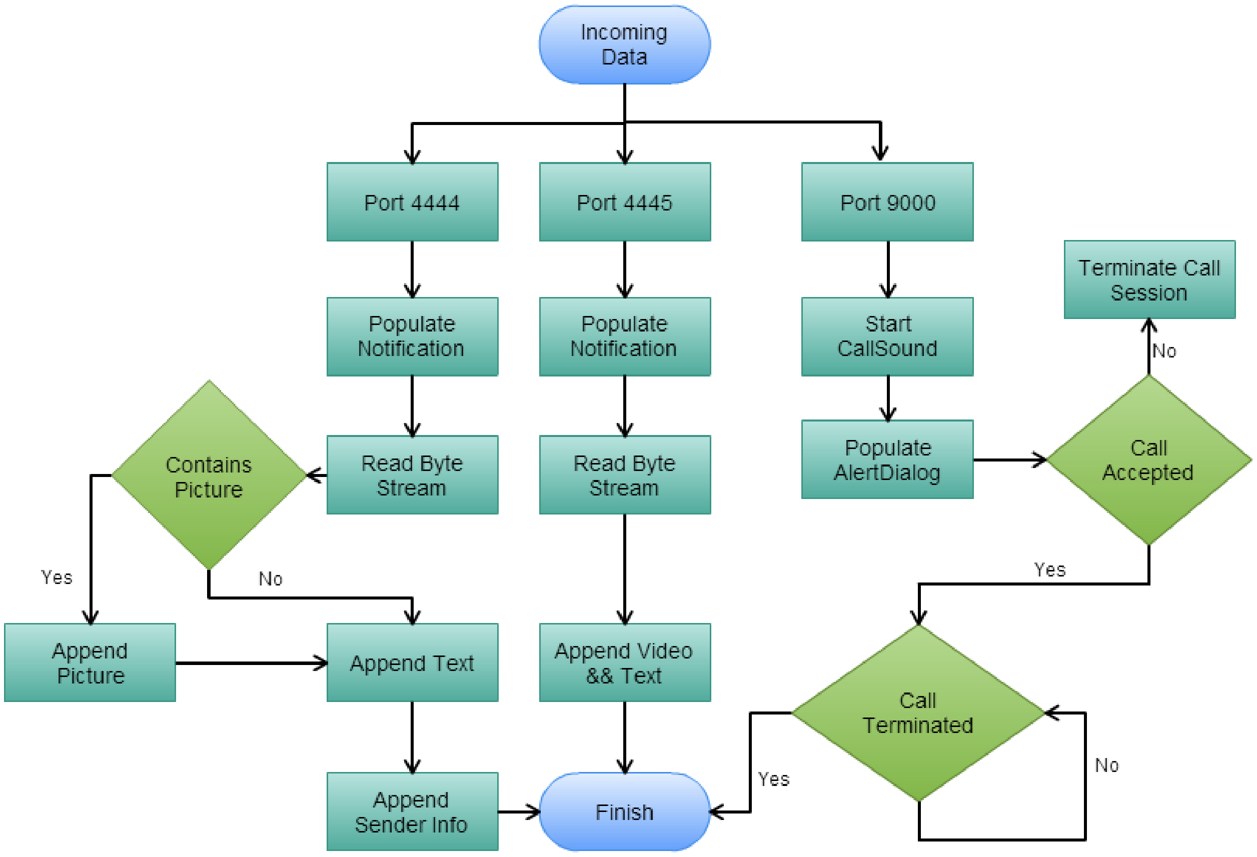


Figure 4.13: Activity diagram: Recipient overview of message, video and call processes.

4.5 Illustration of the Implemented Prototype

In this section we will illustrate the complete prototype system, including five smartphones, two tablets and two developed cases. Figure 4.14 illustrates the developed prototype cases, where we have mounted all the equipment that is needed to extend coverage of the system. Subfigure 4.14a show the case including the enabling Raspberry Pi, TP-Link router and the Intocircuit battery pack. Whilst Subfigure 4.14b show the other case with a TP-Link router and the power pack. The final subfigure show the closed case with mounted antennas. Further on in Figure 4.15 we have the complete system with all the devices in our system.

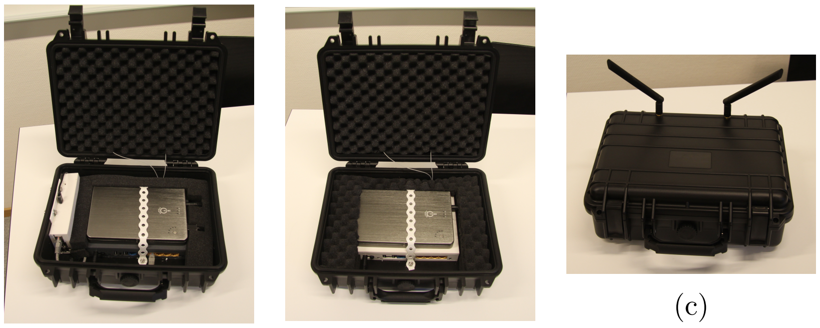

(a) (b) (c)

Figure 4.14: The two designed prototype cases, with all the equipment mounted.

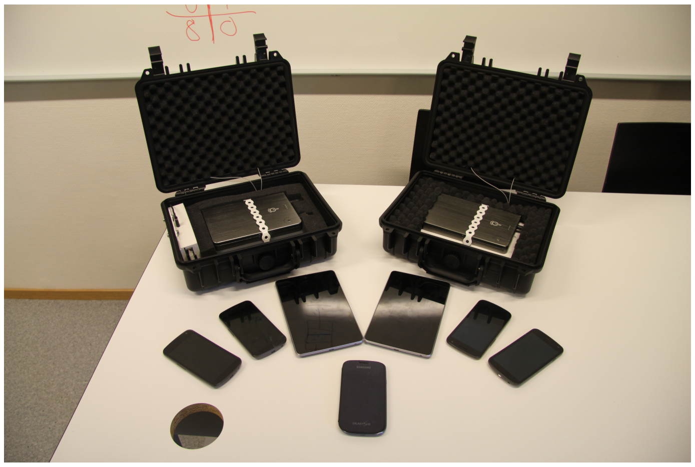


Figure 4.15: The prototype system with two tablets, five smartphones and the two cases.

4.6 Chapter Summary

In this chapter we have elaborated on our implementation efforts, and how our system works in practise. First we give a general introduction to the network topology where the three main entities in our system are introduced. Next we provide a introduction to the enabling OLSR routers, and how they are configured and integrated into our system. Section 4.3 give a detailed overview of the efforts done to set up and develop the Raspberry Pi based hybrid IP allocation server. Here we explained the software needed to make the Raspberry Pi act as we desire, our developed user framework, and IP allocation handling. All are described in detail with a strong correlation to the appendices, which include the source codes and step by step guides required to set up the system. Further on in Section 4.4 we describes the efforts done to develop and implement the Android application, as well as the fundamental functionalities. The last section illustrates the implemented prototype system with all the devices in our system and the developed prototype cases.

Chapter 5

Experiments and Results

This chapter will introduce the efforts done to validate our implementation as well as providing real-life experiments. The first section include the validation of the implementation. Further in Section 2 we will give a general overview of the experiments that we have performed, before we move on to the experiments itself in the following sections.

5.1 Validation of the Implementation

This section aims at demonstrating that our implementation and development efforts work as expected. First we validate our efforts done to integrate the OLSR routers into our system, before we verify that the database and enabling scripts on the Raspberry Pi is working. The third subsection aims at providing our efforts done to validate the developed Android application. When developing and validating each section of the system separately, we gradually merged the validation to test more of the overall system.

5.1.1 OLSR routers

In order to verify that the TP-Link routers work as expected, we first finalized the configuration of the routers, enabled OLSR and connected them to AC power. After this process we connected several smartphones to the system and ensured that the router announced the correct Host Network Address (HNA), and that all the enabled smartphones were able to discover the routers. Due to general lack of documentation on setup of OLSR on these routers, we manually had to test and verify with several configurations. Next, we ensured that the configuration were done correctly by checking all parameters regarding the ad hoc mode and OLSR service.

5.1.2 Raspberry Pi based hybrid IP allocation server

On the Raspberry Pi based hybrid IP allocation server we needed to do more extensive testing in order to validate that everything worked as expected. First we installed all the necessary packages, configured the database and put all the enabling scripts on the web server. To prove that all the scripts worked as desired we developed separate scripts that would prove that the scripts were able to post and retrieve all the necessary information to and from the database. These simulation scripts are available in Appendix C.3. In order to verify that everything worked as expected we divided the validation into the two following tasks.

Manual code validation

This is done by thoroughly reading through the written scripts to identify bad code and possible unused variables. By doing so we were able to find potential for improvement of the code, and were able to fix issues that could have broken the scripts. Finally we verified that the code was written in correspondence to what is considered as good coding.

Verification

- Manually read through the code and look for coding flaws.
- Use debugging logs of the Apache web server, to find errors.
- Simulate request to the server, by issuing test scripts.
- Monitor the database to ensure that the correct information is added and retrieved.

5.1.3 Android application

Validation of the Android application has been done in five main parts, which are:

- Validate code functionality by manually going through the code.
- Validate code functionality by using *Logcat* to identify grow heaps, exceptions, etc.
- Verify that OLSR service performed as expected and intended.
- Sniff traffic between client and server to verify successful communication.
- Application performance on different devices.

In order to avoid the most crucial bugs and glitches in the application, we have gone thoroughly through both manually and by using the *Logcat* tool to fix and optimize the application. By using these two methods we have managed to clean up and fix many bugs that the application initially had under development. For instance, we managed to reduce the *setup* time by 30 seconds when we manually looked through the code. We discovered that we could optimize some of the code from one core java class when the OLSR service was initialized and restarted.

Before validating that the application could communicate to the server, we started by using most of the default configurations of the existing OLSR service. During the progress, the OLSR service have been modified to our own desire to meet the system requirements.

Most of the validation took place by printing out logs and afterwards looking through, and verify that the OLSR service was initiated and performed as we wanted.

To validate that the application could talk to the server we used several tools to sniff and dump the traffic of the phones. By going through the traffic and looking at the request and responses made by both the phones and server we had the opportunity to validate and optimize how the client and server side communicated.

Lastly, we installed the application on the Nexus Tablet, Nexus I9250 and Galaxy S3 devices to check how the application performed on different devices. Since all three models is different in both screen size and performance it gave an good indication if the application had some issues and glitches. From this validation the application UI was rewritten several times to be proper scalable in the main views. We also discovered that the Nexus I9250 phones also at some times had problems with grow heap which resulted in small glitches and lagging on the UI. This made us go back to manually looking through the code and by using *Logcat* to fix these issues.

5.2 Test Scenarios

In order to provide viable results that should provide grounds for validation of the developed system, we have set up several test scenarios that will introduce a few different experiments. The experiments are performed at the university campus. Some of the results may be interfered by the available WiFi zones around the campus area. This is out of our control, and may impact the validity of the achieved results. The following five test scenarios are identified for experiments, and consequently numerical results are presented in the following:

- I Hybrid IP address Allocation: Experiments which ensure that the developed system and Raspberry Pi platform are able to provide hybrid IP allocation, as well as results that show that the initial setup process works as expected.
- II Service Provisioning: Experiments that provide insight into the quality of services, and the obtained delay in real-life.

III Stress Testing: Experiments done to stress the system, in order to provide validity of the developed Raspberry Pi database, to show that it can provide service to a larger number of connected nodes than we are able to perform experiments on.

IV Quantitative Performance Measurements: Experiments that show results on how long the developed prototype cases are able to operate in the network, and the Round Trip Time (RTT) that are achieved by utilizing multi-hop.

V Power Consumption of Android Smartphones: Experiments that are performed to validate how long the developed Android application is able to operate on a smartphone in the network.

5.3 Scenario I: Hybrid IP Address Allocation

In this scenario we will provide results that can prove that the system is working with a hybrid IP address allocation. The results will provide code snippets as well as debugging logs from the application. To verify that the application sends the correct information to the server, we will log all the network packages to sniff out the traffic when the client transmits its data.

5.3.1 Android self-generated IP

When the application is going through the setup for the first time it will generate a random IP in the range of *10.0.X.254*, where X is the variable. The randomizing is accomplished by using the *java.lang.math* class. Documentation for this class can be found here [30]. From Listing 5.1 we can see how this is accomplished in the Android application. The *int randomNum* first generate a random value between 1 and 255. Then we append this into the *String RandomizedIP* to produce a randomized IP. Even though the randomized IP is only used in a limited amount of time it is not an optimal solution since we have $\frac{1}{255}$ chance of IP collisions while registering. Still, we have not had one collision yet during our test so we consider this method as satisfactory in our prototype system.

Listing 5.1: Randomizing IP

```java
int randomNum        = (int) Math.ceil(Math.random() * 254) + 1;
String RandomizedIP = "10.0." + String.valueOf(randomNum) + ".254";
```

5.3.2 Debugging log

From Listing B.1 we have shown how the application log the setup phase. By looking at the output of the log we first get the user information and store it in the *sharedPreference* location [13]. It checks the user logo and compresses it if the image size is > 100KB. This is mainly because sending a large image file as a logo which in our case is supposed to be scaled as an *50x50px* is pointless and will just cause more resources to both download and show within the application.

The application then creates two database tables. The first one is called *Friends* which its neighbour nodes will be placed in later on, when the OLSR discovers new nodes in the network. The second table is created for the user of the application. After this is finished, the OLSR service will be initialized. We can also see that the application has randomized an IP address which in this case is *10.0.20.254* and will register the user *karl hansen* to the server. It then send new request to the server where it should get a valid IP that can be used within the system. In this case the received IP is *10.0.0.9*. Since the OLSR service is already initialized and started, we need to restart the OLSR and tell the service that it should now use *10.0.0.9*.

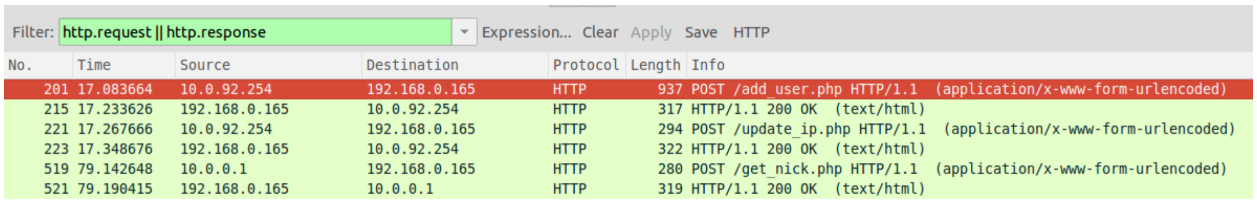
During this sequence there is several *catch exceptions* that has been implemented to try and handle all the errors that could occur. For instance, we discovered that in some rare cases the application gave an exception on sending the HTTP request. This resulted in not obtaining any valid IP and just continued using the randomized IP in the network. After this was discovered we implemented *catch exception* on the HTTP request. If the application enters this state we use a boolean and flag it as *false*. Right before the application goes into the MainActivity class we check if the boolean has been flagged *false*. If this is true then we force the application to not enter the *MainActivity*.

In this case the application managed to send HTTP requests to the server where the users was both registered and it got a new valid IP. The application then enters the MainActivity class. Here we check if the OLSR service has managed to use the new valid IP. If this is not the case then we restart the OLSR once again and use the valid IP. When the application enters the MainActivity class there is several important elements that start. We can for instance see that the application creates three socket listening ports (4444, 4445 and 9000). These ports are used for sending SMS, MMS, and video between the users.

Finally, the MainActivity class has its own function where it triggers when a node enter the network. When this occur, it check the IP of the node against the *Friend* table in the database. If it gets a match from the table, then we populate the information about that user. If the user does not exist in the table, then the application send a request to the server and asks for the user information. In this case we can see that a new IP address *10.0.0.3* has been discovered. Since it does not exist in the table, it then sends the request to the server, and gets information of this user. The information is then inserted into the *Friend* table and populated in the application using our custom *Listview* adapter. This function will continue to trigger and check every node that enters the network until the application is paused or closed.

5.3.3 Smartphone to Raspberry Pi communication

On each smartphone it is necessary to do an initial registration process that involves two vital steps. First the phone is registered in the local database provided in the Raspberry Pi, before the allocation of an IP address is performed. In order to test that this process is executed as desired we set up an experiment where we utilized a listener that logged all activity on the network card on one of the Android smartphones. The recorded log were then analysed by Wireshark to provide visible proof of concept. Figure 5.1 illustrate the initial process that are executed on the smartphone, where we first see that the client issues a request to the Raspberry Pi including the user credentials, before it responds with a *200 OK* message indicating that the request is approved. Furthermore we see that the smartphone sends a request to retrieve a valid IP address, and that the Raspberry Pi responds to the request. Finally we see a third request and response that indicate that the smartphone have noticed another IP address in the network, and that it requires information on who is on the respective address. In order to provide even more information of the requests and responses we have included a short listing of each of the requests and responses. First we have Listing 5.2 which illustrate the message the Smartphone sends to the Raspberry Pi, including *Michael Stensrud*, the image and the nickname *michas*, along with the 200 OK response on line 17. Further in Listing 5.3 the update of IP address is illustrated, here the smartphone sends it current IP address *10.0.92.254* which is a randomed IP address, the server then responds with the first available address that are *10.0.0.1* in this case. The last Listing 5.4 illustrate the process to retrieve information of available IP address. In this case the IP is an TP-Link router, so the Raspberry Pi issues a *false* statement as expected, since we don't want to populate the routers in the *Listview*.


Filter: http.request || http.response
Expression... Clear Apply Save HTTP

```
No.  Time       Source         Destination    Protocol Length Info
 201 17.083664  10.0.92.254    192.168.0.165  HTTP      937 POST /add_user.php HTTP/1.1  (application/x-www-form-urlencoded)
 215 17.233626  192.168.0.165  10.0.92.254    HTTP      317 HTTP/1.1 200 OK  (text/html)
 221 17.267666  10.0.92.254    192.168.0.165  HTTP      294 POST /update_ip.php HTTP/1.1  (application/x-www-form-urlencoded)
 223 17.348676  192.168.0.165  10.0.92.254    HTTP      322 HTTP/1.1 200 OK  (text/html)
 519 79.142648  10.0.0.1       192.168.0.165  HTTP      280 POST /get_nick.php HTTP/1.1  (application/x-www-form-urlencoded)
 521 79.190415  192.168.0.165  10.0.0.1       HTTP      319 HTTP/1.1 200 OK  (text/html)
```

Figure 5.1: Wireshark representation of the initial setup sequence.

Listing 5.2: POST request for registry of new user

```
   POST /add_user.php HTTP/1.1
2  Content-Length: 28381
   Content-Type: application/x-www-form-urlencoded
4  Host: 192.168.0.165
   Connection: Keep-Alive
6  User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

8  NAME=michael&LASTNAME=stensrud&IMAGE=BYTE_FORMATTED&NICK=michasHTTP/1.1 200 OK
   Date: Wed, 29 Apr 2015 14:19:07 GMT
10 Server: Apache/2.2.22 (Debian)
   X-Powered-By: PHP/5.4.39-0+deb7u2
12 Vary: Accept-Encoding
   Content-Length: 3
14 Keep-Alive: timeout=5, max=100
   Connection: Keep-Alive
16 Content-Type: text/html
   200
```

Listing 5.3: POST request for new IP address

```
POST /update_ip.php HTTP/1.1
Content-Length: 26
Content-Type: application/x-www-form-urlencoded
Host: 192.168.0.165
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

NICK=michas&IP=10.0.92.254HTTP/1.1 200 OK
Date: Wed, 29 Apr 2015 14:19:07 GMT
Server: Apache/2.2.22 (Debian)
X-Powered-By: PHP/5.4.39-0+deb7u2
Vary: Accept-Encoding
Content-Length: 8
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
10.0.0.1
```

Listing 5.4: POST request for request of user data

```
POST /get_nick.php HTTP/1.1
Content-Length: 13
Content-Type: application/x-www-form-urlencoded
Host: 192.168.0.165
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

IP=10.0.255.1HTTP/1.1 200 OK
Date: Wed, 29 Apr 2015 14:20:09 GMT
Server: Apache/2.2.22 (Debian)
X-Powered-By: PHP/5.4.39-0+deb7u2
Vary: Accept-Encoding
Content-Length: 5
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
false
```

5.4 Scenario II: Service Provisioning

In order to validate the implemented services from the Android application, we have done experiments on one,two and three hops communication. The results provide a reflection of how the different services function when deployed indoor. The test location has been at the UiA, where the walls mainly consists of concrete and drywall, which may give variance on the tests results. In addition the WiFi zones at the UiA campus may have interfered the channel that the OLSR service is using.

For all experiments we have modified the Android application to automatically reply back with an message when the recipient has received its message from the transmitter. The modifications can been be viewed in Listing B.4. The device that has taken part in the multi-hop communication has been placed in the same position for all experiments. For each test we have used an third-party application called Shark [18]. This application dumps the traffic from the source device and stores to an *PCAC* file. After each experiment the *PCAP* files are opened using Wireshark where we can find the timestamp when packets has been transmitted and received.

	Experiment			
Delay	1	2	3	Average
SMS	10.8 ms	22.3 ms	23.1 ms	18.7 ms
MMS*	403.7 ms	381.8 ms	397.6 ms	394.4 ms
Video**	1.56 s	1.55 s	1.57 s	1.56 s

* Picture size = 717 kB
** Video size = 3.3 MB

Table 5.1: Obtained delays when running the application on two phones over one hop.

From the results provided in Tables 5.1, 5.2 and 5.3 we can clearly see that the delay vary quite much between each experiment. The results when testing one-hop communication is generally quite good. By looking at the traffic we saw that there were not much retransmission of the packets. For the experiments of two and three hops we saw that the retransmissions increased, which also reflects the delay in these results. From the results in Tables 5.1, 5.2 and 5.3 graphical illustrations of these results have been provided in Figure 5.2.

	Experiment			
Delay	1	2	3	Average
SMS	35.6 ms	37.0 ms	31.9 ms	34.8 ms
MMS*	3.18 s	2.29 s	1.86 s	2.44 s
Video**	11.7 s	15.51 s	6.3 s	11.18 s

* Picture size = 717 kB
** Video size = 3.3 MB

Table 5.2: Obtained delays when running the application on two phones over two hops.

	Experiment			
Delay	1	2	3	Average
SMS	46.4 ms	44.5 ms	58.3 ms	49.7 ms
MMS*	8.64 s	8.06 s	8.93 s	8.55 s
Video**	42.9 s	42.7 s	51.9 s	45.85 s

* Picture size = 717 kB
** Video size = 3.3 MB

Table 5.3: Obtained delays when running the application on two phones over three hops.

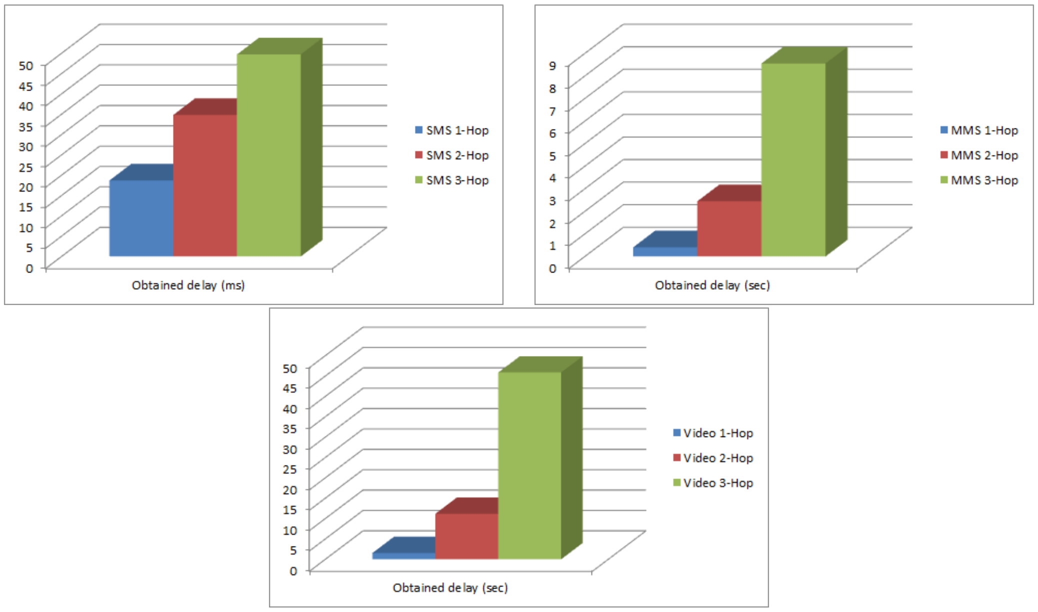


Figure 5.2: SMS, MMS and video delay over one, two and three hops.

5.5 Scenario III: Stress testing

In this scenario we will perform a numerous set of test to validate that the system is able to cope with multiple clients and provide stable service regardless of the clients are utilizing multiple hops. The first experiment will focus on the stability and scalability of the database on the Raspberry Pi. Secondly we will perform tests to ensure that the network are able to cope with a various number of nodes, and that the performance is not affected. Finally we will test the network throughput, this to check how well the network are able to cope with ad hoc routing over multiple hops.

5.5.1 Raspberry Pi database

The Raspberry Pi is a vital part of the entire system, and we want to ensure that the system can be scalable, and will handle much traffic. Below follows two subsections that elaborates and demonstrates that the system is scalable and handle a fair amount of traffic.

Synchronized stability test

To ensure that the Raspberry Pi database would be able to endure stability with a lot of connecting nodes, we set up at synchronized stability test that register 10 000 new user in the database. The test script are previously explained in Section 5.5, and allow us to run a continuous flow of new registry events on the Raspberry Pi from an Android smartphone using the OLSR network. The result show that the system are quite able to handle the flow of new request. From Figure 5.3 we see that the average latency of the system is about 53 ms. This is well within the expected values for the system, and we can conclude that the system should operate quite well.

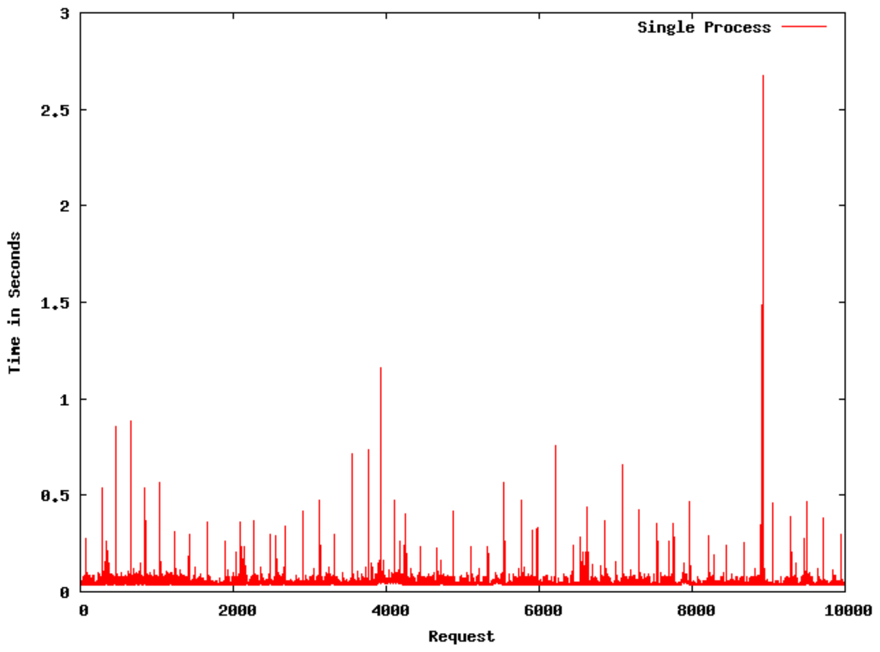


Figure 5.3: Single process executing 10.000 synchronized registration requests.

Furthermore we wanted to check how well the system would handle system calls to retrieve data from the database. The previous script generated a large portion of users in the database, where we wanted to measure how fast the database responded if it could provide stable flow of user data to external clients. We used the script explained in the earlier Section 5.5, and configured it to send 10 000 request to the Raspberry Pi database. The results show that the performance is well within what is to be expected, and the average response time is about 22 ms. This means that the system is swift and are able to cope with continuous traffic to and from the database. Figure 5.4 show the obtained execution delays. Both these tests are performed back to back, meaning that the calls are synchronized and no calls is performed before the previous call receives a response.

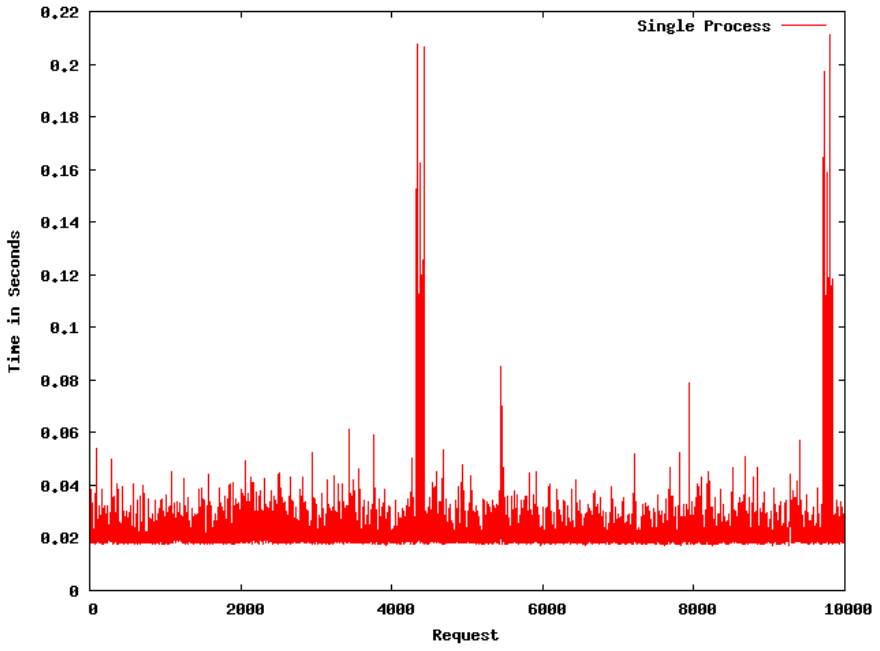


Figure 5.4: Single process executing 10.000 synchronized requests obtaining valid ip.

Stress testing the database

In order to test the Raspberry Pi's performance and database management we needed a more thorough test. Therefore we used the same scripts as in the last example. But this time we spawned 10 concurrent instances of the scripts which should stress the system and database. The test simulate 10 synchronous connections to the database in addition to 1 500 requests per instance, resulting in the system to slow down and possibly halt if there is not enough memory. The first test we initialized was the add user test, where we populated the database with 10 000 users from 10 simultaneous scripts from an external Android smartphone connected to the backbone OLSR network. The result show that the system struggled with the task, but it did not fail. Figure 5.7 show that the system is running at near full performance and that there is almost not enough memory left. Further on the test revealed that the response time is averaged around 272 ms, which is quite well since the system is put under a lot of stress. Figure 5.5 show the ten simulations and the achieved delay times, where we get our averaged delay time.

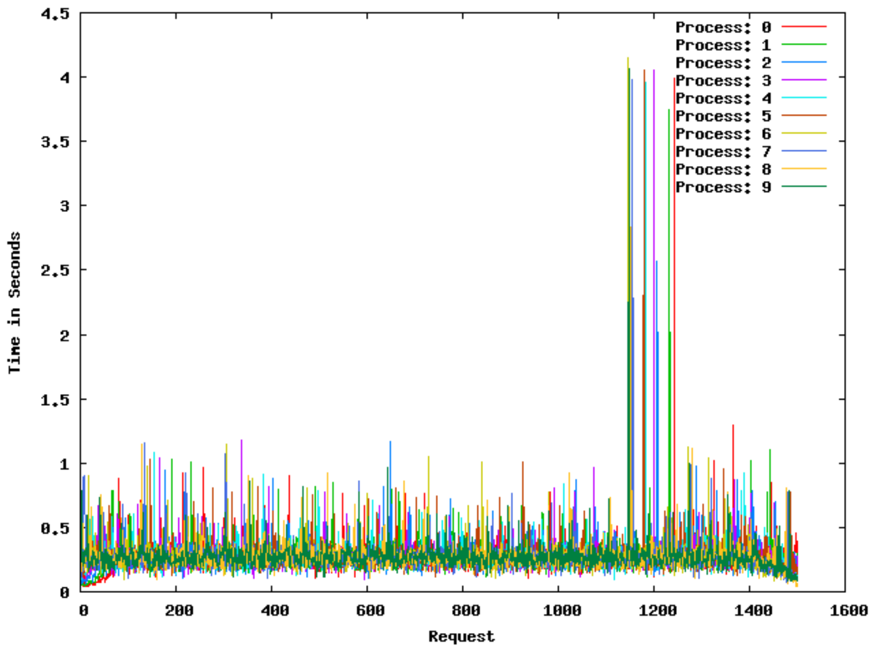


Figure 5.5: 10 Processes executing 1500 synchronized registration requests.

Furthermore we tested the response time when we executed 10 simultaneous calls to retrieve 1 500 individual user data from the database. Here we also used the same script as before and initiated them to run ten simultaneous simulations. The results show that the system struggles a bit, but not quite as much as for the previous case. Figure 5.6 shows us that we have a averaged execution time of about 173 ms.

To illustrate that the client and embedded system are issuing request and responses in parallel to each other, we have included a snippet from both platforms. From Figure 5.7 we can see that there are several instances of request bound by different ports. On the Raspberry Pi we can see that the system is put under some stress, and are running at 54.1% Central Processing Unit (CPU) power, and that there are several instances of the Apache web server, indicating that there are many request to the enabling PHP scripts. On the client we observed that the device spawned 10 instances of request to the embedded system, and that they are bound to different ports.

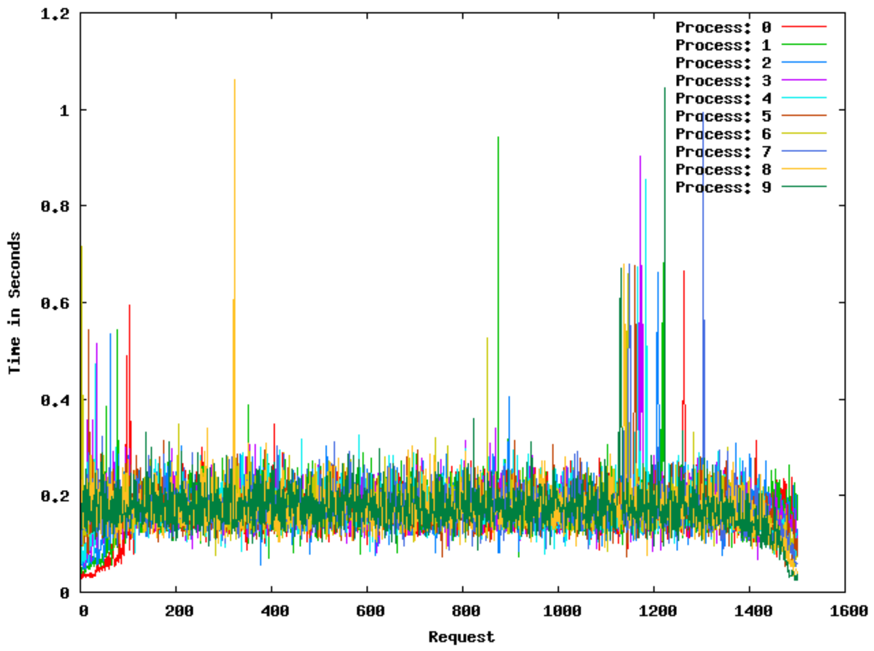
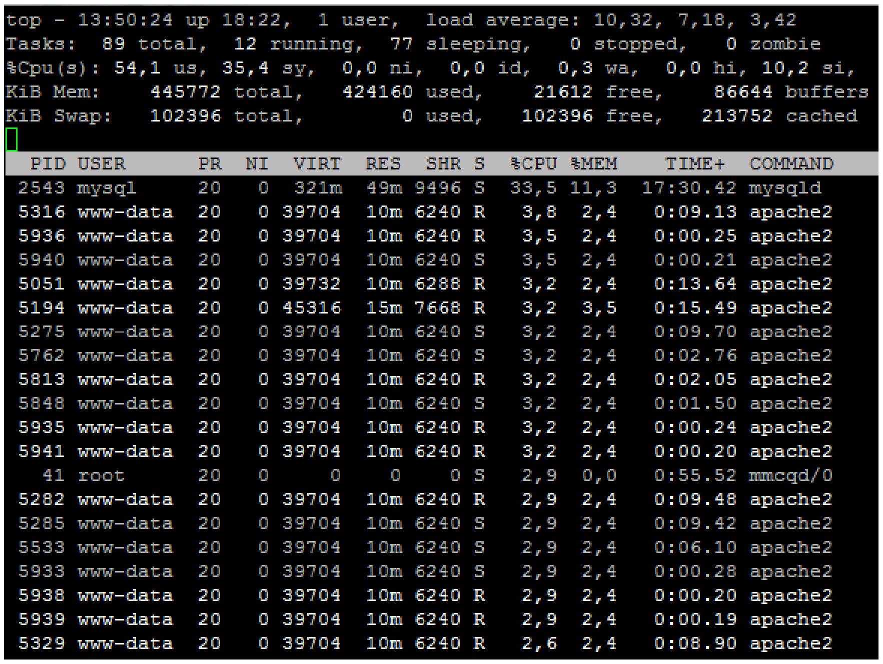


Figure 5.6: 10 Processes executing 1500 synchronized requests obtaining valid ip.



```
top - 13:50:24 up 18:22,  1 user,  load average: 10,32, 7,18, 3,42
Tasks:  89 total,  12 running,  77 sleeping,   0 stopped,   0 zombie
%Cpu(s): 54,1 us, 35,4 sy,  0,0 ni,  0,0 id,  0,3 wa,  0,0 hi, 10,2 si,
KiB Mem:    445772 total,   424160 used,    21612 free,    86644 buffers
KiB Swap:   102396 total,        0 used,   102396 free,   213752 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 2543 mysql     20   0  321m  49m 9496 S  33,5 11,3  17:30.42 mysqld
 5316 www-data  20   0 39704  10m 6240 R   3,8  2,4   0:09.13 apache2
 5936 www-data  20   0 39704  10m 6240 R   3,5  2,4   0:00.25 apache2
 5940 www-data  20   0 39704  10m 6240 S   3,5  2,4   0:00.21 apache2
 5051 www-data  20   0 39732  10m 6288 R   3,2  2,4   0:13.64 apache2
 5194 www-data  20   0 45316  15m 7668 R   3,2  3,5   0:15.49 apache2
 5275 www-data  20   0 39704  10m 6240 S   3,2  2,4   0:09.70 apache2
 5762 www-data  20   0 39704  10m 6240 S   3,2  2,4   0:02.76 apache2
 5813 www-data  20   0 39704  10m 6240 R   3,2  2,4   0:02.05 apache2
 5848 www-data  20   0 39704  10m 6240 S   3,2  2,4   0:01.50 apache2
 5935 www-data  20   0 39704  10m 6240 R   3,2  2,4   0:00.24 apache2
 5941 www-data  20   0 39704  10m 6240 R   3,2  2,4   0:00.20 apache2
   41 root      20   0     0    0    0 S   2,9  0,0   0:55.52 mmcqd/0
 5282 www-data  20   0 39704  10m 6240 R   2,9  2,4   0:09.48 apache2
 5285 www-data  20   0 39704  10m 6240 S   2,9  2,4   0:09.42 apache2
 5533 www-data  20   0 39704  10m 6240 S   2,9  2,4   0:06.10 apache2
 5933 www-data  20   0 39704  10m 6240 S   2,9  2,4   0:00.28 apache2
 5938 www-data  20   0 39704  10m 6240 R   2,9  2,4   0:00.20 apache2
 5939 www-data  20   0 39704  10m 6240 R   2,9  2,4   0:00.19 apache2
 5329 www-data  20   0 39704  10m 6240 R   2,6  2,4   0:08.90 apache2
```

Figure 5.7: Server creating unique threads for each incoming request.

5.5.2 Network HTTP throughput

In order to validate the system network throughput we connected the system to an Internet Service Provider (ISP), configured HNA announcement on the router and connected a smartphone to the network with using an application provided by Ookla which measures the throughput [29]. The results illustrate the system throughput using TCP traffic between the nodes. Table 5.4 illustrates the achieved rates over one and two hops in an indoor environment through concrete and drywalls. As we can see from the table we are able to provide about *3.11 Mbps* download and about *4.84 Mbps* in upload speeds over one hop. These results are well within what is to be expected, but when we look at the achieved results over two hops, the results departs. The network throughput is so low compared to the results over one hop, download speed about *1.5 Mbps* is not what we expected. As well as the upload test failed on two of the tests due to the lack of connectivity. We believe that these results have been influenced partially by the connectivity through walls and much noise on campus due to WiFi zones.

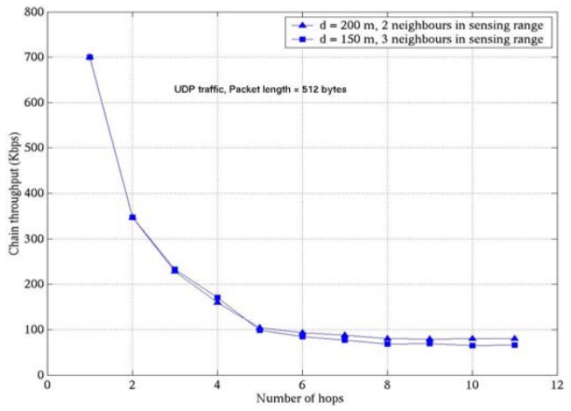
Experiment	Throughput			
	1-Hop (Mbps)		2-Hop (Mbps)	
	Down	Up	Down	Up
One	3.35	5.85	2.02	N.A
Two	3.73	4.23	0.98	N.A
Three	2.24	4.45	1.73	0.02
Average	3.11	4.84	1.58	N.A

Table 5.4: Obtained throughput on one of the Android smartphones over both one and two hops; Indoor through wall.

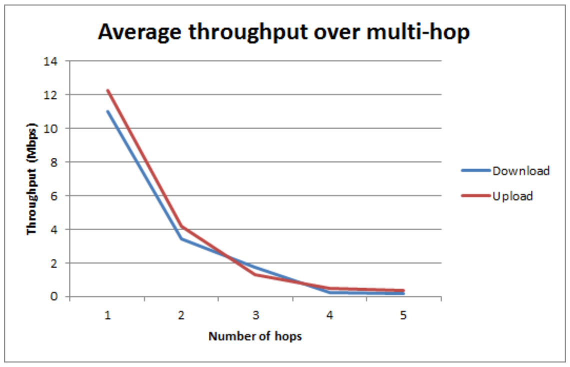
Due to the inconsistent results from the experiments though walls, we decided to perform experiments where every node had *in-line* sight to each other. The experiment was performed in the same indoor environment, but the nodes were placed differently. From Table 5.5 we can see that the results has improved drastically, compared to the results from Table 5.4. We obtained more viable results, with download and upload rates of about 11 Mbps over one hop. The results for two, three, four and five hops indicate that utilizing multi-hop communication reduces the throughput, which was expected.

Experiment	Throughput									
	1-Hop (Mbps)		2-Hops (Mbps)		3-Hops (Mbps)		4-Hop (Mbps)		5-Hop (Mbps)	
	Down	Up	Down	Up	Down	Up	Down	Up	Down	Up
One	14.13	12.73	4.80	4.62	1.97	1.34	0.13	0.33	0.19	0.48
Two	10.97	12.62	2.84	4.59	2.20	1.18	0.16	0.29	0.18	0.33
Three	7.81	11.43	2.58	3.24	1.13	1.38	0.4	0.82	0.16	0.28
Average	10.97	12.26	3.40	4.15	1.77	1.30	0.23	0.48	0.18	0.36

Table 5.5: Obtained throughput on one of the Android smartphones over one to five hops; Line of sight.



(a) Simulated chain throughput over multi-hop using UDP traffic [19].



(b) Obtained average chain throughput over one to five hops with TCP traffic.

Figure 5.8: Comparison of simulated and obtained results of typically ad hoc network.

From Figure 5.8a we can see an simulated example of throughput in an typical ad hoc network [19]. In this test the nodes have been place 150 and 200 meters apart with two and three nodes as neighbours. The simulation has been used by sending UDP traffic with an packet length of 512 bytes. Since our experiment is based on HTTP throughput using TCP traffic, our results will and should provide lower results then the simulated experiment. This is simply because the TCP traffic require *acknowledgement* between each packet that is sent. If a packet is dropped or corrupt, the packet need to be retransmitted, which will increase the latency and decrease the overall throughput. From Figure 5.8b we have merged all the results from Table 5.5 and populated it into a graph to see if we have some similarities to the simulated experiment.

Even though the simulated and real-life experiment uses different transmission protocols, we can clearly see comparisons in how the throughput decreases for each hop. The graphs have the same tendency of decreasing, which indicates and verify both the simulated test, as well as our real-life experiment.

5.6 Scenario IV: Quantitative Performance Measurements

In this scenario we will perform experiments to provide results that illustrate how long the developed prototype cases are able to operate. In addition we have results indicating the impact of multi-hop service based on RTT in the network.

5.6.1 Delay and packet loss

Since the system should operate over multiple hops, we wanted to provide results that show the latency of the system. We therefore set up a test scenario where we validate how the system performs in contexts to delay between nodes over one, two and three hops. The experiment was performed indoor and between both concrete and drywalls.

To provide these results we deployed 100 Internet Control Message Protocol (ICMP) requests from one phone to another. Figure 5.10 illustrates the received data, and provides an general overview of the delay in the network. From Table 5.6 we have illustrated the packet loss and average delay for each experiment.

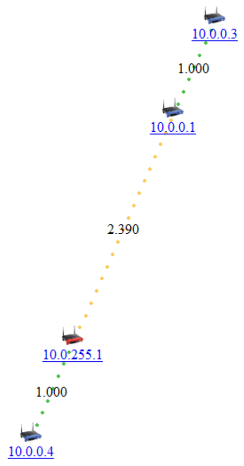


Figure 5.9: Representation of the test enviorment with 3 connected enitites and the connection quality between them.

From the illustrations we can see that one, and two hop have more packet loss than three hops. One explanation could be noise from the WiFi zones which may have interfered with our test. From Figure 5.9

we have illustration of the current network topology when we deployed our test. We can observe from this illustration that the cost between *10.0.255.1* and *10.0.0.3* is the weakest link in this network topology. Since the test between two and three hop test was performed through *10.0.255.1*, this is most likely an cause to why our results varied. This may also be an important reason to why we have an general delay which is four times a much compared to the one hop test.

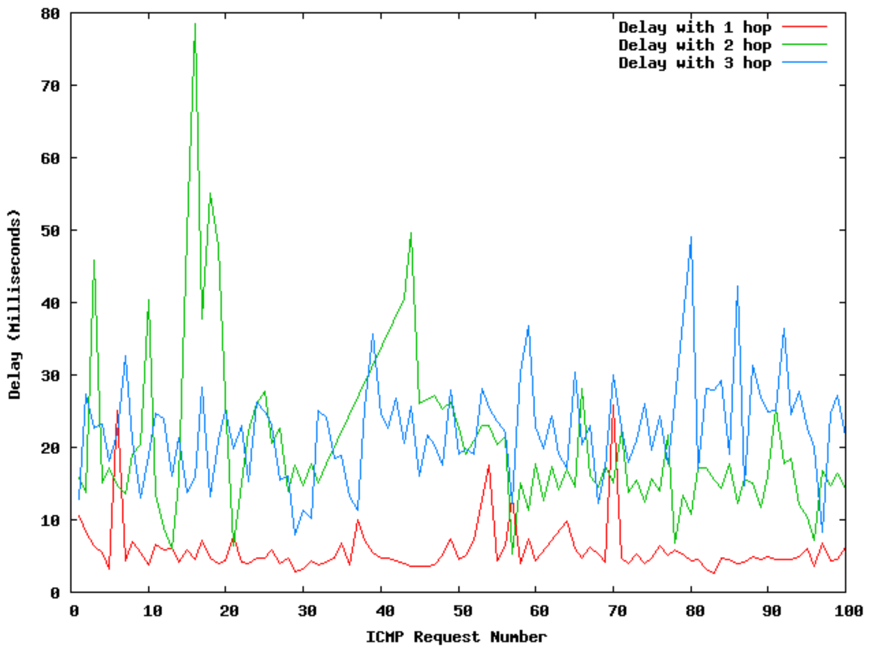


Figure 5.10: ICMP delay over one, two and three hops.

Hop Count	Packet Loss	Average Delay
One	5%	5.755 ms
Two	11%	20.122 ms
Three	1%	22.240 ms

Table 5.6: Packet loss and delay with one, two and three hops

5.6.2 Network lifetime on battery

The prototype cases include battery power that should be able to power the devices for up to 15 hours with the battery pack of 26 Ah. But this is only theoretically obtained, and we therefore needed to do some experiments to validate how long they were able to operate. This is done by fully charging the battery pack, and connecting one TP-Link router along with the enabling Raspberry Pi in normal operational state. On the connected Raspberry Pi we initiated the scripts previously explained in Subsection 4.3.4. The script logs how long the Raspberry Pi are up and running. We ran the simulation a total of three times, which provided us with the results presented in Table 5.7. From the table we can see that we actually obtained an average uptime of 17 hours and 1 minute and an average power drain of about 1.52 A. These results prove that our system is capable of operating longer that we anticipated with regards to the estimated uptime of about 15 hours. Still, the system in its whole won't be able to operate as long as this due to the energy consumptions on the mobile smartphones, which is further explained in Section 5.7.

Experiment	Uptime		Power drain
	Hours	Minutes	(avg)
One	17	12	1.51 A
Two	16	49	1.54 A
Three	17	4	1.52 A
Average	17	1	1.52 A

Table 5.7: Obtained uptime of the prototype system running on battery power.

5.7 Scenario V: Power Consumption of Android Smartphones

This scenario will provide results with regards to the Android smartphones and how long they are able to operate in our system. First we will perform tests to validate how much power the implemented application requires in contrast to a plain initialized smartphone.

Next we will perform several experiments to provide results on how long the system is able to provide selected services. The final subsection will include some comparison of the achieved results. The results presented in this section are related to one specific smartphone namely the Samsung I9250, and does not imply that the same results would be achieved by a different phone. It is simply an approximation of what is to be expected by our developed Android application.

5.7.1 Without data traffic

In order to validate how long the smartphone is able to operate without any data traffic, and to compare the energy consumption between non-operational and operational state. First we initialized a smartphone without any services running except the uptime timer previously explained in Subsection 4.3.4, fully charged it and left it to deplete its battery. After 24 hours we checked if the phone still was active, and found out that it still had 96 % battery remaining. We therefore aborted the test, and concluded that a plain Android smartphone does not require much battery to stay in idle mode, without data traffic. To clarify, we did not have any services like 3G or WiFi enabled, which would have an impact on the low power draw. With that covered, we recharged the phone, initialized our application with ad hoc mode, started the timer and left it to deplete its battery. The test resulted in a uptime of 22 hours, indicating that an ad hoc enabled phone utilizing the OLSR protocol require much power just to keep its routing tables up to date. The phone used in the experiment are equipped with a 1750 mAh battery, and since the battery is reduced to approximately 0%, we can therefore estimate how much power the application are requiring. The phone in idle mode used 4% of the total battery power, resulting in a average drain of 2.9 mA, whilst the OLSR enabled experiment averaged at a power draw of 116.7 mA resulting in the application requiring about 75.5 mA to operate in idle mode. An overview of the achieved results is illustrated in Table 5.8. Where we also display that the system is capable of operate for 7 hours and 11 minutes with the screen on. This test was performed due to the fact that we would like to point out that much of the power on the device are utilized by keeping the screen on. The result show that the screen use about 165,3 mA.

Experiment	Uptime		Power drain
	Hours	Minutes	(avg)
Without OLSR, Screen off*	24	5	2.9 mA
With OLSR, Screen off	22	20	78.4 mA
With OLSR, Screen on	7	11	243.7 mA
* 96% battery remaining after experiment			

Table 5.8: Obtained uptime of one of the Android smartphones running in idle mode with and without OLSR enabled.

5.7.2 Continuous voice transmission

In contrast to the previous experiments, we set up a continuous voice transmission between two enabled smartphones. One of the smartphones were connected to AC power, while the other one was on battery power. To ensure that the system could provide communication in an extended period, and to validate that the voice transmission does not require so much power that the system fails after just a few hours of operation. The results show that the battery powered phone were able to provide service for an average of *10* hours and *1* minute. This is not bad results as continuous voice conversation over regular 3G networks [14] are about the same as we achieved from our application. The results also show us that a voice conversation drain about 96.2 mA, which is more compared to the ad hoc mode operation of 75.5 mA. All experiments for this scenario is listed in Table 5.9.

Experiment	Uptime		Power drain
	Hours	Minutes	(avg)
One	9	16	188.8 mA
Two	10	14	171.0 mA
Three	10	35	165.4 mA
Average	10	1	174.6 mA

Table 5.9: Obtained uptime of one of the Android smartphones running a continuous voice transmission.

5.7.3 Continuous SMS transmission

The smartphones should be able to send out text messages and we want to check how much impact sending these short messages had on the power drain of the smartphones. We used two smartphones and customized the Android application to continuously send out an SMS to the recipient smartphone every 60 seconds. The application code to send SMS every 60 seconds can be viewed in Listing B.3.

In order for the transmitting phone to send SMS continuously every 60 seconds we found out that the screen cannot be turned off. When this happens, the Android system set the application in *onPause* mode. For our results in this section we emphasize that the screen has been on for the whole test scenario. This means that these results may vary from the other scenarios where the screen has been turned off.

To provide some comparisons we have also tested the lifetime of the phones when the application is running and the screen is on. This will then provide us with an estimate of how much the phone uses to transmit an SMS every 60 seconds. The experiment has been performed three times. The results presented in Table 5.10 show that the test achieved an average uptime of 6 hours and 22 minutes. Based on these result we can conclude that SMS does not require much energy since we achieved an average uptime of 7 hours and 11 minutes in the previously explained experiment presented in Table 5.8.

Experiment	Uptime		SMS	Power drain
	Hours	Minutes	count	(avg)
One	6	22	382	274.7 mA
Two	6	43	408	260.4 mA
Three	6	3	364	289.2 mA
Average	6	22	385	274.3 mA

Table 5.10: Obtained uptime on one Android smartphone continuously sending out SMS messages over one hop.

5.7.4 Comparison

The previous experiments allow us to do some comparison of the acquired results. We are able to see that the Android system itself does not require much energy to operate and that it is the application you run on the system that are determining. Figure 5.11 illustrate the different up times we have obtained from our experiments. The first column show the up time of the prototype case with connected Raspberry Pi and a TP-Link router. It is included to provide ease when determining how long the system in its whole could grant services in the network. As we can see from the figure, the system in itself would be able to provide service for approximately 17 hours, without any communication between connected nodes. This is of course not representative for a system that could be used in crisis situations, where there might be lots of communication between rescuers. The system would be able to provide continuous voice service for about 10 hours, which is not bad at all. Regular phones usually provide voice service about the same time, but in our case we are not dependent of any infrastructure after initial setup process.

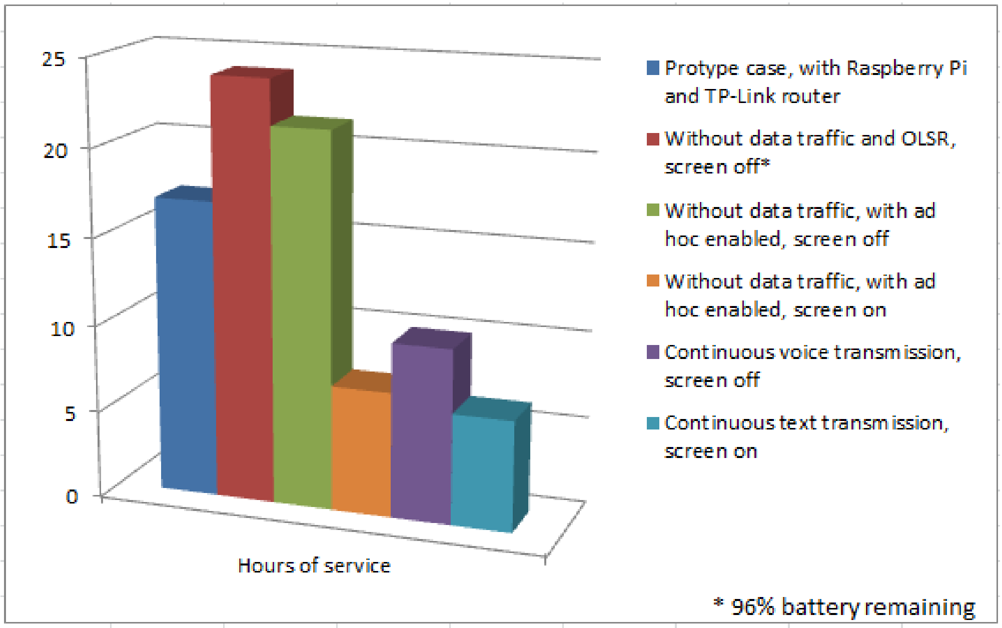


Figure 5.11: Obtained uptime of all the experiments done on one Android smartphone, also including the prototype case operational uptime.

5.8 Chapter Summary

In this chapter we have performed several experiments which validate our requirements for the system. We have done experiments that indicate how long the system is able to operate on battery power, as well as provided results that indicate the availability and delay when utilizing single and multi-hop communication. Furthermore, the system is able to provide hybrid IP address allocation, and the developed application handles the initial setup process. In order to simulate numerous connected nodes, we deployed stress testing on the system that indicates that our system is scalable for a large number of connected nodes.

Chapter 6

Discussions

In this chapter we will discuss several main features that has been considered and discovered during this project. We will discuss challenges regarding design, implementation and make some thoughts about the achieved results that has been obtained.

6.1 Related Projects

In order to provide suggestions towards future work, and point out differences between this project compared to other competitions, we need to look at other related projects.

In Chapter 2 we have mentioned a few of the most similar projects done in this field. The projects have some similarities related to our own work. The main difference is that our system is dependent on having a server that can be reachable in the network topology. By doing so, we are able to identify nodes in the network with names and valid IP addresses. The other projects have no method of handling this as we are aware of, and is one of the major differences.

In this project we have mentioned that this thesis project is based upon three previous projects. Since the main purpose was to enhance the current solution and make it easier and more appealing for others to use, we have designed a system that works around the existing solution and developed new features. From other known related projects we have the Serval project. One obvious advantage of the Serval project, is that they also support not having *ROOT* privileges. As well as utilizing SIP for handling of voice calls, making it possible to interconnect to regular PSTN network through a SIP trunk. Still, we believe that the OLSR protocol is in general a better option to use than the BATMAN protocol. Some studies indicate that OLSR produces more overhead, but performs overall better than BATMAN [32].

6.2 Hardware Limitations

In this project we have used several types of hardware in order to establish the system. Even though all of these devices supports ad hoc to some extent, all of the devices has been modified from stock kernel and distribution. The Android devices had to be given *ROOT* privileges and custom ROM to be functioning correctly.

The routers was flashed from stock distribution to *OpenWRT*, where we had to install and manually configure the OLSR configuration parameters. Even though our study shows that the system works, it currently restrict us to some degree to let anyone use this system due to the amount of configuration, which can be too complicated and confusing for a regular person. Still, by following the installation guide provided in Appendix A.2 this barrier could be narrowed but there are still need for some technical background to integrate the platform into our system.

To operate the prototype cases we require a battery pack that is able to power the system for an extended period of time. The results obtained in Chapter 5 shows that the system cases is able to operate for about 17 hours, and that the Android smartphones is able to operate for about 22 hours in idle mode. This means that in a crisis situation the battery powering will be the limited factor. Still, we are satisfied with the current available uptime of the prototype cases.

6.3 Software Limitations

For every software implementation, there is always some kind of bugs and restrictions that will affect the system to some extent. This also reflects on our project. Even though the system overall perform as we desire, there has been some features that has not been fixed and contains some kind of bug/issues that should be resolved. Some of these issues we are aware on how to fix, but due to time restriction we have not been able to fix all of them. In Appendix E we have listed bugs features that we are aware of.

6.4 Result Observations

From the previous Master's thesis [40], they performed a lot of experiments on audio quality. Since this has been done before, we decided to focus more on how the system entities performed on battery by testing, delay, operational-time and throughput. Due to the large time-consumption to obtain results for each test, we had to limit some of the experiments to three test per experiment. This limits us to not give certain statements on the results. However they give a solid indication of how the system will perform. For instance the results that was achieved in Subsection 5.5.2 shows that simulated behaviour of typically ad hoc network reflects our real-life results.

Chapter 7

Conclusions and Further Work

In this chapter we will present our conclusion, contributions, and make suggestions to possible further work for this project. The conclusion will be based upon what we have discussed from Chapter 6. We will also propose our objective view on how the project overall accomplishment regarding the project goals have been achieved. For the contributions, we will explain what major features we have done compared to the previously project and how this can be used as a standalone system and self-configurable. Finally, we will provide some main features that applies for further work.

7.1 Conclusions

This report demonstrates an implemented and validated system which improves the state-of-the-art technology for D2D communication in crisis management. We have provided numerous test results which have confirmed that our system can be deployed in both indoor and outdoor environments and function as designed. For testing we focused on indoor environment with line-of-sight and connectivity across walls, and demonstrated that the developed solution functions. By designing and implementing the temporary infrastructure system we have showed that the Android application is capable of providing multiple services.

We believe that using ad hoc mode with OLSR offers a lot of exclusive possibilities for the Android platform. Our system works as a stable prototype and can be used as an example on how to create a separate and private communication network that does not rely on any ISP or telecommunication provider. We personally believe that with the developed UI and service capabilities, this system provides a proper and solution to crisis management for rescue responders, as well as in conflict areas where communication with other parties can be of great importance.

7.2 Contributions

Throughout this thesis, we have designed and implemented a new prototype system based on existing OLSR implementation on Android devices. We have minimized and accomplish to only use one application instead of two, which was initially required. The Android application is deployed with an new UI, which makes it more appealing and easier to use and manage.

The results shows us by deploying the system in an temporary infrastructure mode that the Android devices can authenticate themselves to our system server, and obtain a valid IP address that can be used for communication with other devices. In addition, when each device has obtained the valid address, we have showed by activity-diagrams, pictures and detailed information that the nodes is able to automatically find and populate the users in the main view of the application.

The following list below summarizes in short term our contributions in this thesis project.

- We have designed and implemented a prototype system, which includes an Raspberry Pi, multiple routers and several Android devices.
- We have developed Android application with integrated support of ad hoc mode and user registration.
- We have developed hybrid IP allocation mechanism to an Raspberry Pi server and user handling.

- We have developed support for multiple services for sending SMS, MMS and video between the devices.
- We have improved, redesigned and developed Android application GUI.

7.3 Further Work

During this Master's thesis, we have made a prototype that has performed as intended. Still, there are some features that we would like to supplement and optimize.

- The Android application should be deployed with a chat feature, where all nodes can view and participate in.
- SMS, MMS and video should be locally stored and viewed later if desired.
- Services like video streaming and interconnection to regular PSTN network could be a useful feature.
- The user should be given the opportunity to change its nickname and logo, where the server and neighbour nodes would be notified.

Finally, we would like to make all communication *end-to-end* encrypted within devices and server in order to provide more anonymity and privacy.

Bibliography

[1] A. Asadi, Q. Wang, and V. Mancuso, "A survey on device-to-device communication in cellular networks," *Communications Surveys Tutorials, IEEE*, vol. 16, no. 4, pp. 1801–1819, Fourthquarter 2014.

[2] Bloomberg, "Google buys android for its mobile arsenal," August 2005. [Online]. Available: http://www.bloomberg.com/bw/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal

[3] BuiltWith, "Web server usage statistics." [Online]. Available: http://trends.builtwith.com/web-server

[4] F. Carli, "Security issues with dns," 2003. [Online]. Available: http://www.sans.org/reading-room/whitepapers/dns/security-issues-dns-1069

[5] S. Cheshire and M. Krochmal, *RFC 6762: Multicast DNS*, February 2013. [Online]. Available: http://www.rfc-editor.org/info/rfc6762

[6] Clas-Ohlson, "Water proof suitcase." [Online]. Available: http://www.clasohlson.com/no/Koffert/Pr314142000

[7] J. Damas, M. Graff, and P. Vixie, *STD 75, RFC 6891: Extension Mechanisms for DNS (EDNS(0)*, April 2013. [Online]. Available: http://www.rfc-editor.org/info/rfc6891

[8] Droidview, "Samsung galaxy s3 download mode." [Online]. Available: http://droidviews.com/wp-content/uploads/2012/06/Galaxy-S-III-Download-Mode.jpg

[9] D. Eastlake 3rd, *BCP 42, RFC 6895: Domain Name System (DNS) IANA Considerations*, April 2013. [Online]. Available: http://www.rfc-editor.org/info/rfc6895

[10] P. Gardner-Stephen and S. Palaniswamy, "Serval mesh software-wifi multi model management," in *Proceedings of the 1st International Conference on Wireless*

Technologies for Humanitarian Relief, ser. ACWR '11. New York, NY, USA: ACM, 2011, pp. 71–77. [Online]. Available: http://doi.acm.org/10.1145/2185216.2185245

[11] Google, "Android, the world's most popular mobile platform." [Online]. Available: http://developer.android.com/about/index.html

[12] ——, "Managing your app's memory." [Online]. Available: https://developer.android.com/training/articles/memory.html

[13] ——, "Sharedpreferences." [Online]. Available: http://developer.android.com/reference/android/content/SharedPreferences.html

[14] GSMarena, "Samsung nexus spesification." [Online]. Available: http://www.gsmarena.com/samsung_galaxy_nexus_i9250-4219.php

[15] Intocircuit, "Power castle." [Online]. Available: http://www.hisgadget.com/product/intocircuit-power-castle-26000mah-2/

[16] kiwi electronics, "Raspberry pi b+ image." [Online]. Available: http://www.kiwi-electronics.nl/image/cache/data/products/raspberry-pi/boards/RPI-MOD-B+512MB-1-800x533.jpg

[17] K. Koumidis, P. Kolios, C. Panayiotou, and G. Ellinas, "Resilient device-to-device communication in emergency situations," 2015.

[18] E. KuŽtans, "Shark for root." [Online]. Available: https://play.google.com/store/apps/details?id=lv.n3o.shark

[19] J. Li, C. Blake, D. S. J. D. Couto, H. I. Lee, and R. Morris, "Capacity of ad hoc wireless networks," *Mobicom*.

[20] T. Link, "Tp link wdr3600 image." [Online]. Available: http://www.tp-link.com/resources/images/products/large/TL-WDR3600-UN-V1-03.jpg

[21] M. Lottor, *RFC 1033: Domain Administrators Operations Guide*, November 1987. [Online]. Available: http://www.rfc-editor.org/info/rfc1033

[22] Q. T. Minh, K. Nguyen, C. Borcea, and S. Yamada, "On-the-fly establishment of multi-hop wireless access networks for disaster recovery," *Communications Magazine, IEEE*, vol. 52, no. 10, pp. 60–66, October 2014.

[23] P. Mockapetris, *RFC 883: Domain names: Implementation specification*, November 1983. [Online]. Available: http://www.rfc-editor.org/info/rfc883

[24] ——, *RFC 973: Domain system changes and observations*, January 1986. [Online]. Available: http://www.rfc-editor.org/info/rfc973

[25] ——, *STD 13, RFC 1034: Domain names - concepts and facilities*, November 1987. [Online]. Available: http://www.rfc-editor.org/info/rfc1034

[26] ——, *STD 13, RFC 1035: Domain names - implementation and specification*, November 1987. [Online]. Available: http://www.rfc-editor.org/info/rfc1035

[27] Netcraft, "Web server survey," January 2015. [Online]. Available: http://news.netcraft.com/archives/2015/01/15/january-2015-web-server-survey.html

[28] H. Nishiyama, M. Ito, and N. Kato, "Relay-by-smartphone: realizing multihop device-to-device communications," *Communications Magazine, IEEE*, vol. 52, no. 4, pp. 56–65, April 2014.

[29] Ookla, "Ookla speedtest for android." [Online]. Available: http://www.speedtest.net/mobile/android/

[30] Oracle, "Class math." [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

[31] Raspi.tv, "How much less power does the raspberry pi b+ use than the old model b?" [Online]. Available: http://raspi.tv/2014/how-much-less-power-does-the-raspberry-pi-b-use-than-the-old-model-b

[32] D. S. Sandhu and S. Sharma, "Performance evaluation of batman, dsr, olsr routing protocols - a review," *International Journal of Emerging Technology and Advanced Engineering*, vol. 2, no. 1, p. 4, January 2012. [Online]. Available: http://www.ijetae.com/files/Volume2Issue1/IJETAE_0112_35.pdf

[33] F. Sauer, "Eclipse metric." [Online]. Available: http://metrics.sourceforge.net/

[34] SPAN, "Manet manager," July 2012. [Online]. Available: https://play.google.com/store/apps/details?id=org.span

[35] M. Stensrud and H. G. Lie, "D2d communication: Temporarily deployed infrastructure with enhanced secure services," University of Agder, Tech. Rep., 2014.

[36] M. Stensrud and H. Nergaard, "Dynamic address and user interface enhancements for android-based smartphone d2d communication: Implementation and experiments," University of Agder, Tech. Rep., 2014.

[37] Tcpdump, “Official homepage of the tcpdump tool.” [Online]. Available: http://www.tcpdump.org/

[38] TPLink, “Wdr3600 spesification.” [Online]. Available: http://www.tp-link.com/en/products/details/cat-9_TL-WDR3600.html#specifications

[39] P. Vixie, Ed., S. Thomson, Y. Rekhter, and J. Bound, *RFC 2136: Dynamic Updates in the Domain Name System (DNS UPDATE)*, April 1997. [Online]. Available: http://www.rfc-editor.org/info/rfc2136

[40] M. Wennberg and N. E. Skj∅nsberg, “Mobile to mobile communication with or without temporary infrastructure prototype implementation and real-life experiments,” Master’s thesis, University of Agder, 2013.

Appendices

Appendix A

Installation Guide

This appendix include the installation guide on how to set up the Raspberry Pi enabling server, with corresponding configuration files. How to enable everything needed to provide a fully configurable system. Further it covers the installation of the OLSR enabled routers, and how they should be set up to function as expected. Based on the information made available here, you should be able to set up a fully functional system that corresponds to our designed backbone network. Some minor configurations may have been neglected, but they should not have a major impact on the configuration itself.

A.1 Raspberry Pi based Hybrid IP Allocation Server

Here we describe how to set up the different part of the system, in order to have a fully functional embedded server. First we will describe how to install the OS system, before we move on to the dependencies that are required to have a complete system.

A.1.1 Installing the OS

1. Download the latest Raspbian release:
   `https://www.raspberrypi.org/downloads/`

2. Write the downloaded filesystem to a microSD card, if on windows you could use:
   `http://sourceforge.net/projects/win32diskimager/`

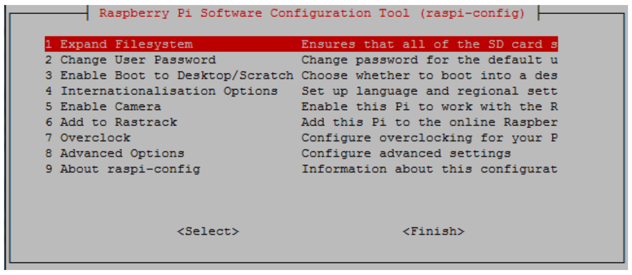
3. Boot up the Raspberry Pi for the first time, and follow the next first boot configurations, illustrated in Figure A.1-A.4.

   (a) Expand the filesystem. Illustrated in Figure A.1

   (b) Enter, Enable Boot to Desktop/Scratch and disable the graphical interface. Illustrated in Figure A.2

   (c) Enter, Internationalisation Options, and set the desired keyboard layout and timezone. Illustrated in Figure A.3

   (d) Finnish the initial setup and reboot the Raspberry Pi. Illustrated in Figure A.4

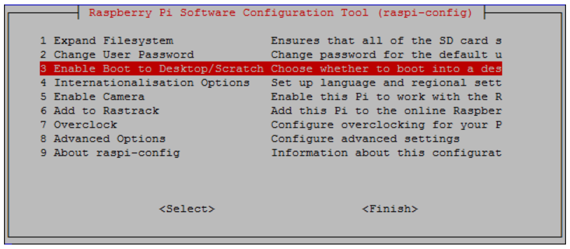


```
Raspberry Pi Software Configuration Tool (raspi-config)

1 Expand Filesystem              Ensures that all of the SD card s
2 Change User Password           Change password for the default u
3 Enable Boot to Desktop/Scratch Choose whether to boot into a des
4 Internationalisation Options   Set up language and regional sett
5 Enable Camera                  Enable this Pi to work with the R
6 Add to Rastrack                Add this Pi to the online Raspber
7 Overclock                      Configure overclocking for your P
8 Advanced Options               Configure advanced settings
9 About raspi-config             Information about this configurat

                <Select>                     <Finish>
```

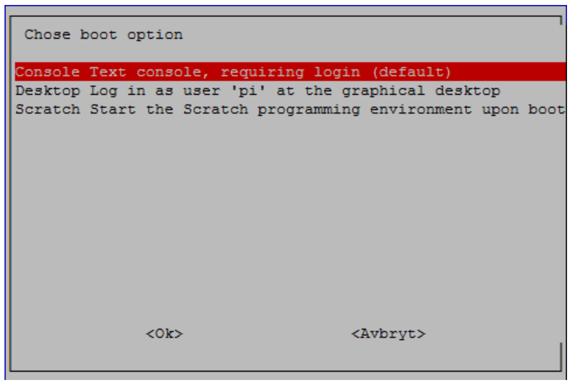
Figure A.1: Raspberry Pi install: Extend filesystem



```
Raspberry Pi Software Configuration Tool (raspi-config)

1 Expand Filesystem                Ensures that all of the SD card s
2 Change User Password             Change password for the default u
3 Enable Boot to Desktop/Scratch   Choose whether to boot into a des
4 Internationalisation Options     Set up language and regional sett
5 Enable Camera                    Enable this Pi to work with the R
6 Add to Rastrack                  Add this Pi to the online Raspber
7 Overclock                        Configure overclocking for your P
8 Advanced Options                 Configure advanced settings
9 About raspi-config               Information about this configurat

                 <Select>                    <Finish>
```

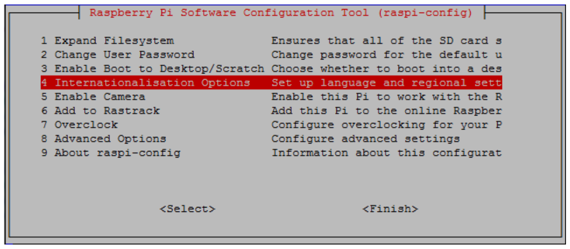


```
Chose boot option

Console Text console, requiring login (default)
Desktop Log in as user 'pi' at the graphical desktop
Scratch Start the Scratch programming environment upon boot

              <Ok>                    <Avbryt>
```

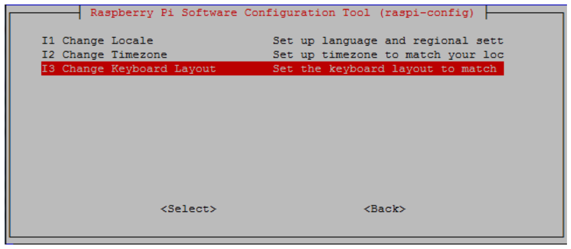
Figure A.2: Raspberry Pi install: Enable boot to desktop/scratch



```
Raspberry Pi Software Configuration Tool (raspi-config)

1 Expand Filesystem                Ensures that all of the SD card s
2 Change User Password             Change password for the default u
3 Enable Boot to Desktop/Scratch   Choose whether to boot into a des
4 Internationalisation Options     Set up language and regional sett
5 Enable Camera                    Enable this Pi to work with the R
6 Add to Rastrack                  Add this Pi to the online Raspber
7 Overclock                        Configure overclocking for your P
8 Advanced Options                 Configure advanced settings
9 About raspi-config               Information about this configurat

                 <Select>                    <Finish>
```

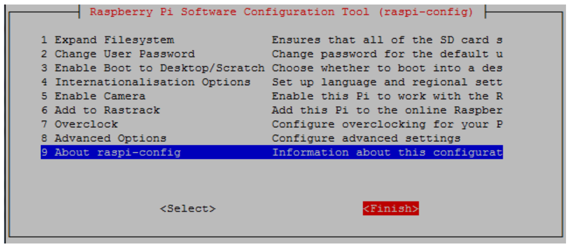


```
Raspberry Pi Software Configuration Tool (raspi-config)

I1 Change Locale                   Set up language and regional sett
I2 Change Timezone                 Set up timezone to match your loc
I3 Change Keyboard Layout          Set the keyboard layout to match

                 <Select>                    <Back>
```

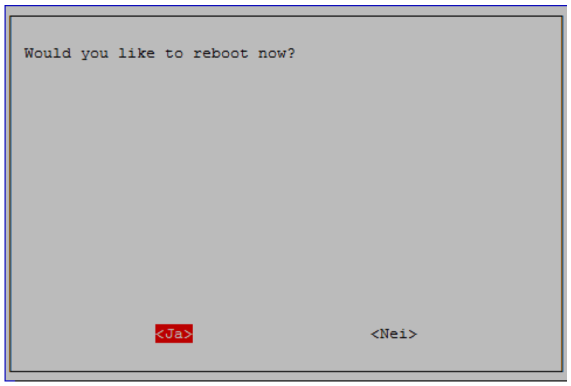
Figure A.3: Raspberry Pi install: Internationalisation options



```
Raspberry Pi Software Configuration Tool (raspi-config)

1 Expand Filesystem                Ensures that all of the SD card s
2 Change User Password             Change password for the default u
3 Enable Boot to Desktop/Scratch   Choose whether to boot into a des
4 Internationalisation Options     Set up language and regional sett
5 Enable Camera                    Enable this Pi to work with the R
6 Add to Rastrack                  Add this Pi to the online Raspber
7 Overclock                        Configure overclocking for your P
8 Advanced Options                 Configure advanced settings
9 About raspi-config               Information about this configurat

                 <Select>                    <Finish>
```



```
Would you like to reboot now?

              <Ja>                    <Nei>
```

Figure A.4: Raspberry Pi install: Finalize configuration

A.1.2 Updating the OS

After the reboot, connect your device to a Ethernet cable, allowing it access to the Internet. And issue the following commands in the terminal to update the system.

```
# apt-get update
# apt-get -y upgrade
# apt-get -y dist-upgrade
```

A.1.3 Installing the LAMP package components and phpMyAdmin

The next to do is to install all the required packages that would enable the system to operate as expected. By issuing the following commands, we will install the entire LAMP package along with phpMyAdmin.

- When promted, select Apache2 as the web server.
- Use dbconfig-common to set up the database.

```
# apt-get update
# apt-get -y install apache2 mysql-server php5-mysql php5 libapache2-mod-php5 phpmyadmin
    apache2-utils
# service apache2 restart
```

A.1.4 Configuring installed packages

After successful install of all the previously explained packages we need to do some minor configurations.

phpMyAdmin

First we configure phpMyAdmin to be more secure. This is done by modifying the generated apache2 config file, creating an restriction and user file. Follow the following guide and you will achieve a secured phpMyAdmin configuration.

First edit the apache2 configuration file:

```
# pico /etc/apache2/conf-available/phpmyadmin.conf

Add AllowOverride All as following:

<Directory /usr/share/phpmyadmin>
    Option FollowSymLinks
    DirectoryIndex index.php
    AllowOverride All
```

Secondly, create a .htaccess file with the following restriction rules:

```
# pico /usr/share/phpmyadmin/.htaccess
%% Insert the following %%

AuthType Basic
AuthName "Restricted Files"
AuthUserFile /etc/phpmyadmin/.htpasswd
Require valid-user
```

The next thing to do is to generate the .htpasswd file, with your own user credentials.

```
# htpasswd -c /etc/phpmyadmin/.htpasswd username
%% Additional user can be added by issuing the following command;
# htpasswd /etc/phpmyadmin/.htpasswd username
```

Finally we need to restart the apache2 server by issuing the following command.

```
# service apache2 restart
```

Now you should be promted for a valid username and password when trying to reach the graphical phpMyAdmin interface located at:

`http://local_ip/phpmyadmin`

Apache2 configurations

To filter out what are available through the http interface, we need to do some configuration on the web server. First we enable customated .htaccess files by changing the preferences in the apache2.conf file.

```
pico \etc\apache2\apache2.conf
```

Find the listing of the directory where your web files are located and modify the AllowOverride to All as following:

```
 AllowOverride All
```

Next we create some .htaccess files to prevent listing of the directories.

In the directories that we do not want to be listed create the following file with the following commands.

```
#pico .htaccess

%%And insert the following text%%

 Option −Indexes
```

MySQL database

If you forget your *ROOT* password it can be reset by first stopping the server, creating a recovery file and issuing a failsafe.

First locate the PID of your MySQL database server and kill it.

```
# top | grep mysql
# kill PID
```

Secondly you need to create a recovery file with the following statement:

```
#pico failsafe
%% Insert the following statement:%%

SET PASSWORD FOR 'root'@'localhost' = PASSWORD('MyNewPass');
```

Finally you need to issue the failsafe, by starting the server and pointing to the failsafe file.

```
# mysqld_safe −−init−file=/DIRECTORY_WITH_FILE/failsafe &
```

Delete the file after reset is done.

```
# rm /DIRECTORY_WITH_FILE/failsafe
```

A.2 OLSR Routers

The TP-Link routers need to be flashed with the custom firmware OpenWRT. In order to do so you need to log into the stock router and upload the new firmware, this is done by following the following step by step guide:

- Download the latest OpenWRT rom: `https://downloads.openwrt.org/barrier_breaker/14.07/ar71xx/generic/`
- Log on to the stock router, available at `http://tplinklogin.net` with login: *admin* and password: *admin*
- Browse to the firmware upload page, and flash the downloaded rom.
- Wait for the process to be done.

The router is now available on the local IP address: *192.168.1.1*, so point your browser at this IP and we will do some configuration to set up the system as desired.

A.2.1 Install necessary packages

Browse to the software section found at: *System* ⇒ *Software*. Select *Update lists* and wait for the lists to be updated. Next install the following packages: olsrd, luci-app-olsr, luci-app-olsr-viz, olsrd-mod-arprefresh, olsrd-mod-dyn-gw, olsrd-mod-httpinfo, olsrd-mod-jsoninfo, olsrd-mod-nameservice, olsrd-mod-p2pd, olsrd-mod-txtinfo, luci-lib-json, nano.

With all the packages installed, reboot the router.

A.2.2 Configure interfaces

Now we need to configure the interfaces of the router, browse to the interface section found at: *Network* ⇒ *Interfaces*. First remove the unnecessary interfaces and create a new OLSR interface. Configure the interfaces as following:

OLSR interface

Protocol: Static address

IPv4 address: 10.0.255.x

IPv4 netmask: 255.255.255.0

IPv4 gateway: 192.168.0.165
if 3G/4g adapter is installed

IPv4 broadcast: 10.0.0.255

IPv6 assignment length: disabled

DHCP Disabled.

LAN interface

Protocol: Static address

IPv4 address: 192.168.0.x

IPv4 netmask: 255.255.255.0

IPv6 assignment length: disabled

DHCP Enabled.

A.2.3 Configure OLSR

In order to enable the OLSR interface, first navigate the the OLSR configuration page located at: *Services ⇒ OLSR IPv4*. Here we first configure the plugins, and enable all of them. Next configure the HNA announcement to announce 192.168.0.0 255.255.255.0. Finally make sure that the OLSR interface is enabled, at the bottom of the page. HNA announcement should announce the net and subnet that the DHCP server of the respective LAN interface.

A.3 Android Smartphones

A.3.1 ROOT privliges and custom kernel

Providing *ROOT* privileges to Nexus and Samsung Android phones is generally easy. This section will cover how to *ROOT* and flash a customized kernel to the Galaxy S3 phone. The devices that is running need to be flashed with a customized kernel in order to make the OLSR work properly. First, we need to download some general tools that is required to *ROOT* and flash a new kernel to the device:

I Download and install Samsung Kies (Samsung official program). In order for the computer to recognize the Galaxy S3 phone, we need to install Samsung Kies. This program will identify the phone and download the required drivers.
`http://www.samsung.com/no/support/usefulsoftware/KIES/`.

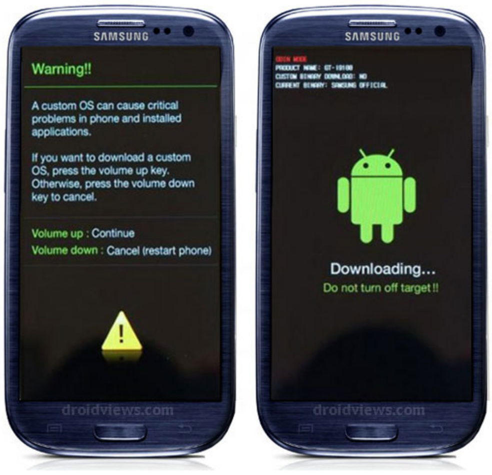
II Download and install Odin. This program is used for creating *ROOT* on the phone.
`http://odindownload.com/`

III Download SuperSu and move it to the Sdcard of the Galaxy S3 phone. . The SuperSu application can be deployed on the phone when *ROOT* has been provided. The SuperSu application handles and decided which application gets *ROOT* privileges.
`http://forum.xda-developers.com/showthread.php?t=1538053`

IV Download kernel that supports OLSR
`https://github.com/monk-dot/SPAN/tree/master/kernels`

V Download *ROOT* File for Galaxy S3
`https://autoroot.chainfire.eu/`.

Once all of the above tools is downloaded and installed, we can set the phone in *Download Mode*. The Odin program requires the phone to be in this mode when we are granting the phone *ROOT*. To enter the *Download mode*, we need to start the phone by holding in the following buttons on the phone: *Volume Down Button + Home Button + Power Button*. If this is done correctly the phone should show as Figure A.5 illustrates.



Warning!!

A custom OS can cause critical problems in phone and installed applications.

If you want to download a custom OS, press the volume up key. Otherwise, press the volume down key to cancel.

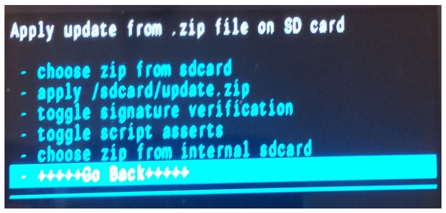
Volume up : Continue
Volume down : Cancel (restart phone)

Downloading...
Do not turn off target !!

Figure A.5: Galaxy S3 phone in download mode [8].

Next we launch the Odin program. Here we select the *PDF* button, and find the *ROOT* file that we previously downloaded. Make sure that *Auto Reboot* and *F. Reset Time* is enabled. Then press *Start*. After a while the phone should indicate that the installation went successful.

The device has now installed an custom bootloader that we can access and install the custom ROM and kernel. By pressing *Volume Up button + Home Button + Power Button* the phone will now enter the custom bootloader.Illustration if shown in Figure A.6. From the bootloader we can install both ROM and kernel by selecting *Choose zip from sdcard*. When this is done, the installation is complete and the phone is properly configured.



```
Apply update from .zip file on SD card

 - choose zip from sdcard
 - apply /sdcard/update.zip
 - toggle signature verification
 - toggle script asserts
 - choose zip from internal sdcard
 - +++++Go Back+++++
```

Figure A.6: Illustration of CWM bootloader of Galaxy S3.

Appendix B

Android Snippets and Logs

B.1 Tests

B.1.1 Debugging logs

Listing B.1: Debugging log of first setup process

```
04-27 21:18:48.325: Welcome to D2D setup!
04-27 21:19:00.105: Get user information and store it in sharedPreferences for later use.
04-27 21:19:18.315: Checking Image size. It is 425 KB
04-27 21:19:18.320: Image is > 100KB and is TOO BIG! Start compressing
04-27 21:19:18.470: CREATE TABLE if not exists Friends (_id integer PRIMARY KEY
    autoincrement,firstname,lastname,nickname,ip,ttl,imageUrl,imagestatus BLOB);
04-27 21:19:18.470: CREATE TABLE if not exists User (_id integer PRIMARY KEY autoincrement,
    firstname,lastname,username);
04-27 21:19:18.505: User credential has now been saved
04-27 21:19:18.730: Application is registering user to server. This will take some time...
04-27 21:19:18.730: User Data | Firstname: karl lastname: hansen username: karhan ip:
    10.0.20.254
04-27 21:19:18.755: MESSAGE SERVICE STARTED!!
04-27 21:19:21.110: onConfigUpdated
04-27 21:19:23.535: onAdhocStateUpdated
04-27 21:19:23.535: removeDialog
04-27 21:19:23.545: onServiceConnected
04-27 21:19:23.545: removeDialog
04-27 21:19:23.545: Service Connected. Restarting MANET to use the new config!
```

```
04-27 21:19:24.405: Initializing Ad hoc with OLSR
04-27 21:19:40.130: LoadRegister: onPeersUpdated
04-27 21:19:42.400: onAdhocStateUpdated
04-27 21:19:42.400: removeDialog
04-27 21:19:50.430: Waiting. Timer 1: 0 min, 1 sec
04-27 21:19:51.345: LoadRegister: onPeersUpdated
04-27 21:19:52.415: Obtain valid IP from server
04-27 21:19:52.415: Register user by sending data to addUser script
04-27 21:19:52.415: UpdateIP executed!
04-27 21:19:52.430: Waiting. Timer 2: 0 min, 6 sec
04-27 21:19:52.625: 200. Request approved
04-27 21:19:52.625: Received data is: 200
04-27 21:19:52.710: Did not receive reply or unknown message sent by server
04-27 21:19:52.710: Received data is: 10.0.0.9
04-27 21:19:57.430: Waiting. Timer 2: 0 min, 1 sec
04-27 21:19:58.915: check if any errors occoured!
04-27 21:19:58.970: onConfigUpdated
04-27 21:20:04.200: onAdhocStateUpdated
04-27 21:20:04.200: removeDialog
04-27 21:20:14.805: LoadRegister: onPeersUpdated
04-27 21:20:17.165: onAdhocStateUpdated
04-27 21:20:17.165: removeDialog
04-27 21:20:17.170: User is registered and valid IP (10.0.0.9) is received.
04-27 21:20:17.170: Setup is finished. Enter MainActivity.
04-27 21:20:17.180: Going into MainActivity!
04-27 21:20:17.305: address == 10.0.20.254
04-27 21:20:17.510: Main: main and communicationMaster recreated
04-27 21:20:17.520: CommunicationMaster Started!
04-27 21:20:17.520: CommunicationListener Started!
04-27 21:20:17.635: MESSAGE SERVICE STARTED!!
04-27 21:20:17.635: MMSSERVICE STARTED!!
04-27 21:20:17.635: Server started. Listening to the port 4444
04-27 21:20:17.635: RECEIVEVIDEO SERVICE STARTED!!
04-27 21:20:17.635: Server started. Listening to the port 4445
04-27 21:20:17.645: onServiceConnected
04-27 21:20:17.645: removeDialog
04-27 21:20:17.645: Service Connected. Restarting MANET to use the new config!
04-27 21:20:22.310: onConfigUpdated
04-27 21:20:22.585: onAdhocStateUpdated
04-27 21:20:22.585: removeDialog
04-27 21:20:22.640: onConfigUpdated
04-27 21:20:22.860: onAdhocStateUpdated
04-27 21:20:22.865: removeDialog
04-27 21:20:28.875: setIpAddress == 10.0.0.9
04-27 21:20:28.885: We still have the randomizedIP in the OLSR service. Restart and use the
    new IP provided from the Server.
04-27 21:20:28.885: our new final IP is: 10.0.0.9
```

```
04-27 21:20:36.580: Main: main and communicationMaster recreated
04-27 21:20:36.590: CommunicationMaster Started!
04-27 21:20:36.595: CommunicationListener Started!
04-27 21:20:36.605: MMSSERVICE STARTED!!
04-27 21:20:36.605: Server started. Listening to the port 4444
04-27 21:20:36.610: RECEIVEVIDEO SERVICE STARTED!!
04-27 21:20:36.610: Server started. Listening to the port 4445
04-27 21:20:36.610: MESSAGE SERVICE STARTED!!
04-27 21:20:36.610: Server started. Listening to the port 9000
04-27 21:20:43.245: onPeersUpdated
04-27 21:20:43.245: HashSet with Peers is empty!
04-27 21:20:45.275: onAdhocStateUpdated
04-27 21:20:52.270: Do not restart. We have the correct IP! Just send Start Command to the
    OLSR service!
04-27 21:20:55.040: onAdhocStateUpdated
04-27 21:20:55.550: onPeersUpdated
04-27 21:20:55.560: Ip dosen't exist in database | 10.0.0.3
04-27 21:20:55.565: Send request on ip 10.0.0.3 to address:
04-27 21:20:55.580: GetNick executed!
04-27 21:20:55.675: Adding User into Database: Name: christoffer Lastname: stensrud Nickname
    : frisco IP: 10.0.0.3 Expires: 100 Image: http://192.168.0.165/images/frisco.jpg
```

B.1.2 Snippet code

Listing B.2: Server reply class

```
/**
 * Class for handeling Server replies,
 * Checks the reply from the server, and logs accordingly to this.
 *
 * @author Michael Stensrud
 */
public class ServerReply {

    Identifiers ID = new Identifiers();
    String TAG = "ServerReply.class";

    public ServerReply(String Reply) {

        if(Reply.equals("200"))       { Log.i(TAG, ID.OK200);      }
        else if (Reply.equals("401")) { Log.i(TAG, ID.Error401);   }
        else if (Reply.equals("500")) { Log.i(TAG, ID.Error500);   }
        else if (Reply.equals("500a")){ Log.i(TAG, ID.Error500a); }
        else if (Reply.equals("500d")){ Log.i(TAG, ID.Error500d); }
        else{ Log.i(TAG, ID.ErrorUnkwnown);}
    }
}
```

Listing B.3: SMS test source code

```java
smsTest inner = new smsTest();
inner.start();

private class smsTest extends Thread {
    public void run() {
        while (true) {
            try {
              sleep(60000);
              String msg = firstname + " " + lastname + " (" + app.manetcfg.getIpAddress() +
                    ") " + "\n" + " \n Hey, \n Check out the picture that I've sent you! ";
                String retval = null;
                try {
                    smsCounter++;
                    System.out.println("sending request number: " + smsCounter);
                    SharedPreferences settings = getSharedPreferences(MainActivity.
                        PREFS_NAME, 0);
                    SharedPreferences.Editor editor = settings.edit();
                    editor.putInt("smsCounter", smsCounter);
                    editor.commit();

                    SendMessageTask task = new SendMessageTask();
                    task.execute(new String[] {address, msg});
                    retval = task.get();
                }
                catch (Exception e) {
                    retval = "Error: " + e.getMessage();
                    System.out.println(retval);
                }
            }
            catch (InterruptedException e) {
              throw new RuntimeException(e);
            }
        }
    }
}
```

Listing B.4: SMS auto reply

```java
/*
* SMS has been received on listening port 9000.
* Once message has been handeled, prepare by sending an autoreply back
*/
System.out.println("Start by sending reply back!");
String fromIP        = null;
SharedPreferences sp = getSharedPreferences("userSettings", Activity.MODE_PRIVATE);
```

```
String firstname     = sp.getString("FIRSTNAME",null);
String lastname      = sp.getString("LASTNAME", null);
String ip            = sp.getString("IP", null);
Matcher m            = Pattern.compile("\\(([^)]+)\\)").matcher(from);
while(m.find()) {
    fromIP = m.group(1);
 }
try {
    AutoReplyText task  = new AutoReplyText();
    String msg          = firstname + " " + lastname + " (" + ip +  ") " + "\n" +  " \n Hey,
         \n I Just received your awesome message!!!";
    task.execute(new String[] {fromIP, msg});
    System.out.println("Reply finished");
}
catch (Exception e) {
    System.out.println(e);
}

public class AutoReplyText extends AsyncTask<String, Void, String> {

    Identifiers ID = new Identifiers();

    @Override
    protected String doInBackground(String... params) {
        String address = params[0];
        String msg     = params[1];
        String retval  = sendMessage(address, msg);
        return retval;
    }
    private String sendMessage(String address, String msg) {

        String retval          = null;
        DatagramSocket socket = null;
        try {
            socket = new DatagramSocket();
            byte buff[] = msg.getBytes();
            int msgLen  = buff.length;
            boolean truncated = false;
            if (msgLen > MessageService.MAX_MESSAGE_LENGTH) {
                msgLen = MessageService.MAX_MESSAGE_LENGTH;
                truncated = true;
            }
            DatagramPacket packet = new DatagramPacket(buff, msgLen, InetAddress.getByName(
                address), ID.MessagePort);
            socket.send(packet);
            System.out.println("VERIFY! message sent to: " + InetAddress.getByName(address)
               + " packet size: " + packet.getLength() + " msglength: " + msgLen);
```

```
                if (truncated) {
                    retval = "Message truncated and sent.";
                } else {
                    retval = "Message sent.";
                }
            }
            catch (Exception e) {
                e.printStackTrace();
                retval = "Error: " + e.getMessage();
            }
            finally {
                if (socket != null) {
                    socket.close();
                }
            }
            return retval;
            }
    };
```

Appendix C

PHP Source Codes

This Appendix includes the written scripts that the application utilized to push and request information to the MySQL databases. The scripts include a short description on what is happening as in-line comments. It also include the scripts written for test and clean up service on the local Raspberry Pi embedded system.

C.1 PHP Source Code

C.1.1 Add a new user

Listing C.1: PHP add a new user

```php
<?php
/* ADD PERSON
 * What this file does is it:
 * 1) Creates connection to database.
 * 2) Retrieve the data being send.
 * 3) Add the retrieved data to database.
 * 4) Close database connection.
 */
error_reporting(0); //Turn error reporting off, for security reasons.
//1: Connect to a database/disconnect handler.
require_once 'db/connectDB.php';
```

```php
define('UPLOAD_DIR', 'images/'); //Define the user image directory

//2: Retrieve the data.
$name =$_POST['NAME'];
$lastname = $_POST['LASTNAME'];
$image = $_POST['IMAGE'];
$nick = $_POST['NICK'];

//Check that the user does not exist already
if($stmt = $db->prepare("Select COUNT(Nickname) FROM `Users` WHERE Nickname = :nick LIMIT
    0,1")){
    $stmt->bindParam(':nick',$nick, PDO::PARAM_STR);
    $stmt->execute() or die("500");
    if($result = $stmt->fetchColumn(0)){
        die("409");
    }
}
else{
    die('500');
}

//3: Create the new user.
if($stmt = $db->prepare("INSERT INTO `Users` (Name,Lastname,Nickname,Image)
    VALUES (:name,:lastname,:nick,:image) ON DUPLICATE KEY UPDATE Nickname=:nick")){ //
        Prepare the statement for security reasons.
        // Bind the parameters, such that they are available for the prepared statement.
        $stmt->bindParam(':name',$name, PDO::PARAM_STR);
        $stmt->bindParam(':lastname',$lastname, PDO::PARAM_STR);
        $stmt->bindParam(':nick',$nick, PDO::PARAM_STR);
        // Modify the image, such that it can be downloaded correctly.
        $image = str_replace('data:image/jpeg;base64,', '', $image);
        $image = str_replace(' ', '+', $image);
        $data = base64_decode($image);
        $imagename = $nick . '.jpg';
        $file = UPLOAD_DIR . $imagename;
        $tmp = 'http://192.168.0.165/images/' . $imagename;
        $stmt->bindParam(':image',$tmp, PDO::PARAM_STR); //Bind the absolutepath to the
            image.
       //Get remote image($data) and store it in local path($file).
        if(file_put_contents($file,$data) !== FALSE)
            $stmt->execute() or die("500"); // Execute the prepared statement to complete
                the user registry.
        else die('500');
}
else{
        die('500');//Return 500, if there is an internal server error.
}
```

```

echo '200'; //Return 200, if the registry is completed.

//4: Disconnect from database.
$db=null;
?>
```

C.1.2 Update IP address

Listing C.2: PHP update IP address

```
<?php
/* Update IP
 * What this file does is it:
 * 1) Creates connection to database.
 * 2) Retrieve the data being send.
 * 3) Check retrieved data with database entry and add if correct.
 * 4) Close database connection.
 */

error_reporting(0); //Turn error reporting off, for security reasons.
//1: Connect to a database/disconnect handler.
require_once 'db/connectDB.php';
define('IPSTART', '10.0.');

//2: Retrieve the data
$nick = $_POST['NICK'];
$ip = $_POST['IP'];

function endsWith($haystack, $needle) {
    // search forward starting from end minus needle length characters
    return $needle === "" || strpos($haystack, $needle, strlen($haystack) - strlen($needle))
        !== FALSE;
}

//Check that the nickname exists in the database
if($stmt = $db->prepare("Select COUNT(Nickname) FROM `Users` WHERE Nickname = :nick LIMIT
    0,1")){
    $stmt->bindParam(':nick',$nick, PDO::PARAM_STR);
    $stmt->execute() or die("500");
    if(!$result = $stmt->fetchColumn(0)){
        die("404"); // Return if the nick does not exist;
    }
}
else{
    die('500'); // Internal server error message.
```

```
}

//3: Check validity of IP address, and add if its valid.
// If randomed IP
if(endsWith($ip,'.254')){
    $counter = 1; $i = 0;
    $temp = IPSTART . $i . '.' . $counter;
    if($stmt = $db->prepare("Select Count(IP) FROM `Users` WHERE IP = :ip LIMIT 0,1")){
        $stmt->bindParam(':ip',$temp,PDO::PARAM_STR);
        $stmt->execute() or die('500');
    }
    else{
        die('500');
    }
    // While IP exist, increase counter.
    while($result = $stmt->fetchColumn(0)){
        if($counter == 253){
            $counter = 0; $i++;
            if(i==255) die('500');
        }
        $counter ++;
        $temp = IPSTART . $i . '.' . $counter;
        $stmt->bindParam(':ip',$temp,PDO::PARAM_STR);
        $stmt->execute() or die('500');
    }
    $ip = $temp;
    // Insert the correct IP into the database, corresponding to the correct nickname
    if($stmt = $db->prepare("INSERT INTO `Users` (Nickname, IP, Expires) VALUES (:nick,:ip,
        UNIX_TIMESTAMP(NOW() + INTERVAL 1 DAY)) ON DUPLICATE KEY UPDATE Expires=
        UNIX_TIMESTAMP(NOW() + INTERVAL 1 DAY),IP=:ip")){
        $stmt->bindParam(':nick', $nick ,PDO::PARAM_STR);
        $stmt->bindParam(':ip', $ip ,PDO::PARAM_STR);
        $stmt->execute() or die('500');
    }
    else{
        die('500');
    }
    echo $ip;
}
// If not a randomed IP
else{
    //Check validity of IP address
    if($stmt = $db->prepare("Select Count(IP) FROM `Users` WHERE IP = :ip AND Nickname = :
        nick LIMIT 0,1")){
        $stmt->bindParam(':ip', $ip, PDO::PARAM_STR);
        $stmt->bindParam(':nick', $nick, PDO::PARAM_STR);
        $stmt->execute() or die('500');
```

```php
        // if valid, update lease time.
        if($result = $stmt->fetchColumn(0)){
            if($stmt = $db->prepare("INSERT INTO `Users` (Nickname, IP, Expires) VALUES (:
                nick,:ip,UNIX_TIMESTAMP(NOW() + INTERVAL 1 DAY)) ON DUPLICATE KEY UPDATE
                Expires=UNIX_TIMESTAMP(NOW() + INTERVAL 1 DAY),IP=:ip")){
                $stmt->bindParam(':nick', $nick ,PDO::PARAM_STR);
                $stmt->bindParam(':ip', $ip ,PDO::PARAM_STR);
                $stmt->execute() or die('500');
            }
            else{
                die('500');
            }
            echo '200'; // Return 200 OK, idicating the the IP can be used.
        }
        // else, find new IP.
        else{
            $counter = 1; $i = 0;
            $temp = IPSTART . $i . '.' . $counter;
            if($stmt = $db->prepare("Select Count(IP) FROM `Users` WHERE IP = :ip LIMIT 0,1"
                )){
                $stmt->bindParam(':ip',$temp,PDO::PARAM_STR);
                $stmt->execute() or die('500');
            }
            else{
                die('500');
            }
            // While IP exist, increase counter.
            while($result = $stmt->fetchColumn(0)){
                if($counter == 253){
                    $counter = 0; $i++;
                    if(i==255) die('500');
                }
                $counter ++;
                $temp = IPSTART . $i . '.' . $counter;
                $stmt->bindParam(':ip',$temp,PDO::PARAM_STR);
                $stmt->execute() or die('500');
            }
            $ip = $temp;
            // Insert the correct IP into the database, corresponding to the correct
                nickname
            if($stmt = $db->prepare("INSERT INTO `Users` (Nickname, IP, Expires) VALUES (:
                nick,:ip,UNIX_TIMESTAMP(NOW() + INTERVAL 1 DAY)) ON DUPLICATE KEY UPDATE
                Expires=UNIX_TIMESTAMP(NOW() + INTERVAL 1 DAY),IP=:ip")){
                $stmt->bindParam(':nick', $nick ,PDO::PARAM_STR);
                $stmt->bindParam(':ip', $ip ,PDO::PARAM_STR);
                $stmt->execute() or die('500');
            }
            else{
                die('500');
            }
            echo $ip; // Return the new IP address
        }
    }
    else{
        die('500');
    }
}

//4: Close database connection
$db = null;

?>
```

C.1.3 Get user, by IP

Listing C.3: PHP get user by IP address

```
<?php
/* Get User by name
 * What this file does is it:
 * 1) Creates connection to database.
 * 2) Retrieve the data being send.
 * 3) Query the database and returns the result.
 * 4) Close database connection.
 */
error_reporting(0);
require_once 'db/connectDB.php'; //connect to a database.

//Retrieve the data.
$name = $_POST['NAME'];
$lastname = $_POST['LASTNAME'];
$nick = $_POST['NICK'];
$name = $name . "%";
$lastname = $lastname . "%";
$nick = $nick . "%";

$result_array = array();
if($stmt = $db->prepare("SELECT * FROM `Users` WHERE
                        `Nickname` LIKE :nick AND `Name` LIKE :name AND `Lastname` LIKE :
                            lastname LIMIT 10")){
    $stmt->bindParam(":nick",$nick, PDO::PARAM_STR);
    $stmt->bindParam(":name",$name, PDO::PARAM_STR);
    $stmt->bindParam(":lastname",$lastname, PDO::PARAM_STR);
    $stmt->execute() or die("500");

    while($row = $stmt->fetch(PDO::FETCH_ASSOC)){
        $result_array[] = $row;
    }
    echo json_encode($result_array);
}
else{
    die('500 prepare');
}

//Disconnect from database.
$db=null;
?>
```

C.1.4 Get user, by nickname

Listing C.4: PHP get IP of user

```
<?php
/* Get DNS
 * What this file does is it:
 * 1) Creates connection to database.
 * 2) Retrieve the data being send.
 * 3) Query the database and returns the result.
 * 4) Close database connection.
 */
error_reporting(0);
require_once 'db/connectDB.php';

//Retrieve the data.
$nick = $_POST['NICK'];

//Check that the nickname exists in the database
if($stmt = $db->prepare("Select COUNT(Nickname) FROM `Users` WHERE Nickname = :nick LIMIT
    0,1")){
    $stmt->bindParam(':nick',$nick, PDO::PARAM_STR);
    $stmt->execute() or die("500");
    if(!$result = $stmt->fetchColumn(0)){
        die("404"); //Return if the nick doesn't exist.
    }
}
else{
    die('500');
}
```

```php
// Retrieve the IP corresponding to the specific nickname
if($stmt = $db->prepare("SELECT IP FROM `Users` WHERE Nickname= :nick")){
    $stmt->bindParam(':nick',$nick, PDO::PARAM_STR);
    $stmt->execute() or die("500");
    $result = $stmt->fetchColumn(0);
}
else{
    die('500');
}
echo $result; // Return the IP address
//Disconnect from database.
$db=null;
?>
```

C.1.5 Update image

Listing C.5: PHP update user image

```php
<?php
error_reporting(0);
require_once 'db/connectDB.php';
define('UPLOAD_DIR', 'images/'); //Define the user image directory

//Retrieve the data.
$nick = $_POST['NICK'];
$image = $_POST['IMAGE'];

//Check that the nickname exists in the database
if($stmt = $db->prepare("Select COUNT(Nickname) FROM `Users` WHERE Nickname = :nick LIMIT
    0,1")){
    $stmt->bindParam(':nick',$nick, PDO::PARAM_STR);
    $stmt->execute() or die("500");
    if(!$result = $stmt->fetchColumn(0)){
        die("404");//Return if the nickname doesn't exist.
    }
}
else{
    die('500');
}

$image = str_replace('data:image/jpeg;base64,', '', $image);
$image = str_replace(' ', '+', $image);
$data = base64_decode($image);
$imagename = $nick . '.jpg';
$file = UPLOAD_DIR . $imagename;
$tmp = 'http://192.168.0.165/images/' . $imagename;
```

```
$stmt->bindParam(':image',$tmp, PDO::PARAM_STR); //Bind the absolutepath to the image.
//Get remote image($data) and store it in local path($file).
if(file_put_contents($file,$data) !== FALSE){
    echo '200';
}
else die('500');
}

$db = null;
?>
```

C.1.6 Check if nick exists

Listing C.6: PHP check if nick is available

```
<?php
/* Check Nickname
 * What this file does is it:
 * 1) Creates connection to database.
 * 2) Retrieve the data being send.
 * 3) Check retrieved data with database entry.
 * 4) Close database connection.
 */

error_reporting(0); //Turn error reporting off, for security reasons.
//1: Connect to a database/disconnect handler.
require_once 'db/connectDB.php';

//2: Retrieve the data
$nick = $_POST['NICK'];

//3: Check that the nickname does not exist already
if($stmt = $db->prepare("Select COUNT(Nickname) FROM `Users` WHERE Nickname = :nick LIMIT
    0,1")){
    $stmt->bindParam(':nick',$nick, PDO::PARAM_STR);
    $stmt->execute() or die("500");
    if($result = $stmt->fetchColumn(0)){
        die("409"); // Nick exists message;
    }
}
else{
    die('500'); // Internal server error message.
}

echo '200'; // Return 200, if nick does not already exist.
```

```
//4: Close database connection
$db = null;

?>
```

C.2 Testing and Clean-Up Scripts

C.2.1 Clean up expired IP addresses

Listing C.7: Clean up old IP addresses

```
<?php
/*
    Clean expired IP addresses
    1) Establish connection to MySQL database
    2) Check validity of IP address
    3) Remove invalid addresses
    4) Close database connection
*/

error_reporting(0); //Turn error reporting off, for security reasons.
//1: Connect to a database/disconnect handler.
try{
        $db = new PDO('mysql:dbname=d2dphone;host=127.0.0.1','d2dphone','awesome1');
}
catch(PDOException $ex){
        echo 'Connection failed: ' . $ex->getMessage();
}
//2: Check validity of IP addresses.
if($stmt = $db->prepare("Select Nickname FROM `Users` WHERE `Expires` > NOW()")){
    $stmt->execute() or die();
    $result = $stmt->fetchAll(PDO::FETCH_COLUMN, 0);
    //3: Remove invalid addresses.
    $stmt = $db->prepare("UPDATE `Users` SET IP='',Expires='' WHERE Nickname = :nick");
    foreach($result as $temp){
        $stmt->bindParam(':nick',$temp,PDO::PARAM_STR);
        $stmt->execute() or die();
    }
}
//4: Close database connection.
$db=null;
?>
```

C.2.2 Calculate Raspberry Pi uptime

Listing C.8: Log Raspberry Pi uptime

```
#!bin/bash

# Initialize the time variables
h=0
m=0
# Define the current time
now=$(date +"%m_%d_%Y")
# Run for ever
while :
do
    # Define the uptime and print to log file
    temp="Uptime: $h:$m"
    echo $temp > /home/pi/logs/counter_$now.log

    #slepp for 60 seconds
    sleep 60

    # Increase the minutes and hours counter variable.
    if [ $m -lt 59 ]; then
        m=$(($m +1))
    else
        m=0
        h=$(($h +1))
    fi
done
```

C.2.3 Stress test add user script

Listing C.9: Continuously add new users to database

```
<?php
$url = 'http://192.168.0.165/add_user.php';

$counter = 1500;
$latency = array();
$accepted = 0;
for ($x=0;$x<=$counter;$x++) {
    $rand = substr(md5(microtime()),rand(0,26),10);
    $rand1 = substr(md5(microtime()),rand(0,26),10);
    $rand2 = substr(md5(microtime()),rand(0,26),10);
    $data = array('NAME'=>$rand, 'LASTNAME'=>$rand1, 'IMAGE'=>'\var\www\image.jpg', 'NICK'=>
        $rand2);
    $options = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data),
    )
);

$context  = stream_context_create($options);
$latency[$x] = -microtime(true);
$result = file_get_contents($url, false, $context);
$latency[$x] += microtime(true);
    if ($result == '200'){
        $accepted++;
    }

}

$data = $accepted . " of " . $counter . " accepted\n" ;
file_put_contents('latency' . $argv[1] . '.txt',var_export($latency,true), FILE_APPEND |
    LOCK_EX);
file_put_contents('report' . $argv[1] . '.txt',$data, FILE_APPEND | LOCK_EX);

?>
```

C.2.4 Stress test get user by nickname

Listing C.10: Continuously retrieve user data by nickname

```
<?php
$url = 'http://192.168.0.165/get_ip.php';

$counter = 1500;
$latency = array();
$accepted = 0;
for ($x=0;$x<=$counter;$x++) {
    $data = array('NICK'=>'habbiz');
    $options = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data),
    )
);
```

```php

$context  = stream_context_create($options);
$latency[$x] = -microtime(true);
$result = file_get_contents($url, false, $context);
$latency[$x] += microtime(true);

}
file_put_contents('latency_get_ip' . $argv[1] . '.txt',var_export($latency,true),
    FILE_APPEND | LOCK_EX);
?>
```

C.2.5 Start add user stress test

Listing C.11: Spawn multiple instances of add new user stress

```php
<?php
    for($j=0;$j<10;$j++){
        $pipe[$j] = popen('php stress_add_user.php ' . $j,'w');
        sleep(1);
    }

    for($j=0;$j<10;++$j){
        pclose($pipe[$j]);
    }
?>
```

C.2.6 Start get ip stress test

Listing C.12: Spawn multiple instances of retrieve user data stress

```php
<?php
    for($j=0;$j<10;$j++){
        $pipe[$j] = popen('php stress_get_ip.php ' . $j,'r');
        sleep(1);
    }

    for($j=0;$j<10;++$j){
        pclose($pipe[$j]);
    }
?>
```

C.3 Simulation Scripts

Listing C.13: Add two new users to the database

```php
<?php
$url = 'http://127.0.0.1/add_user.php';
$data = array('NAME'=>'Donald Duck1','LASTNAME'=>'User','IMAGE'=>'abcdefghijklmnopqrst','
    NICK'=>'donald1dsg');
$options = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data),
    )
);

$context  = stream_context_create($options);
$result = file_get_contents($url, false, $context);
var_dump($result);

$data2 = array('NAME'=>'Test','LASTNAME'=>'User1','IMAGE'=>'abcdefghijklmnopqrst','NICK'=>'
    Testerdg');
$options2 = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data2),
    )
);

$context2  = stream_context_create($options2);
$result2 = file_get_contents($url, false, $context2);
var_dump($result2);

?>
```

Listing C.14: Check if nickname is already in the database

```php
<?php
$url = 'http://127.0.0.1/check_nick.php';
$data = array('NICK'=>'donald1dsg');
$options = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data),
    )
);

$context  = stream_context_create($options);
$result = file_get_contents($url, false, $context);
var_dump($result);

$data2 = array('NICK'=>'Tester');
$options2 = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data2),
    )
);

$context2  = stream_context_create($options2);
$result2 = file_get_contents($url, false, $context2);
var_dump($result2);

?>
```

Listing C.15: Retrieve all information of two users from the database based on nickname

```
<?php
$url = 'http://127.0.0.1/get_ip.php';
$data = array('NICK'=>'donald1dsg');
$options = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data),
    )
);

$context  = stream_context_create($options);
$result = file_get_contents($url, false, $context);
var_dump($result);

$data2 = array('NICK'=>'Tester');
$options2 = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data2),
    )
);
```

```php
$context2  = stream_context_create($options2);
$result2 = file_get_contents($url, false, $context2);
var_dump($result2);

?>
```

Listing C.16: Retrieve all information of two users from the database based on IP address

```php
<?php
$url = 'http://127.0.0.1/get_user.php';
$data = array('IP'=>'10.0.0.2');
$options = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data),
    )
);

$context  = stream_context_create($options);
$result = file_get_contents($url, false, $context);
var_dump($result);
?>
```

Listing C.17: Test that the server allocates a new IP address to users and that it returns the correct information

```php
<?php
$url = 'http://127.0.0.1/update_ip.php';
$data = array('NICK'=>'donald1dsg','IP'=>'10.0.0.2');
$options = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data),
    )
);

$context  = stream_context_create($options);
$result = file_get_contents($url, false, $context);
var_dump($result);

$data2 = array('NICK'=>'Tester','IP'=>'192.168.0.1');
$options2 = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data2),
    )
);
$data3 = array('NICK'=>'Tester','IP'=>'192.168.0.2');
$options3 = array(
        'http' => array(
        'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
        'method'  => 'POST',
        'content' => http_build_query($data3),
    )
);

$context3  = stream_context_create($options3);
$result3 = file_get_contents($url, false, $context3);
var_dump($result3);

?>
```

Appendix D

Android Source Code Overview

In order to provide some overview of the complexity of the application, we have used an metric plugin which supports Eclipse [33]. By using this plugin we are able to get useful information such as *total line of Java codes*. It should be mentioned that the total amount of source code which is represented in this appendix is partially from the *Manet Manager*. Even though most of this code has been edited, modified or completely re-coded, we believe it should be mentioned. This has also been stated several times in Chapter 3 and 4.

In Table D.1 we can see some general information about the application. In Section D.1 we have listed every Java class with its respected code length. Finally in Section D.2 we have listed every XML class for the UI with its respected code length.

Total line of Java codes	8846
Total line of XML codes	1992
Number of Java classes	106
Number of XML classes	27
Number of attributes	400
Number of static attributes	276
Number of methods	553
Number of packages	13
Number of static methods	765
Method lines of codes	5266

Table D.1: Android code data analysis.

D.1 Java Project Structure

Package name	***Class name***	***Lines of code***
uia.d2d.install	FinalStage	169
uia.d2d.install	InstallFragment_1	34
uia.d2d.install	InstallFragment_2	52
uia.d2d.install	LoadRegister	215
uia.d2d.install	MainInstaller	65
uia.d2d.localdb	DbAdapter	217
uia.d2d.manager	AlertDialogClass	92
uia.d2d.manager	ApproveCall	66
uia.d2d.manager	AutoReplyText	47
uia.d2d.manager	Calling	66
uia.d2d.manager	ChangeSettingsActitivy	442
uia.d2d.manager	CommunicationListener	108
uia.d2d.manager	CommunicationMaster	139
uia.d2d.manager	CommunicationService	55
uia.d2d.manager	CustomSimpleCursorAdapter	59
uia.d2d.manager	EditIgnoreListActitivy	130
uia.d2d.manager	Identifiers	56
uia.d2d.manager	MainActivity	651
uia.d2d.manager	ManetManagerApp	132
uia.d2d.manager	MessageService	128
uia.d2d.manager	NavDrawerItem	42
uia.d2d.manager	NavDrawerListAdapter	50
uia.d2d.manager	PopupDialog	53
uia.d2d.manager	Protocol	62
uia.d2d.manager	ReceiveMMSservice	88
uia.d2d.manager	ReceiveVideoService	79
uia.d2d.manager	RtpService	128

uia.d2d.manager	SendMessageActivity	220
uia.d2d.manager	SMS	397
uia.d2d.manager	Validation	152
uia.d2d.manager	Video	363
uia.d2d.manager	ViewLogActivity	127
uia.d2d.manager	ViewLogActivityHelper	29
uia.d2d.manager	ViewMessageActivity	97
uia.d2d.manager	ViewRoutingInfoActivity	129
uia.d2d.manager	ViewVideoActivity	94
uia.d2d.manager.adapter	LruBitmapCache	29
uia.d2d.manager.adapter	PlacesListAdapter	47
uia.d2d.manager.adapter	PopupDialog	51
uia.d2d.requests	AddUser	69
uia.d2d.requests	GetNick	65
uia.d2d.requests	ServerReply	30
uia.d2d.requests	UpdateIP	71
uia.d2d.service	CircularStringBuffer	26
uia.d2d.service	LogObserver	4
uia.d2d.service	ManetHelper	220
uia.d2d.service	ManetObserver	16
uia.d2d.service	ManetParser	51
uia.d2d.service.core	ManetBootBroadcastReceiver	17
uia.d2d.service.core	ManetCustomBroadcastReceiver	18
uia.d2d.service.core	ManetService	133
uia.d2d.service.core	ManetServiceHelper	615
uia.d2d.service.routing	Dijkstra	80
uia.d2d.service.routing	Edge	34
uia.d2d.service.routing	HelloMessage	19
uia.d2d.service.routing	Node	50
uia.d2d.service.routing	OlsrProtocol	216
uia.d2d.service.routing	Route	27

uia.d2d.service.routing	RoutingProtocol	13
uia.d2d.service.routing	SimpleProactiveProtocol	489
uia.d2d.service.routing	SimpleReactiveProtocol	38
uia.d2d.service.system	CoreTask	446
uia.d2d.service.system	DeviceConfig	155
uia.d2d.service.system	DnsmasqConfig	28
uia.d2d.service.system	HostapdConfig	36
uia.d2d.service.system	ManetConfig	462
uia.d2d.service.system	ManetConfigHelper	80
uia.d2d.service.system	RoutingIgnoreListConfig	23
uia.d2d.service.system	TiWlanConf	39
uia.d2d.service.system	WpaSupplicant	54
uia.d2d.service.legal	EulaHelper	54
uia.d2d.service.legal	EulaObserve	4
uia.d2d.helpers	onAddNewUser	4

Table D.2: Android Java class structure.

D.2 XML Project Structure

Class name	***Lines of code***
aboutview	99
callanswerui	85
callui	95
clientrow	65
contacts_adapter	147
contacts_adapter2	73
drawer_listview_item	33
finalstage	97
ignoreviewwrapper	29

loadregister	43
logview	38
main_installfragment	14
main	66
navigation_drawer_info	53
nonetfilterview	34
norootview	25
routinginfoview	41
sendmessageview	61
settingsview	120
settingsviewrapper	29
sms	168
smsview	99
step0	67
step1	72
toolbar	10
video	203
videoview	126

Table D.3: Android XML class structure.

Appendix E

Android Bugs and Suggested Solutions

This appendix has as purpose to provide detailed guidelines of known bugs that should be looked at or fixed in further work.

- When *onPeersUpdated* is triggering in *MainActivity*, it has tendence to double and duplicate users into the database.
  Suggested Solution: The datbabase could define that the IP field should be unique to avoid duplication.

- Tilting the phones/tablets in some UI classes results in layout floating out of bound. The last part of *Setup* has this problem.
  Suggested Solution: The application needs to be styled and aligned correcly in the *XML* files so it supports to be tilted, and renders correctly to differenct screen sizes.

- The initialization setup does in some cases tend to do odd operations. In the end of the setup where the application enters the *MainActivity* and reloads the OLSR service in order to have reference to the *MainActivity* it can bounce back to the setup, which it should not do.

Suggested Solution: This issue is currently unknown why it actually does bounce back. One solution could perhaps be to force-close the activity that is referrenced to the setup. This need to be done in the beginning of the *MainActivity* class.

- When the application sends request to register and obtain an valid IP from the server, the applicaton can in rare cases not manage to either send or obatin the response from the server. This could either be that the application gets an *HttpException* or the server is currently not in the system.
  Suggested Solution: This bug actually has to some extend a solution, where the application gets closed if this step fail. Still, this bug should be handled in a better way. For example, an re-attempt could be done, and/or pause the *setup* until it actually manages to register and obtain valid IP from the server.

- If the system get more than 255 registered users it can result in IP-collisions. Currently there is none method for Time-To-Live (TTL) on the IP-adresses.
  Suggested Solution: The clients and server can deploy TTL on the IP-adresses.

- The voice establishment does curretly not have any busy tone and mechanism for using speaker-phones.
  Suggested Solution: By modifying the voice establishment from the *Communication* class, it should be possible to listen for any input if the desired target is busy or have declined the call.

- The application can freeze if one tries to transmit large files. This occur for both picture and video services.
  Suggested Solution: The system currently relies on receiving data on custom port to the socket, and reading the bytes when the packets are decoded. Most likely the issue occur when stacking the bytes, where it normally uses the RAM for caching before saving the file. If the file is too large, we suspect that the Android's native *garbage collector* removes some of the retrieved data, resulting in weird behaviour on the application on both transmitter and recipient.

- The application does curretly not have any indication if the router has announced HNA and has connection to the Internet.

Suggested Solution: This could be checked/verified by using a green color that indicate if HNA is announced and the router has Internet connection.