FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Self-organizing mesh network of android devices

Rui Archer

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Cristina Costa Aguiar (Ph. D.)

July 2013

Self-organizing mesh network of android devices

Rui Archer

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: José Manuel Magalhães Cruz (Ph. D.)

External Examiner: Susana Isabel Barreto de Miranda Sargento (Ph. D.)

Supervisor: Ana Cristina Costa Aguiar (Ph. D.)

23th July 2013

Abstract

This thesis focuses on bringing ad-hoc mode networking to the users of Android devices in a simple way. On the devices that support being put in ad-hoc mode, doing so involves a series of complex steps and the manual configuration of the network.

This remains a complicated affair because Android does not provide native support and most solutions remain a hack. By combining those steps into an application, which allows the user to switch to ad-hoc mode with the push of a button we hope to make this easier.

To demonstrate the potential and challenges faced when developing applications capable of operation in ad-hoc mode, we developed two proof-of-concept applications.

The AndroidMesh application we developed lets users manage their transition to and from ad-hoc mode without losing their previous network configurations; this makes it simples to switch between modes.

The Ad-Hoc communicator application allows for the exchange of messages and files, but its usefulness is not limited to android devices. It can be used to interconnect with other IP enabled devices due to the use of standard protocols like IP and serverless XMPP. The application is instantly compatible with existing applications like the Pidgin instant messenger allowing communications with other users in any computer, not just in other phones running this same app.

The use of the Zero-configuration networking methodology and a software stack based on the IETF standards for auto configuration and service discovery makes our applications interoperable with many other implementations available today for other platforms.

Empowering users with an easy to use ad-hoc mode is a great way to make their networking experiences more flexible, but we think it is possible to go beyond that by making network architectures and applications tolerant to intermittent network connectivity and enable the exploitations of multiple methods of data transfer.

Resumo

Esta dissertação foca-se em tornar possível a utilização do modo *ad-hoc* em dipositivos Android de forma fácil e simples para o utilizador. Nos dispositivos que suportam o modo *ad-hoc*, mudar para esse modo envolve uma sequência de passos complicados e a configuração manual da rede.

A complexidade tem persistido porque o sistema Android não fornece uma forma nativa de o fazer levando a que a maioria das soluções não seja mais do que um remendo. Ao combinar os passos complexos numa aplicação que permita ao utilizador mudar para modo *ad-hoc* com o premir de um botão esperamos tornar mais fácil este processo.

Para demonstrar o potencial e os desafios encontrados quando se desenvolvem aplicações capazes de operar em modo *ad-hoc*, foram desenvolvidas duas aplicações prova de conceito.

A aplicação AndroidMesh que desenvolvemos permite ao utilizador gerir a mudança de e para o modo *ad-hoc* sem perder as definições das redes que possuía anteriormente, isto torna fácil e simples alternar entre modos.

A aplicação Ad-Hoc Communicator permite a troca de mensagens e ficheiros, mas a sua utilidade não está limitada aos dispositivos Android. Podendo ser também utilizada para interagir com qualquer outros dispositivos capazes de operar numa rede IP dado ao uso de protocolos padrão como o IP e o serverless XMPP. A aplicação é de imediato compatível com aplicações existentes, tais como a aplicação de mensagens instantâneas Pidgin, permitindo ao utilizador comunicar com outros utilizadores que estejam a usar um computador e não só outros utilizadores desta mesma aplicação em dispositivos Android.

O uso da metodologia *Zero-configuration networking* e de camadas aplicacionais baseadas em padrões da IETF para autoconfiguração da rede e descoberta de serviços faz com que as aplicações sejam compatíveis com várias outras implementações existentes para outras plataformas.

Dotar os utilizadores de uma forma fácil de usar o modo *ad-hoc* é uma boa forma de flexibilizar as suas experiências com redes, mas é possível ir além disso tornando aplicações e a própria arquitetura da rede tolerantes a conectividade intermitente e permitir que possam ser usadas múltiplas formas de transmitir os dados.

Acknowledgments

Here are some people to whom thanks are in order.

Ana Aguiar my supervisor for the helpful input and guidance.

Bruno Randolf for his work in bringing ad-hoc mode to the masses in CyanogenMod.

The XDA-Developers community for their continued work to create roms for devices long past their prime, in particular thanks to those who worked in the roms for the galaxy550, galaxy510 and ZTE blade.

The Cyanogen Mod developers for bringing us even more freedom and nice features to a great mobile OS.

Those who share their knowledge in the stackoverflow community.

The people who create great libraries that fill the functionality holes left by Google, namely Jake Wharton for actionbarsherlock, Paul Burke for aFileChooser and Stephen Erickson for RootTools.

Finally, I would like to thank my family and all those who have supported me one way or another, in special Ermelinda for bringing me joy and keeping me company despite being almost 2000 kilometers away.

Rui Archer

Contents

Lists of Figures

Abbreviations

ARP	Address Resolution Protocol
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Service
IANA	Internet Assigned Numbers Authority
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
MANET	Mobile Ad-hoc Networks
OLSR	Optimized Link State Routing Protocol (Protocol)
OS	Operating System
PDA	Personal Digital Assistant
PSN	Pocket Switched Network
ROM	Read-Only Memory
SDK	Software Development Kit
USB	Universal Serial Bus
WLAN	Wireless Local Area Network
WPA	Wi-Fi Protected Access

Chapter 1

Introduction

1.1 Context

The case for Mobile Ad-hoc Networks (MANET) revolves around the ability to establish a wireless network to connect devices without the need of previously deploying network infrastructure, thus requiring minimal upfront costs and making this a fast and low complexity process that can happen in any location.

A MANET is composed of several mobile devices such as smart-phones, personal digital assistants (PDAs), laptops or other mobile devices capable of wireless communication, where the devices form a multi-hop network that is self-organizing and self-configuring. The main differentiator from common wireless mesh networks is the mobility of the nodes that make the network.

The nodes need to cooperate to setup the network and adjust to the dynamic changes a MANET suffers, such as intermittent connectivity, change in the network path between any two node and network mergers. There is a need to provide network services usually handled by centralized infrastructure such as name assignment and address allocation, service discovery among others.

1.2 Motivation and Objectives

Much of ongoing research in the area of MANETs is oriented towards military and disaster recovery scenarios [1], [2], [3] what has somewhat affected the appearance of MANETs solutions for regular users. The work that goes into those military scenarios complies with specific requirements that might not be fully necessary in civilian applications.

As such, MANETs have not seen much popularity outside of military and disaster recovery applications and academic research.

Not saying that research into civilian commercial applications isn't happening, in fact there is a growing interest in that area in big part due to the proliferation of user devices that implement the IEEE 802.11 standard for wireless local area network (WLAN) computer communication and to the standardization efforts of the Internet Engineering Task Force (IETF) MANET Working Group.

This type of networks has yet to have an impact on the way we use wireless networks. Lindgren and Hui 2009 [3] point out the fact that the lack of a killer application that makes MANETs and other ad-hoc networks useful outside of the niche scenarios mentioned before is the reason why there are skeptics that question the usefulness of this research.

While this thesis does not search for the killer application, it will take into consideration the characteristics identified in their paper as belonging to a killer application, namely user penetration.

Also the point made by Bruno et al. 2005 [2] that "users are interested in general-purpose applications" and that "To turn mobile ad hoc networks into a commodity, we should move to more pragmatic opportunistic ad hoc networking" lays the fundamental argument for making MANETs more accessible to end users and making the case for its use in our daily lives.

Taking in this motivation, this thesis will focus on integrating MANETs in regular user's networking experience.

For that we are basing our work in the foundation laid at Instituto de Telecomunicações (IT) Porto where researchers have enabled the capability of mobile phones to build a mesh network, this was achieved through configuration and modifications of the Android operating system.

The Android operating system provides the best foundation for our work. The open nature of the operating system, makes it easy to modify and extend its capabilities, its market penetration also enables us to achieve the goal of user reach.

Our effort extends the work that has already started at IT Porto to meet the goals of a pervasive ad hoc networking experience where any user can easily use MANET features to complement their networking experience by making it easier to set up the mesh network with reduced amount of configuration and start an application on top of it.

The work required to achieve our goals takes several steps:

- Adding auto configuration
    - Implementation of an distributed address allocation mechanism
    - Implementation of an network service discovery mechanism
- Integrating the MANET feature into the core OS experience
    - Changing from infrastructure to ad-hoc networking should be seamless and easy to achieve
- Developing an example application

- Create application covering the use case where users want to establish personal communication and exchange files with each other in the absence or without resorting to the existing network infrastructure.

The application to be developed is an Instant Messaging application that resorts to distributed messaging protocols, this application enables each user to view which other users are available to communicate and any other information they have chosen to share.

As many of the service discovery and messaging protocols are designed with single hop local area networks in mind, we will look into how to enable operation on a multi-hop MANET.

1.3 Report Structure

In Chapter 2, we look into the state of the art research that is happening in the area of MANETs, ad-hoc networks and into improving networking in general.

In Chapter 3, we move on to how to enable ad-hoc networking on the Android Operating System (OS), we start by analyzing the limitations of the operating system in this regard and their role in preventing easy transition between infrastructure and ad-hoc modes and the added complexity this brings to both developers and users. With the Android OS limitations established we lay down the approach we undertook to overcome those limitations and make it possible to enable ad-hoc mode, simplify transition between networking modes and enable easy development and deployment of applications designed to operate in ad-hoc mode.

In Chapter 4, we describe implementation details of the solutions developed, starting with the application that enables easy switching between ad-hoc and infrastructure modes and then describe the instant messaging application that allows users to exchange messages and files in ad-hoc mode.

In Chapter 5, we present the results obtained through our work, we show how the applications operate and achieve their intended purposes.

Finally in Chapter 6, we present the conclusions, our opinion on the current state of MANET offerings to users and developers, what we expect in the future from research community and the industry and point out what we think needs to be done to improve the penetration MANETs and their usefulness into the daily networking experiences of mobile users.

Chapter 2

Literature Review

2.1 Mobile Ad-Hoc Networks

To better understand where we are on the road towards enabling ad-hoc networking experiences, we looked into what are the state of the art approaches to creating infrastructure less networks, we also looked into where the industry, researchers and users think mobile networks are going and where demand for ad-hoc networks emerges.

2.1.1 Existing Ad-Hoc Research

One of the focuses of research being conducted in the field of ad-hoc networks is in making networking more flexible, both on the physical network side and on the application side, as with most applications nowadays being developed with infrastructure networks and always on connectivity in mind, they are not prepared for operation in ad-hoc networks.

Sathiaseelan and Crowcroft (2012) [4] present a report on a workshop that brought academia, industry and regulators to discuss the challenges on bringing mobile internet to everyone. The workshop focused on how to bring better performance and coverage by looking at all layers of the network stack.

Most recent advances have brought us better speeds (3G, 4G LTE, and LTE Advanced) but have not done much to improve coverage, which is not as far reaching as mobile operators would lead us to believe. Further increases in speed can be achieved by bonding network resources together. This can be done at the link level by linking several 3G channels, provided by different mobile operators to provide expanded bandwidth, for example. But also new mobile applications have appeared that allow bonding of resources with greater flexibility at the application level one

of such applications is Open Garden [5] which allows several users to create a mobile overlay mesh network to share their internet connections.

Creating an application capable of operating in a mesh network requires architectures that are less infrastructure centric and connection centric as in these kinds of networks connectivity can be spotty and sporadic due to user mobility. Regarding this issue, a number of researchers have proposed mechanisms to allow applications to adapt to changes in attachment points, and switching between connection technologies, extensions and modifications to the TCP protocol that support connection migration like Multi-Homed TCP [6], or Multipath TCP [7], [8]. Those modifications devise the establishing of several paths between two endpoints across different technologies providing redundancy and reliability.

Other researcher provide propose greater changes to the way connectivity is considered and have explored solutions using Pocket Switched Networking (PSN) [9] where communication takes advantage of brief connection opportunities.

Scott et al. (2006) [10] have propose Haggle, a networking architecture designed around mobile users. This proposed architecture exploits three types of connectivity: ad-hoc, infrastructure and mobility. A discussion about the problems caused by the TCP/IP status quo is presented as well, starting with IP centric applications requiring synchronous connections; this prevents an application from simply sending data and closing the connection without having to worry if there is and actual connection to the destination or when it will be available. As is, an application needs to keep itself running and check itself for network resources availability what is not very efficient and wastes power on devices that are already limited in that regard.

They also put forward a new set of networking principles, like taking advantage of brief connection opportunities, exploiting all data transfer methods, enabling asynchronous network operations to reduce application complexity and forward data using application layer information instead of using IP addresses to identify endpoints.

This approach ruptures with the status quo and the authors themselves recognize that the problems and questions raised are as many as the answers provided. But as many challenges are faced in PSN we think that the effort could bring forward ways of networking far more flexible than what we have today and maybe an hybrid approach combing the TCP/IP suite with PSN could be achieved for enabling greater networking flexibility.

Other aspect of research we find in the literature are cases where advances are being made in architectures that rely on the existence of ad-hoc networks, revealing a need for the higher availability of the capability.

The architecture for wireless ad hoc multiplayer games presented by Huang et al. (2011) [11] assumes that each player is equipped with a mobile device with wireless communication capabilities that can communicate locally with nearby nodes without networking infrastructure.

Many others are solving performance problems in ad-hoc networks. All this research reveals a need to have greater availability of ad-hoc networking in consumer devices and the ability to create great experiences.

2.1.2 IETF Standards

The IETF groups has been working on the definition of standards that allow the interoperability of systems operating in MANETs. In this section, we present the standards that are widely used in existing implementations.

2.1.2.1 Auto Configuration

To achieve a self-configuring network, some work is needed to enable automatic, decentralized IP address attribution.

To achieve this, one of the most established standards is the one described in RFC3927 [19] which "describes how a host may automatically configure an interface with an IPv4 address within the 169.254/16 prefix that is valid for communication with other devices connected to the same physical (or logical) link."

The IPv4 prefix 169.254/16 is registered with the Internet Assigned Numbers Authority (IANA) for the specific purpose of achieving link local communication. To select an address in this space, each device uses a pseudo-random number generator with a uniform distribution in the desired range, from 169.254.1.0 to 169.254.254.255.

For the seed of that generator, some piece of static device information such as the IEEE 802 MAC address can be used so that the number generated is always the same for each device even if the device has no persistent storage.

To ensure that a device does not begin to operate with an address that is already in use and cause disruption to the device using that address, it must first detect conflicts. This is done using Address Resolution Protocol (ARP) probes, and in the case of a conflict, both hosts are supposed to renumber.

The conflict detection and address claiming algorithm is defined in the standard in the following way:

Before probing wait random time between zero and one seconds, then send three probe packets randomly spaced between one and two seconds apart.

If during the probing period that starts when the first probe is sent to two seconds after the last probe is sent, the host receives any ARP packet on the interface where the packet's sender IP address is the address being probed, the host must treat this address as being in use, and must select another address.

Another possible scenario is one where, while probing, the host receives an ARP probe where the target IP address is the address being probed but the sender hardware address is not the address of the host's interface, in this case the host must also consider the address in use as in this case another host has also generated the same IP address and is trying to claim it.

It is also defined that the host must keep track of the number of address conflicts it has experienced and if that number exceeds ten it must limit the rate at which it probes for new

addresses to a limit of sixty seconds. This is done to prevent APR storms in limit cases such as when a rogue host answers to all the ARP probes it is being sent, preventing the other hosts from selecting an address.

If after two seconds after the last probe is sent the host has not received an ARP reply or probe for the address being probed, it has successfully claimed the desired address.

After claiming the address, the host must announce that it is now using it; this is done so that other hosts in the network can update their ARP cache entries in case they have stale entries for the same address associated with a previous host.

This is performed by sending two announcements spaced two seconds apart.

After successfully claiming the IP address, the device can start to use it.

Before creating a network, the devices need to have a routing protocol in place to deliver packets to the destination. Of the existing routing protocol proposals for ad-hoc networks, the Optimized Link State Routing (OLSR) protocol [20] provides the best support for the Android platform as it is easy to compile a new Android compatible binary from source.

After the device has self-configured and the OLSR daemon is started, and we are left with a self-configure network of Android devices.

2.1.2.2 Network Service Discovery

For successful interaction between the mesh participants, each device must know what services the other devices support and in which ports they are operating.

In a network with infrastructure, a Domain Name System (DNS) server is usually used to resolve names and services to hosts and port numbers. In a mesh network, a decentralized solution must be used.

Two extensions to the DNS protocol can be used for that purpose, they are multicastDNS [21] (mDNS) and DNS Service Discovery [22] (DNS-SD). mDNS extends the Domain Name Service system to operate over link-local multicast in a distributed manner. DNS-SD defines how to discover network services over DNS.

Both mDNS and DNS-SD are flexible and generic enough to be used for any service discoverability purposes on a link-local network.

Multicast DNS provides with the ability to perform DNS queries in a link-local network. In this model of operations, each node contains a DNS database with its own service records (Figure 1) and those announced by others. Each time a node registers a service, a broadcast is made to let the other nodes in the network know, then the nodes can perform DNS lookup queries to obtain more details about the announced service and store it in the local database.

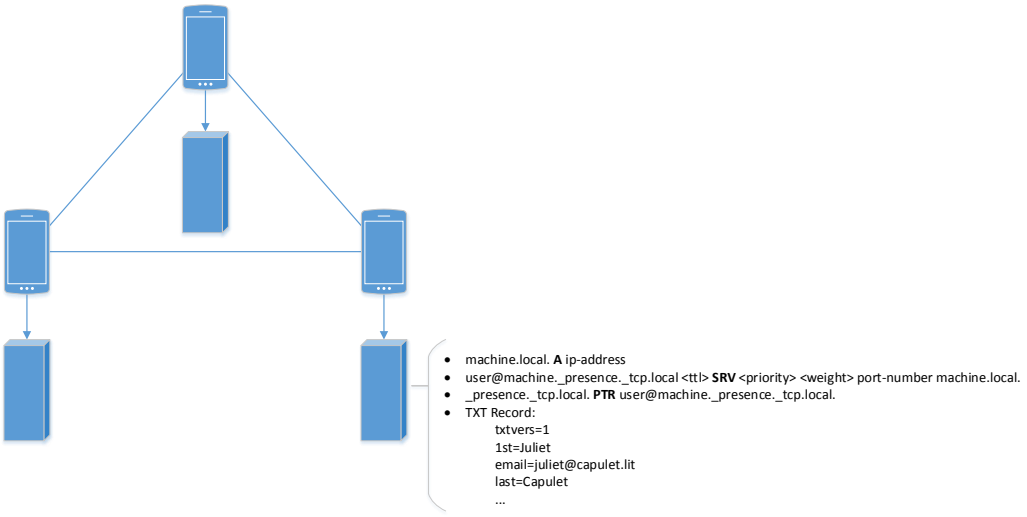

Figure 1 - Multicast DNS

DNS-SD specifies how DNS records are named and structured to facilitate service discovery as seen on Figure 1. An address record, a service record a pointer record and a TXT record are the four DNS records required for service announcement, discovery and resolution.

2.2 Existing Ad-Hoc Applications

2.2.1 Alljoyn

Alljoyn [12] is an open source software system that provides an environment for distributed applications running across different device classes with an emphasis on mobility, security, and dynamic configuration, developed by Qualcomm's Innovation Center.

Alljoyn's aim is to provide a universal framework and set of services to enable different systems to interoperate independently of how they are connected, be it Wi-Fi, Ethernet, Bluetooth, etc. It provides service level discovery, broadcasting of capabilities, connectivity, security and management of ad-hoc networks.

The interoperability is boosted by both the independence from the technology being used for connectivity and the independence of the underlying operation system. It provides software development kits for Android, iOS, Windows RT, Windows 8, and 7. Support for Linux including embedded Linux distributions is also available.

Their work targets not only scenarios where users using different devices can interact, but also where smart devices can exchange information among themselves in an internet of things where every device is connected.

The provided framework has great flexibility, but the solution implemented does not use existing standards, instead implementing their own logic. Service discovery for example is implemented around a distributed software bus that can only be used by other applications that also use the Alljoyn framework. This makes the framework usefulness be very dependent on its success and adoption by developers.

2.2.2 Open Garden

Open Garden [5] allows users to share an internet connection between devices. Its goal is to solve the problems of coverage and throughput in a seamless way.

This solution comes in the form of an application that creates a mesh network and allows the devices that make part of that network to share their internet connection with each other. It is capable of multi-hop routing when there is no direct internet access allowing devices that do not have an internet connection to use the internet connection of other devices in the network and of channel bonding for increased throughput and automatic path choice to select the best performing internet access when in the presence of several alternatives. It also allows several internet connections to be bonded together for greater speeds.

Open Garden goes a long way to solve the coverage and throughput problems, but the implementations encounters some problems on the Android OS. The way it is implemented makes the connections provided by the application not appear as “first class” networks, leading Android to not recognize the network and think that there is no connectivity, this causes applications that test for connectivity to receive a “no connection available” response from the OS when they could in fact connect. This issue is recognized by Google but a fix is not in the near horizon, it can be tracked on Google’s android issue tracker as issue 33666 [13]. Still the problem is limited to applications that test for connectivity in a technology dependent way, only recognizing Wi-Fi or 3G connections.

2.2.3 Cuckoo

Cuckoo is presented [14] as a decentralized microblogging application created as an alternative to the centralized services, mitigating the problems such an architecture carries, such as having a single point of failure and performance bottlenecks, this decentralized architecture also helps save on bandwidth costs and creates a more robust and scalable service.

Pastry [15] is used as the underlying P2P network overlay for the dissemination of information among the network nodes while minimizing used bandwidth and storage consumption and providing self-organization, and adaptation to changes in network conditions such as node failures and network partitions.

Cuckoo's design rationale and architecture looks to overcome the issues of centralized services, but does not exclude dedicated servers from services providers by being capable of using them as a back up to guarantee availability

The presented design shows a solution where an ad-hoc network can be used to offset costs from centralized service providers and increase the robustness of the service and increasing scalability.

2.2.4 The Serval Project

The goal of the Serval Project [16] is to bring communications services to everyone, not just the ones who live in places where those services are offered by corporations or even the ones who can afford those services.

To do this the project steps in to provide service where providers do not, whether for economical or technical reasons. As mobile phones stop being able to communicate when cellular infrastructure fails or where it does not reach, the Serval Mesh is an Android application [17] that allows phones to form impromptu networks of phones allowing nearby people to keep communicating. This allows for communication in the event of natural disasters that destroy the cellular network or in cases where the infrastructure becomes overloaded with too many users. Letting phones communicate directly between them permits communication where cellular networks are not available, which despite bolstered coverage numbers provided by operators falls short in certain areas.

The Serval Mesh also provides privacy and is built on a foundation engineered to support security and privacy through encryption. With privacy, or the lack of it, being a recurring theme in the modern world, this is a much-desired property.

The project goals, principles and roadmap provide one of the most complete solutions for using ad-hoc network to improve people's lives and bring connectivity for all while providing a way to overcome the limitations of the offerings provided by using the phone as infrastructure.

2.2.5 Samsung Chord

Samsung Chord [18] is similar to Alljoyn in that they both have the same objectives of enabling decentralized networking with easy service discovery, where Chord differs is that it does not encompass ad-hoc network connection.

It offers an SDK providing a set of services and application interfaces that allow for interaction between devices, broadcast of messages and data transfer to nearby devices and multiplayer games. The focus is on decentralized services without the need for dedicated servers, the way it operates revolves around the concept of channels, a way to partition the network in groups of nodes that are interacting with each other. These channels can be private or public, with

the public channel being used for administrative functions and to discover all of the nodes in the network that are using Chord and the private channels for actual interaction.

For each channel, it is possible to obtain a list of nodes and send and receive messages and files, these interactions can be 1-to-1, 1-to-many or many-to-many.

Samsung has just recently made Chord available and its features are limited, requiring the developers to do much of the heavy lifting, but with Samsung's commitment to improving it, this can get better. Still they face the same challenges of Qualcomm in getting developer traction to adopt the technology.

2.3 Summary

Several efforts are in development to address many of the requirements to achieve decentralized ad-hoc networking of devices, aspects like coverage and throughput are the focus of Open Garden, Alljoyn makes ad-hoc networking possible across different technologies and enables service discovery, as does Chord.

The Serval project shows how it is possible to enable phone based ad-hoc networks and enable communication on top of them, improving the lives of many and having in place the technology provide connectivity in situations of emergency.

Finally, Cuckoo provides a compelling example of the benefits of adopting these technologies that appeal to the decision makers at the telecommunication companies, services providers and the remaining stakeholders. The lack of penetration of these technologies is mostly due to a lack of incentives or user demand for its incorporation in devices and operating systems.

The current implementations of ad-hoc networking experiences targeting Android devices rely on new and/or proprietary standards, ignoring the existing standards used across many existing implementations. This limits their usefulness unless they gain mass-market adoption.

Chapter 3

Enabling Ad-Hoc Networking in Android

Just like the Nintendo DSi and Sony PSP, support ad-hoc networking to enable local multiplayer games, android devices could benefit of the same feature. Not only for gaming but also for transferring a file to a friend who is right next to you in a situation where there is no Wi-Fi infrastructure and other useful applications that can emerge once support is in place.

3.1 Android Limitations

The Android OS is composed of several layers atop of a Linux kernel (see Figure 2).

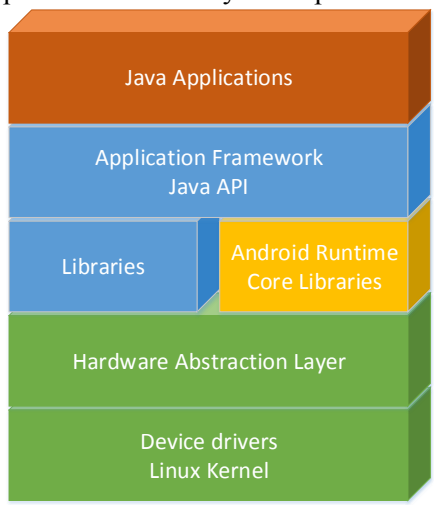

Figure 2 - The Android OS architecture

Unlike a Linux kernel targeting a desktop PC, in the kernel used for the Android OS, the device drivers are statically linked into the kernel at compile time, to save storage space.

This is one of the constraints that developers must be aware when developing for mobile devices along with limited memory and processing capability. Although at the current pace of evolution observed in the mobile device, technologies those limitations are becoming less important.

That makes changing or patching drivers a laborious task as it requires a new compilation of the whole Android OS.

Despite having a robust network management system, the Android OS has no logic in place to support ad-hoc mode. This lack of ad-hoc support comes down to two factors, lack of driver support and lack of both logic and user interface to enable this mode.

Driver support varies between chipset manufacturers and even between driver versions. In our experience with the Broadcom chipset inside the Samsung Google Nexus S, the BCM4329, in versions of the Android OS before 4.0 the driver in use was the bcm4329 driver, after 4.0, the OS started shipping with a new version now named bcmhd, result of an effort by Broadcom to unify their drivers for the various chipsets. Under the bcm4329 driver, the phone could be put in ad-hoc mode, but with bcmhd this was no longer possible. Thanks to the efforts of the development community, specifically Bruno Randolf [23] a patch that restored ad-hoc support was developed.

Other devices we have tested had Wi-Fi chipsets developed by Atheros and where using the ath6kl driver, which supported ad-hoc mode.

Due to the lack of logic to handle the ad-hoc mode, enabling ad-hoc by creating a custom network configuration in the appropriate configuration file is not enough to achieve network connectivity, as the OS cannot configure an IP address due to there not being a DHCP server present in the network. This forces the user to manually configure the network via the system user interface, what is an annoyance and requires technical expertise.

Furthermore, as in an ad-hoc network we have a distributed architecture, with no central nodes with special roles, discovering the other nodes in the network and whatever services they offer is not possible. This forces each developer to implement their own solutions for service discovery, potentially leading to incompatibilities, and additional complexity.

3.2 Enabling Simplified Ad-Hoc Networking on Android OS

After identifying the limitations of Android OS in enabling ad-hoc networking, we need to overcome those.

To do so and enable simplified ad-hoc networking we need to enable and simplify ad-hoc network creation and configuration, enable automatic IP address configuration, enable multi-hop packet forwarding and enable node and service discovery.

Each topic will be handled in detail in Sections 3.2.1 to 3.2.4.

3.2.1 Creating Ad-Hoc Networks

As Android OS does not provide a way for the user to enable ad-hoc mode and create a network, the only way to achieve this is to perform a determined sequence of step to put the device into ad-hoc mode.

Before those steps are taken, the device needs to satisfy two pre-requisites: the first is that the driver for the Wi-Fi chipset supports being put into ad-hoc mode, the second is that the device allows for root access.

Ad-hoc support at the driver level depends on whether the manufacturer ships the driver with the support enabled of if the source for the driver is available and we can make those changes ourselves.

To get root access we need to perform a process called rooting, this process usually involves exploiting a flaw in the OS code to gain root user permissions. This process varies from device to device and between OS versions and the results obtained after it can range from permanent to temporary. Due to the multitude of way his can be achieved the best way to obtain knowledge on how to perform it is to consult with internet communities such as XDA-Developers [24] to get to know what specific tools or procedures work for the devices at hand.

An alternative, sure proof and the most currently used way to get root access in an Android device is to flash it with a custom compilation of the Android OS (an Android ROM as the community calls them) that already offers root access. There are many custom ROMs offering a wide set of enhancements over the stock version of the OS and most of them include root access and the su utility to regulate access to root user permissions by applications.

This second method was the one we used and we will now describe how to perform it.

Before performing the step mentioned below one needs a computer to which to connect the device to for manipulation, this computer must have installed either the complete suite of Android development tools or at least the android debug bridge (adb). On windows systems, a driver specific to the device is also needed to allow for computer to device communication.

To flash a new ROM we first changed the stock recovery with a custom one that allowed for flashing of custom ROMs. The recovery is a system that resides in a separate partition on the device and that after booting into the special recovery mode allows the user to reset the device to factory settings, clean user data and update the system. The boot loader code responsible for preparing the system pre-boot process also resides in the recovery partition, this code includes, in

most of stock recoveries, code to check the ROM on the device and blocks the booting of unauthorized versions.

As such to flash a custom recovery first we will need to put the device in the download mode, this is one of the special modes an Android device usually supports. In this mode, the phone boots into a state that allows the device's recovery partition to be written by a computer over an USB connection in a process known as downloading. The steps to put the device usually involve having the device turned off and then pressing a combination of physical buttons on the device, in some device this is achieved by pressing the power button while pressing another button such as the volume up or volume down at the same time. Again, an internet search can yield the specific button combination needed to a specific device.

To upload a new recovery to the phone, we need a special piece of software that is most of the times specific to the device at hand and created by the hacker community, again we recommend an internet search to find the appropriate one. For example to download a new recovery into Samsung devices the tools to use is the Odin Multi Downloader, with different versions supporting different devices. Some of these tools require additional files specific to the device into which we are downloading the recovery.

Having all the pieces into place, we also need to get a version of a custom recovery compatible with the device. During our research we used the ClockWorkMod [25] recovery that has been modified to support a wide range of devices by the community.

Then we take the following steps:

1. Put the device into download mode
2. Launch the downloader software on the computer
3. Connect the device to the computer via USB
4. Use the downloader interface to download the recovery into the device

Note:

Putting the device into download mode can also be achieved by introduction the following command in the terminal while the device is powered up and connected to the computer via USB:

```
adb reboot download
```

When the download process ends, the tool might send a command to the device to reboot, if this does not happen we need to perform a long press of the power button to do this.

After this, we can install the custom ROM, after acquiring the file containing the ROM, which usually comes as a .zip file we need to perform these steps:

1. Put the file in the memory card of the phone, either a physical card or an internal partition reserved by the manufacturer for this purpose
2. Reboot the phone in recovery mode
3. Use the install update from sdcard option and select the update file

4. After the update finishes use the recovery option to restart the device

After rebooting, the device will boot into a brand new OS installation with root access enabled.

Despite requiring more steps than the previous method, which usually only requires downloading and application that enables root at the press of a button, this method ensures root access and allows for more flexibility.

All these steps to gain root access are needed because enabling ad-hoc mode in and android device involves modifying system configuration files, which are not accessible to the regular user.

Enabling ad-hoc mode is done by writing a network configuration entry in the WPA supplicant configuration file that defines an ad-hoc network and then restarting the supplicant.

The WPA configuration file's location and name can vary between devices some of the know locations and names are:

- `/data/misc/wifi/wpa_supplicant.conf`
- `/data/wifi/bcm_supp.conf`
- `/data/misc/wifi/wpa.conf`

A template for the configuration file is the following:

```
update_config=1
ctrl_interface=wlan0
eapol_version=1
ap_scan=2
fast_reauth=1
network={
  ssid="Ad-Hoc"
  scan_ssid=1
  key_mgmt=NONE
  mode=1
}
```

The key options in the configuration are ap_scan=2 which can force the network to be created regardless of scan results and mode=1 which defines the IEEE 802.11 operation mode and sets it to IBSS (ad-hoc, peer-to-peer) mode.

Another detail that needs attention is the name of the network control interface define by the ctrl_interface=wlan0 option. In Android, the Wi-Fi interface is a virtual device, which only exists when Wi-Fi is turned on and its name can vary. The best way to obtain this parameter is to read the previous configuration existent in the WPA supplicant configuration file and retrieve it.

It is also recommended to back up the entire contents of this configuration file before writing our configuration and restore the configuration file to its previous state when disabling ad-hoc mode.

After modifying the WPA configuration file, we must issue two commands using the wpa_cli utility:

- wpa_cli reconfigure - to make the WPA supplicant daemon re-read the configuration file
- wpa_cli terminate - to terminate the supplicant, which will cause the Android network management system to restart it.

At this point, the device will have created an ad-hoc network with the name “Ad-Hoc”, but we still need to configure the network settings to achieve connectivity.

By default, the network settings configured by the Android network management stack default to using DHCP to automatically configure the network, but in an ad-hoc setting a DHCP server is not present, as such we must reconfigure the settings for this scenario.

The way to achieve this depends on the version of the Android OS as before version 3.0 network settings like IP address, netmask, gateway and DNS servers where defined globally for all the networks and changes had to be made to these settings whenever we changed from network to network.

With versions 3.0 and 4.0+ the Android OS introduced per network settings allowing for configuration of separate networks with different settings for each allowing for ease of transition between networks that used either DHCP or ones that required and static IP to be configured.

To configure the network interface we are going to use the Android versions of the Linux network configurations tools. These tools do not come by default with the system and must be installed separately. The best way to do this is to install a version of BusyBox that was compiled for the arm architecture of most Android devices. BusyBox combines small stripped down versions of many common UNIX utilities into a single executable binary, it has been written with size and limited resources of mobile device in mind.

BusyBox can be installed in a rooted Android device via applications available on the Google Play Store, two examples of applications that install BusyBox are BusyBox [26] and BusyBox Installer [27].

Now we demonstrate how the network configurations are performed.

For Android OS versions before 3.0, the following commands can be used to configure the network device with the pretended network settings.

First, we configure the interface with the IP address and netmask:

- `busybox ifconfig <interface name> <IP address> netmask <netmask>`

Then we configure the default gateway, this is not relevant for ad-hoc mode operation, but must be configured:

- `busybox route add default gw <gateway IP address> dev <interface name>`

Finally, we use Android's setprop command to set the DNS servers, again not relevant but must be set:

- `setprop dhcp.<interface name>.dns1 <DNS server IP address>`

After the configuration, we restart the WPA supplicant and the device will connect to the newly configured network.

For Android OS versions after 3.0 if a network does not have custom settings and relies on DHCP for configuration, the only configuration present will be its entry in the WPA supplicant configuration file, the rest will be handled by Android's network management system. In the case that a network's configuration has been customized, that configuration will be stored in a file, this file is usually stored in the same location as the WPA supplicant configuration file's location and is named ipconfig.txt.

Unfortunately, despite the .txt extension the contents of this file are not plain text but the serialization of a binary data structure containing the custom configurations for each network.

The data structure in question is a Java List (java.util.List) containing instances of Android's WifiConfiguration class (android.net.wifi.WifiConfiguration).

The list of networks and their configurations can be obtained via a call to the getConfiguredNetworks() method of Android's WifiManager, but there is a caveat, the network configuration obtained as an WifiConfiguration object does not expose the API that allows to change the IP address, gateway or DNS fields, they are private. To overcome this limitation we can use reflection in the java language to manipulate those fields despite them not being accessible to external code.

With that knowledge, we can determine the steps to configure the settings of an ad-hoc network in Android 3.0 and up:

1. Obtain the list of configured networks
2. Search in that list for the ad-hoc network we just created
3. Use reflection to change the network configuration object
    a. Set IP address assignment to "STATIC"

   b. Set the IP address
   c. Set the gateway IP address
   d. Set the DNS server IP address
4. Call the updateNetwork(WifiConfiguration) method of the WifiManger and pass it the configuration object we modified

After the configuration, we restart the WPA supplicant and the device will connect to the newly configured network.

A more detailed overview of this procedure is explained in Section 4.1 where we present the implementation details of our application that automates the whole process of creating and configuring ad-hoc networks.

3.2.2 Automatic IP Address Configuration

Automatically configuring the network requires the definition of an IP address, a gateway IP address and a DNS server IP address. While the last two are not important for an ad-hoc network and probably should be configured to an address not in the network, the IP address must be properly configured in the 169.254/16 range of addresses.

As specified in RFC 3927 [19] the IP address should be picked from the pool of available addresses by using a pseudo-random number generator that ensures a normal distribution so that different hosts do not generate the same sequence of numbers. It is also recommended that if the host has access to persistent information that is reasonably unique to each host, such as the MAC address of its network interface that information should be used as seed to the random number generator.

The Java API in Android offers the java.util.Random that can generate a stream of pseudorandom numbers and allows for the specification of a seed to be used by the generator.

Armed with such a generator, we now have several options in terms of what information to use as a seed. In a blog post on the Android Developers Blog [28] about identifying the devices an application is installed into so as to track application installations, Tim Bray provides some insight about each one of them.

The alternatives proposed are:

- The TelephonyManager.getDeviceId() call which returns the IMEI, MEID or ESN of the phone
- The Wi-Fi or Bluetooth interface's MAC address
- Serial number available via android.os.Build.SERIAL
- ANDROID_ID available via Settings.Secure.ANDROID_ID

While the blog post focuses on meeting the requirements for tracking applications installs, including the cases where one would want to count installs after a factory reset as a different install for the scenario when the user sells his device. Our requirements are different; we need an identifier that is available in all Android devices, not just phones ruling out the device id returned by the telephony manager. The serial number is required for all devices that do not have telephony like most tablets but is not obligatory for phones so we cannot rely on it.

That leaves us with the MAC address and the ANDROID_ID, each with their respective drawbacks. As we are targeting devices with Wi-Fi capability, we do not need to worry about the fact that some Android devices do not have MAC addresses, but the second drawback pointed out in the post was observed by us. As we mentioned early the wireless network interface on many Android devices is a virtual one that only exists when Wi-Fi is turned on, and that causes a query for the interface's MAC address to return empty if Wi-Fi is off.

The drawbacks with using ANDROID_ID as reported by the blog are that it is not reliable in releases of Android before 2.2 and there has been at least an instance of an error that caused a popular handset from a major manufacturer to have the same ANDROID_ID for every device.

We consider that both the devices wireless interface MAC address and ANDROID_ID are good candidates for being used as a seed to the random number generator, to ensure that we can generate an IP address even if the Wi-Fi connection is turned off we are going to choose the ANDROID_ID for this purpose.

With the seed chosen, we now have to generate an IP address in the 169.254.1.0 to 169.254.254.255 range.

Given that ANDROID_ID is a 64-bit quantity represented as a hexadecimal string, we need to convert into a long representation of its value. To do that we first convert it into a java.math.BigInteger by using BigInteger class constructor and passing it the ANDROID_ID and the radix, which in this case is 16 as the string is an hexadecimal representation. We can then invoke the longValue() method from the BigInteger class to obtain the long value representation and use that value to create a new instance of the java.util.Random class.

Having the random number generator seeded we now need to generate the last two bytes of the IP address, so we first generate a value between 1 and 254 inclusive and a second value between 0 and 255 inclusive, thus creating the address.

Having the address and given that in ad-hoc mode there is no central infrastructure to keep track of what IP addresses are in use, so as specified on RFC 3927 [19] we need to probe the network to check if it is already in use via ARP probes and detect and resolve any conflicts.

The IP address claiming protocol used is the one defined in Section 2.1.2.1.

To broadcast an ARP packet we need to have low-level access to the socket interface, which is not available to Android applications via the Java API as it only provides high level TCP or UDP access.

This issue can be overcome in two ways. One way is using the Java Native Interface (JNI) to execute low level code that writes the probes to a raw socket, this solution requires the use of

Android's Native Development Kit (NDK) to compile that code and incorporation into the java applications.

The other way is by using the arping utility, which is included in certain versions of the BusyBox binary and allows sending ARP requests to neighboring hosts.

The arping interface requires some considerations when using it, because as we need to listen to replies the utility cannot exit immediately. In addition, as the arping utility does not allow for specifying the random interval between probes, we cannot use it to send several probes and must use it to send one at a time and space them apart in the application logic.

The arping options relevant to the address-claiming scenario are:

- -c <count> - the number of ARP packets to send, in our case it will be one
- -s <source> - the source address, as we are probing, this address will be all zeros
- -w <deadline> - the time arping waits before exiting, with this option if does not stop after sending the number of packets specified by -c <count>, it wait either for deadline seconds or until the probes are answered. In our case as the probes are spaced by 2 seconds maximum, this option will be set to that value, including for the last probe.
- -I - <interface> - this option specifies the network interface to use for probing
- <destination> - the target address being probed
- -U - unsolicited mode used to update neighbor ARP caches, no replies are expected

As such an example invocation of the arping utility for address claiming will be:

```
arping -c 1 -w 2 –s 0.0.0.0 -I wlan0 169.254.234.80
```

If after invoking the arping command 3 times randomly spaced between 1 to 2 seconds no reply is received, then we send two announcement ARP probes spaced 2 seconds apart by using the following command for each one:

```
arping -U –c 1 –s 169.254.234.90 –I wlan0 169.254.234.90
```

3.2.3 Service Discovery

With the network setup complete and IP connectivity established we now need a way for the different nodes in the network to know the existence of each other and what services each is running.

This problem has been addressed by the DNS Service Discovery [22] (DNS-SD) and Multicast DNS [21] (mDNS) specifications.

DNS-SD specifies how to name and structure DNS records to facilitate serviced discovery, this mechanism allows the client to discover a list of named instances of services he desires, it specifies that a service instance can be described by using the DNS SRV (RFC2782 [29]) and DNS TXT (RFC1035 [30]) records.

The DNS SRV record describes a service with a structure such as <Instance>.<Service>.<Domain> along with the port the service is running.

The DNS TXT record with the same name provides additional information about the service structures as a series of key/value pairs.

The discovery of the list of available instances of a service via a query for DNS PRT (RFC1035 [30]) records with a name in the format <Service>.<Domain>, which returns a set of names which are the DNS SRV/TXT pairs.

Multicast DNS allows clients to perform DNS like operations in the local link in the absence of a conventional unicast DNS server. It specifies how a client queries for DNS records and responds to those queries. In this mode of DNS, each client stores the DNS records published by other nodes in the network. This advertisement of records happens when the service is first registered or in response to a query.

It should be noted that multicast support in Android has been available since version 1.6 but reports from developers indicate that until later versions issues where found with its implementations.

Since Android version 4.1, these two mechanisms have been included as part of the API under the name of Network Service Discovery (NSD), after initial experimentation with those APIs we found that they do not support the retrieval of DNS TXT records what limits the information we can obtain about discovered services.

As such we have use the JmDNS library [31] which provides an implementation of multicast DNS that can be used for service registration and discovery in local area networks. This library provides a pure java implementation of the service discovery mechanisms that runs on most JDK1.6 compatible java virtual machines as is the case of Android's Dalvik-VM.

We found that the library performs well on the Android OS, but the library has an issue with determining the address used by the interface in Android versions above 4.0, thankfully, this can be overcome by using the Android APIs to obtain the address of the network interface and pass it to the constructor of the JmDNS instance.

3.2.4 Multi-Hop Networking

To make ad-hoc mode more useful for various scenarios, we need the ability to create a multi-hop networks where nodes without direct network links can communicate by having their packets routed to the destination by intermediate network nodes.

Of the several ad-hoc routing protocols such as AODV [32] and OLSR [20], OLSR has the best support and an implementation for the Android OS that is already in use by several projects and easily compiled from source to target the Android OS.

To use it we must first place the olsrd binary[1] in the /system/bin folder and any relevant plugins in the /data/local/lib folder.

To start the OLSR daemon we can issue the following command:

/system/bin/olsrd -f <path to configuration file>

Where the path to configuration file is the location to the file where we have written the configuration for the daemon.

An example of a basic configuration is:

```
DebugLevel 0
IpVersion 4
FIBMetric "flat"
ClearScreen yes
AllowNoInt yes
IpcConnect
{
  MaxConnections 0
  Host 127.0.0.1
}
UseHysteresis no
NicChgsPollInt 3.0
TcRedundancy 2
MprCoverage 3
LoadPlugin "/system/lib/ olsrd_bmf.so.1.7.0"
Interface "wlan0"
{
  Ip4Broadcast 255.255.255.255
  Mode "mesh"
}
```

[1] olsrd build instructions can be found under the “Building olsrd for Android” section at http://www.olsr.org/?q=olsr_on_android

Because the OLSR implementation used does not forward multicast packets by default and we need that ability for the service discovery processes we use the Basic Multicast Forwarding Plugin which floods IP-multicast and IP-local-broadcast traffic over an OLSRD network. It uses the Multi-Point Relays (MPRs) as identified by the OLSR protocol to optimize the flooding of multicast and local broadcast packets to all the hosts in the network. To prevent broadcast storms, a history of packets is kept; only packets that have not been seen in the past 3-6 seconds are forwarded.

3.3 Summary

With all the technical knowledge we detailed above we set to create an Android application that would allow the user to switch to ad-hoc mode with just the flick of a switch and configure the network properties in a simple manner, thus bringing ad-hoc mode networking to regular users of the Android OS.

Rooting the device still remains the major hurdle for the end user, but the wealth of information on how to perform this made available by the community makes this process less painful.

Chapter 4

Implementation

For enabling ad-hoc mode networking in Android and run a simple instant messaging application on top of it, we have created two applications. The first, AndroidMesh allows the user to connect to an ad-hoc network with just the press of a button and configure that network. The second Ad-hoc Communicator is an instant messaging application that allows the user to find other users in the network and exchange messages and files with them.

4.1 Design

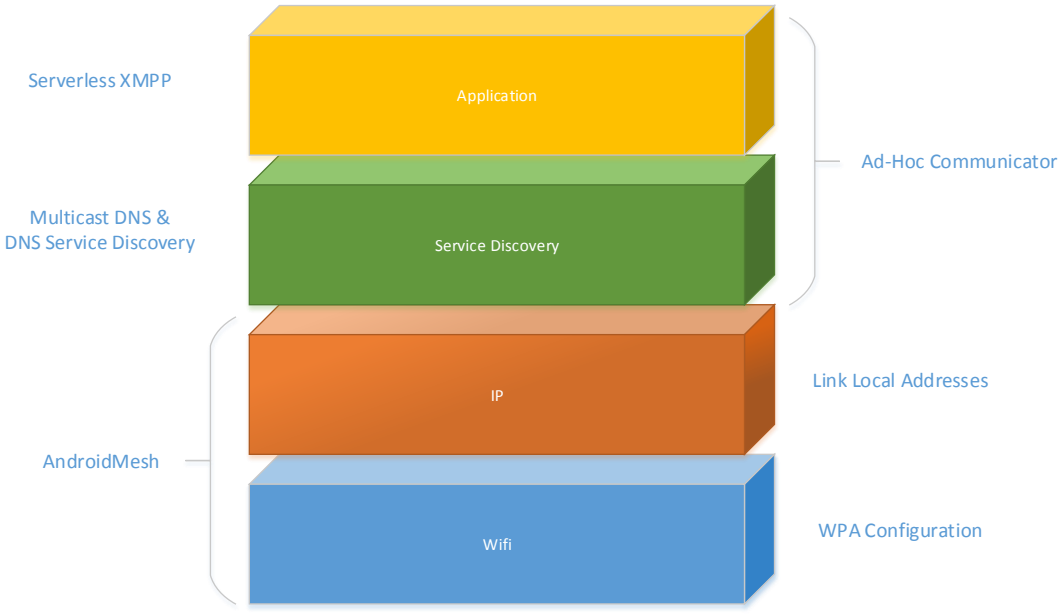

Figure 3 - Network stack interventions

The diagram in Figure 3 shows in which levels of the Android network stack our implementation intervenes and how those interventions are made by each application.

To enable ad-hoc mode, AndroidMesh first modifies the WPA configuration file to enable the creation of an ad-hoc network, then it automatically configures the IP address so that the network is IP enabled.

To allow device to device communication over the ad-hoc network, the Ad-Hoc Communicator application first needs to enable service discovery, for that purpose it uses a Java library that provides a MulticastDNS and DNS-SD implementation. At the application level, it uses the Serverless XMPP distributed messaging protocol.

4.2 Android Mesh

The requirements of the Android Mesh application were that it needed to provide a simple way for the user to switch from infrastructure to ad-hoc mode, while hiding the complex operations that must be performed to attain that objective.

To do this the application provides three views. One where it provides a button and a list of nearby networks. Another where the user can configure settings such as the network name. A third view is provided to show a log of the operations performed by the application as to allow to immediately see if any of the step did not complete without having to connect the device to a computer.

The application resorts to the RootTools library [33] to allow for easy execution of shell commands as the root user, needed to manipulate the system.

On the first launch, the application performs a series of device capability and configurations steps.

First it tries to locate the WPA supplicant configuration file in the list of know locations, if the file is not located, then the application will not proceed.

After finding the WPA supplicant configuration file, the file's contents are read and the ctrl_interface parameter is extracted and stored in the application's global preferences, a persistent key value store where the application can store its settings.

The next step is generating and IP address and storing it in the global preferences as well.

Then it backs up the current WPA supplicant configuration file in the application's local working directory, a directory only accessible to the application, where it does not require root permission to manipulate files.

Finally, the application generates the configuration file for the OLSR daemon, based on the network interface it obtained before.

After these steps, the application has gathered all the information necessary to enable ad-hoc mode networking and the user can perform some additional configurations and use the switch provided in the interface to turn on ad-hoc.

When ad-hoc mode is enabled, the application will write the ad-hoc network entry to the WPA supplicant configuration file followed by the `wpa_cli reconfigure` command.

After this step, the application will enter a state where it periodically checks in the list of currently know networks for the entry added to the WPA supplicant configuration file. By the time the application reaches this step the network management system of the Android OS still is reconfiguring, it first checks if the list of know networks is empty and schedules a retry 2 seconds later.

If the application does not find the entry for the network we just added, it will restart the WPA supplicant and retry 2 seconds later as well.

The flow chart in Figure 4 illustrates the process we use.

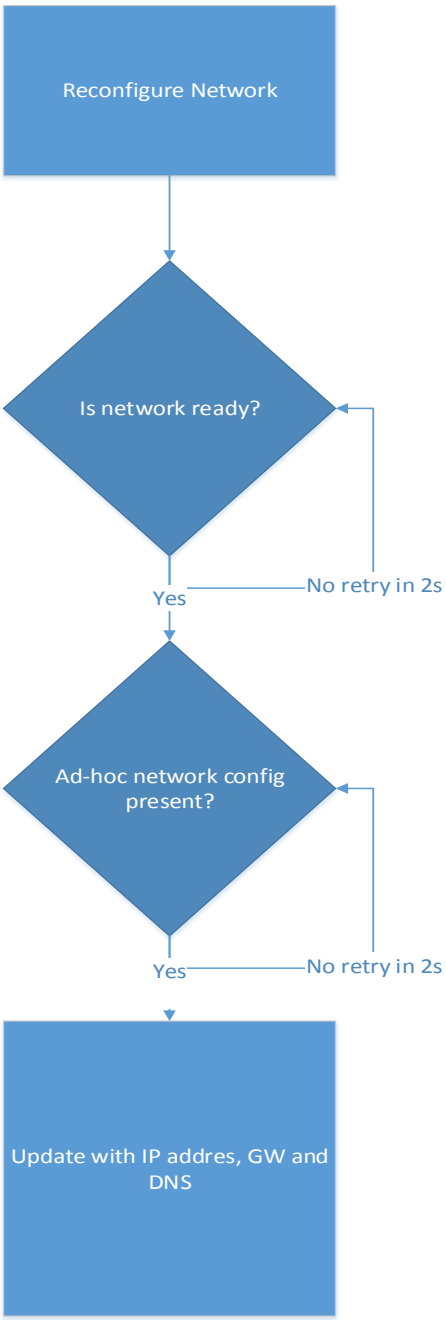

Figure 4 - Network configuration flowchart

As mentioned in Section 3.2.1 the network configuration object does not expose the IP address, gateway and DNS configuration field in the API and as such, we need to use java reflection. For this purpose, we create a class, AdvancedWifiConf that contains methods to manipulate each of the fields in the network configuration object.

That class AdvancedWifiConf has the following public methods:

```
public static void setIpAssignment(String assign , WifiConfiguration wifiConf)
```

```
    public static void setIpAddress(InetAddress addr, int prefixLength,
WifiConfiguration wifiConf)
    public static void setGateway(InetAddress gateway, WifiConfiguration
wifiConf)
    public static void setDNS(InetAddress dns, WifiConfiguration wifiConf)
```

These methods manipulate the WifiConfiguration instance via reflection and change the desired field to the values we need.

The setIpAssignment method changes the enumerate field ipAssignment to the value contained in the parameter assign.

The setIpAddress method changes the member mLinkAddresses which is an ArrayList by first clearing if of any pre configure address and then adds the address passed in the parameter addr to the list.

The setGateway method changes the member mRoutes which is an ArrayList by first clearing if of any pre configure addresses and then adds the address passed in the parameter gateway to the list.

The setDNS method changes the member mDnses which is an ArrayList by first clearing if of any pre configure addresses and then adds the address passed in the parameter dns to the list.

After configuring the network configuration object we use it to update the network settings via the updateNetwork(WifiConfiguration config) method and restart the WPA supplicant, when the Wi-Fi network comes back, the device will connect to the network we just configured and be ready to communicate with other devices in the same network.

With the ad-hoc mode now enabled the application proceeds to start the OLSR daemon to enable a multi-hop network, the application does this by issuing the following command:

```
/system/bin/olsrd –f /data/data/pt.up.it.adhocomms/olsr.conf
```

Where /data/data/pt.up.it.adhocomms/olsr.conf is the path to the OLSR configuration file we generate during the first time setup phase.

After this point, the application stores in its internal settings that the ad-hoc mode is enabled and the user can exit the application if he wishes. Only having to return to the application to turn the ad-hoc mode off and return to the previous configured networks by toggling the switch to the off position.

To return to the previous configuration, the application first removes the ad-hoc network configuration from the list of known networks, then it restores the WPA configuration file that was backed up before and finally invokes the commands wpa_cli reconfigure and wpa_cli terminate to force the WPA supplicant to re-read the configuration file and then to restart.

4.3 Ad-hoc Communicator

The Ad-hoc Communicator application was created to address the use case of several users exchanging messages and files over an ad-hoc network.

The requirements of this application were:

1. Manage the user presence so that the instances of the application running on the other nodes can view each other.
2. List the other users in the network and their presence status (Online, Away or Offline)
3. Exchange messages with other users and keep a history of exchanged messages
4. Exchange files with other users

The application architecture is composed of a service that is always running while the user wants to maintain a presence on the network and be able to be contacted, an application core that keeps application state and views that allow the user to interact with the application and the other users in the network.

The service module is structured as shown in Figure 5.

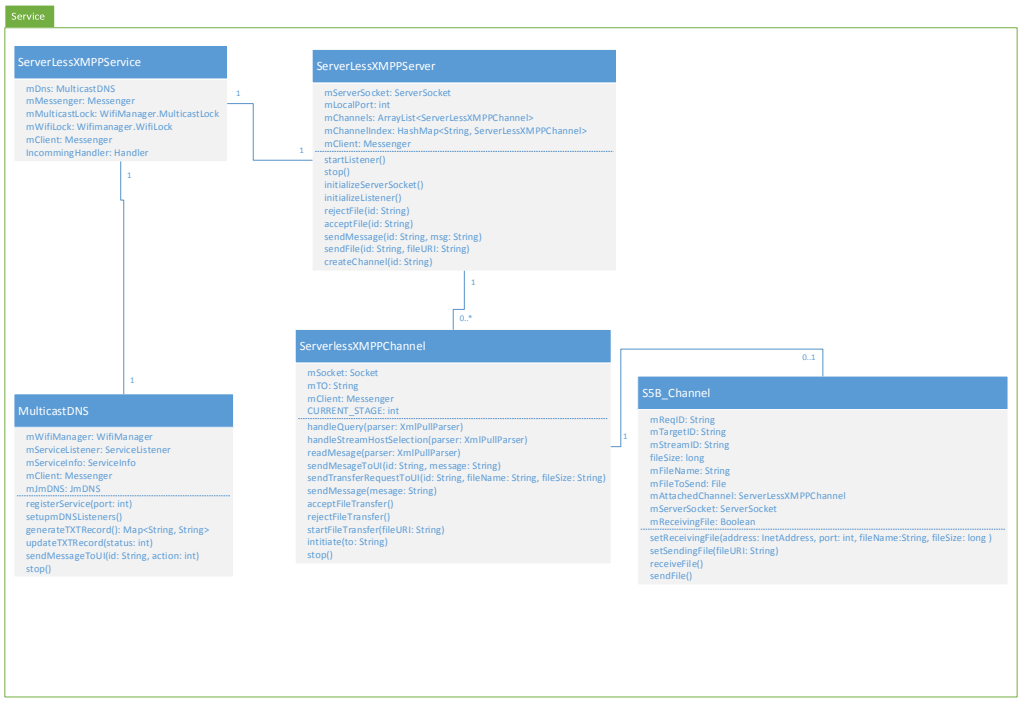

Figure 5 – Ad-Hoc Communicator service class diagram

The following classes compose the service module:

- ServerLessXMPPService, the main service class, it implements Androids service logic and provides communications between the other service classes and the applications module.
- MulticastDNS provides the logic to setup the multicast DNS and DNS-SD service listeners as well as registering the application's own service.
- ServerLessXMPP implements the server logic for the serverless XMPP service, listening to incoming connections to the service's port and starting a new thread running the ServerLessXMPPChannel logic.
- ServerLessXMPPChannel implements the serverless XMPP message exchange protocol, it handles session start and ending as well as messages sent and received
- S5B_Channel implements the SOCK5 Bytestreams protocol for file transfer, it is instanced from the ServerLessXMPPChannel whenever a file transfer is accepted of initiated by the user.

4.3.1 The Service

The service is launched when the application starts and keeps running until the user terminates it via the application interface. To do this the service is started as a sticky service, this is one of the modes of operation for services on the Android OS, this mode is used for services that are started and terminated as needed. A non-sticky service would be terminated as soon as the user navigated away from the application interface, but as we want the application to keep receiving messages even while the user is using other applications we need to use a sticky service for this purpose.

When started the application starts the service, binds to it and creates a messenger object to be able to send and receive messages to and from the service. This messenger object is created from the IBinder object returned when binding with the service.

When the service starts, it first sets up by acquiring the multicast lock that will be needed by the multicast DNS component and creates a Wi-Fi lock so that the Wi-Fi radios are not turned off while the service is running. After this phase, the service will start the ServerLessXMPPServer, which is responsible for listening to incoming connections, and finally will start the multicast DNS service and daemon, which will announce the presence of the device in the network. The service will then enter a state where it will be listening for messages from the application core and views, Figure 6 shows the startup sequence for the service.

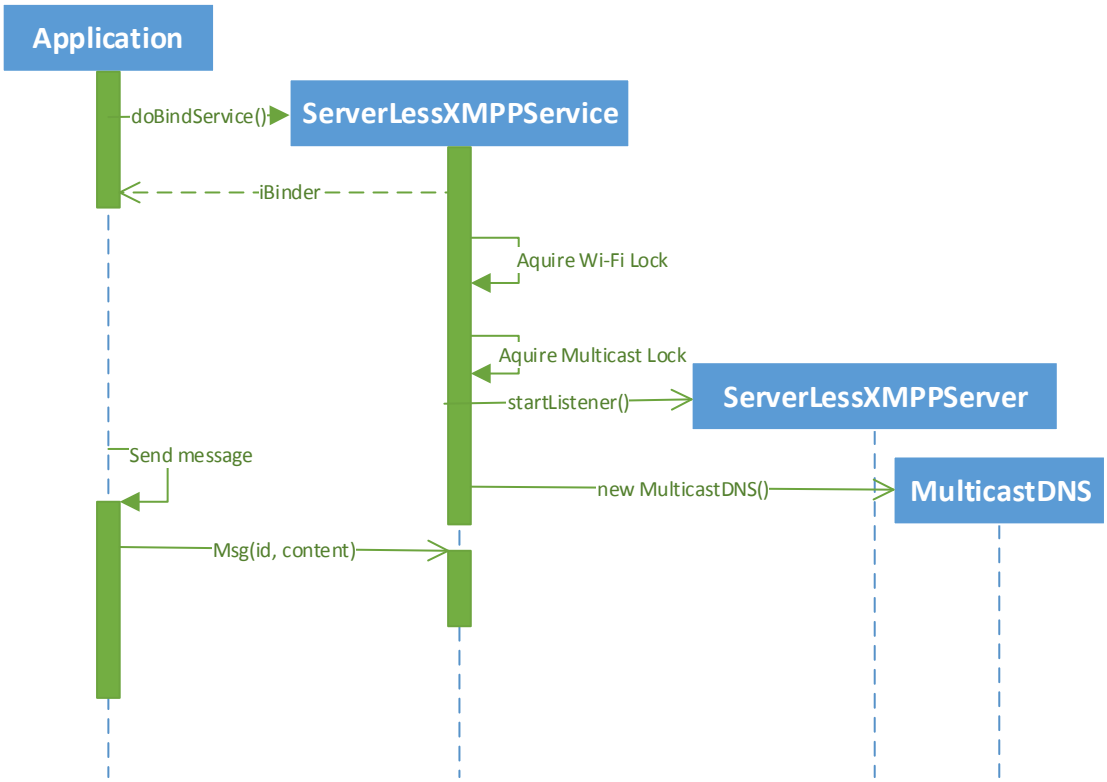

Figure 6 - Application Startup

The ServerLessXMPPServer will open a server socket on the system's first available port to listen to incoming connections for the serverless XMPP protocol, the main thread for this server will then block when the accept() call is made on the server socket. Every time that call returns, it is because another host is opening a socket on the server port and an instance of a socket representing that connection is returned, that instance is then passed to a newly created instance of the ServerLessXMPPChannel class that will spawn a new thread. The sequence diagram in Figure 7 illustrates the ServerLessXMPPServer behavior.

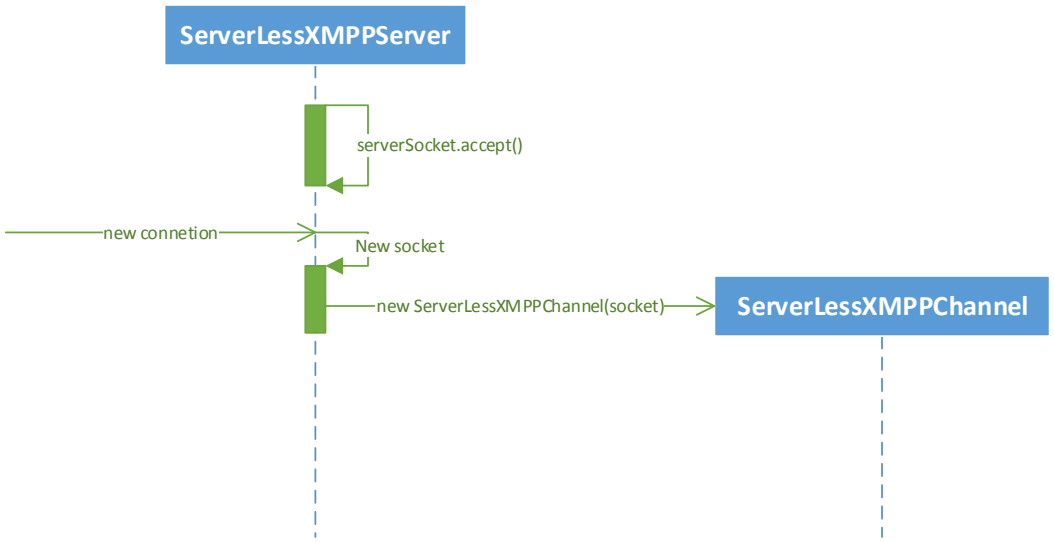

Figure 7 - ServerLessXMPPServer

The MulticasDNS object will start by creating an instance of the JmDNS daemon and then setup the DNS listener callback that will be responsible for listening for new announcements of services in the network. With the listeners in place, it will register this instance's service.

As mentioned in Section 3.2.3 JmDNS cannot retrieve the Wi-FI interface's IP address, so we must pass it in the constructor.

The multicastDNS service listener has three callback functions one for when a service is added serviceAdded(ServiceEvent event), this means the host has received the broadcast from another device advertising a service, another for when the service is resolved serviceResolved(ServiceEvent event), this usually follows the service added callback, unless the service cannot be resolved. Finally, there is the service removed callback serviceRemoved(ServiceEvent event) which is called when the host has received the multicast DNS goodbye announcement from another host. Figure 8 shows the sequence of events that happen when service resolved and service removed events are triggered for the registered ServiceListener.

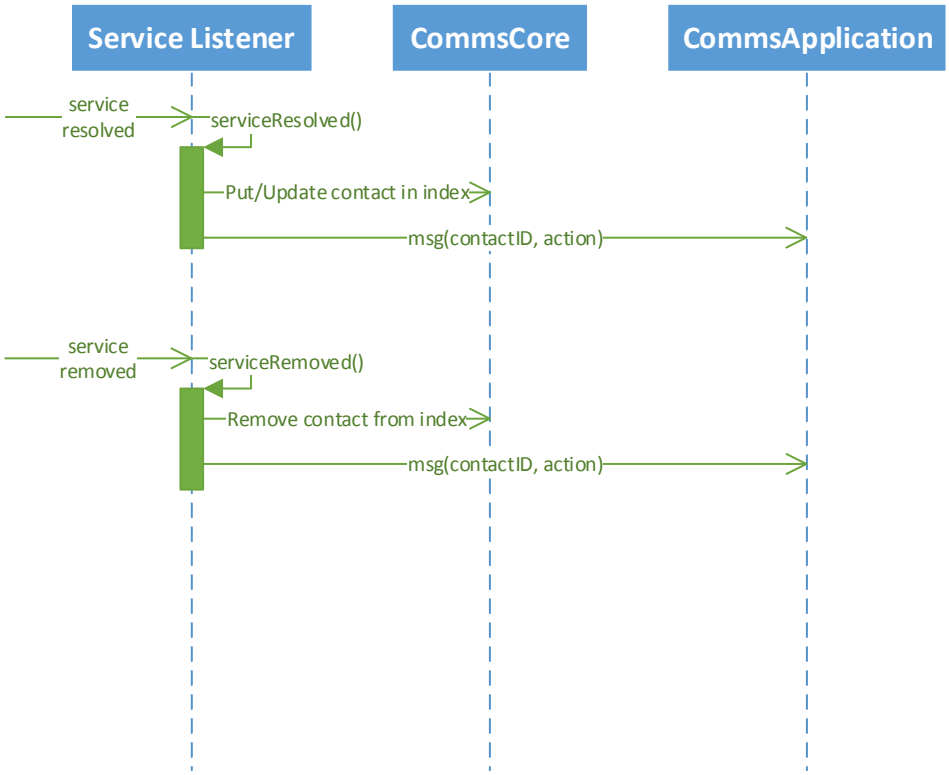

Figure 8 - ServiceListener implementation

When a service announcement is received and the service added callback is invoked, the mDNS daemon still does not have information about the service, so our application only reacts to the service resolution events. When a service is resolved, the listener will check the application's contact index and see if it is already present, if not it creates a new contact instance and adds it to the index followed by a message to the application notifying it that there is a new contact. The application will in turn notify the interface if it is present, if not it will create a system notification. If a service resolved message is received for a contact that already is in the index. That means that there was an update to the service details and the application will update the existing contact.

After the listeners are setup, the MulticastDNS object will register the service for the application. First it will query what port the ServerLessXMPPServer instance is listening to as that will be the service's port, then it will generate a TXT record from the application's preferences, including user defined nickname, name, alias, and other parameters, and add it to the ServiceInfo object used to register the service, finally it registers the service.

4.3.2 The Application

The application module is structured as show in the class diagram in Figure 9.

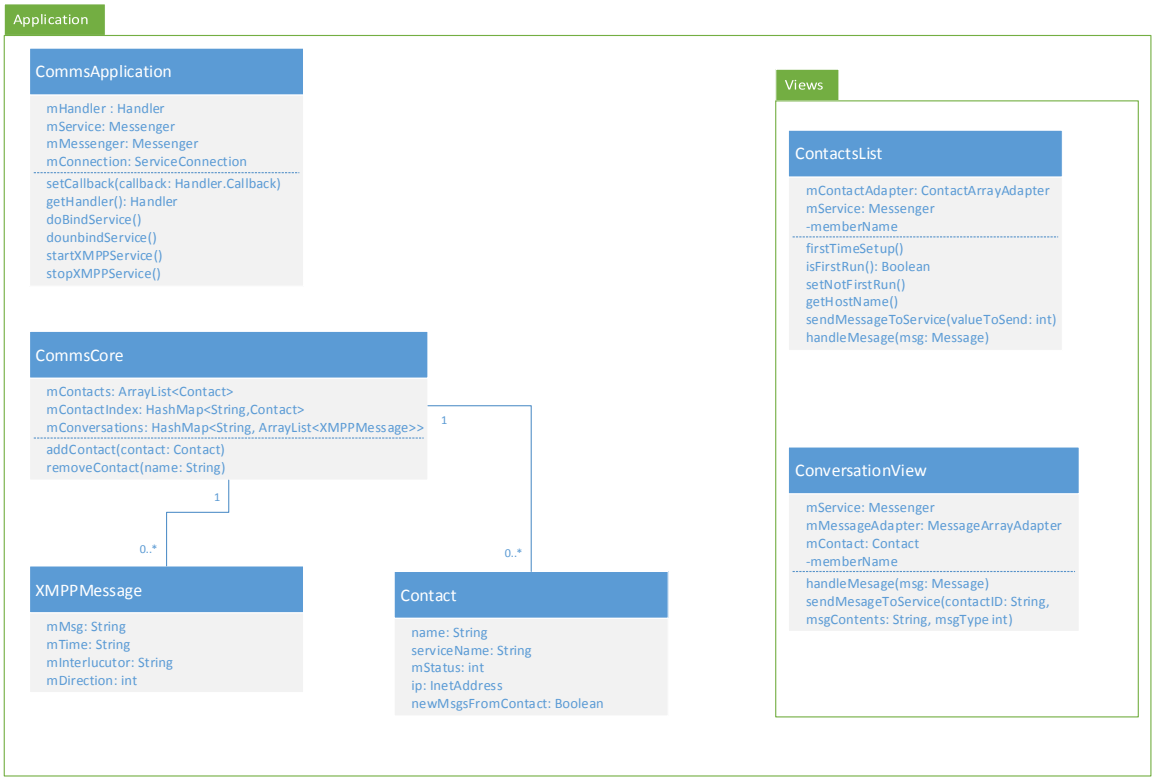

Figure 9 - Ad-Hoc Communicator application class diagram

The following classes compose the application module:

- CommsApplication is a singleton that maintains application state and provides communications between the service and the remaining classes in the application module
- CommsCore is another singleton that maintains the lists of contacts and messages
- Contact is the class that represents a contact
- XMPP message represents a message sent between users
- ContactsList is an Android activity that shows the contact list with the know contacts
- ConversationView is another activity that shows a conversation between the user and a contact

When started, the application starts the service that is responsible for registering the service, setting up service listeners and start the serverless XMPP server.

The first time the application starts, it asks the user to edit the service details, nickname, first and last names, email address and local alias, the application tries to automatically fill the last one with the system hostname but it is not always possible.

Every time the user edits the settings, the application sends a message to the service so that the TXT record is updated with the new information.

The application has two activities besides the edit settings activity, whenever these activities start they register themselves to handle messages from the service so that they can react to updates as well, likewise they unregister when destroyed. By registering to handle messages, the activity receives the messenger object from the application that it can use to send messages directly to the service. (See Figure 10.)

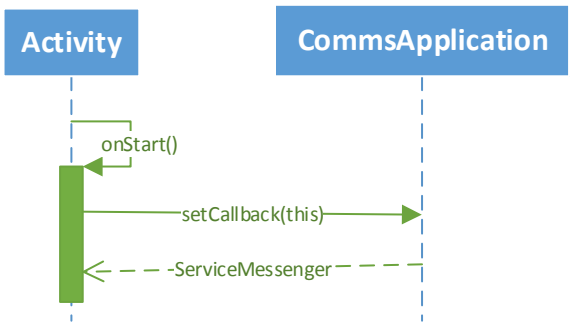

Figure 10- Activity registration

4.3.3 Protocol Implementation

In regular XMPP, presence is handled through special XML tags, in serverless XMPP the presence of an instance is handled via the TXT record field `status` and every change to the presence status requires a change to the TXT record and a re-announcement of the service.

An instance of the ServerLessXMPPChannel class is created every time the serverless XMPP server receives a new connection request and it is run on a new thread, this class is responsible for receiving, sending and processing the XMPP protocol XML tags and for the protocol session logic.

If the chat session is initiated, from an activity, which is the case when the user sends a message to another user and no session is already initiated, the ServerLessXMPPChannel opens a new socket to the remote host. Then it creates an instance of the ServerLessXMPPChannel class, initiates the stream and finally handles the processing to a new thread that runs the ServerLessXMPPChannel logic. Figure 11 shows the sequence of events.

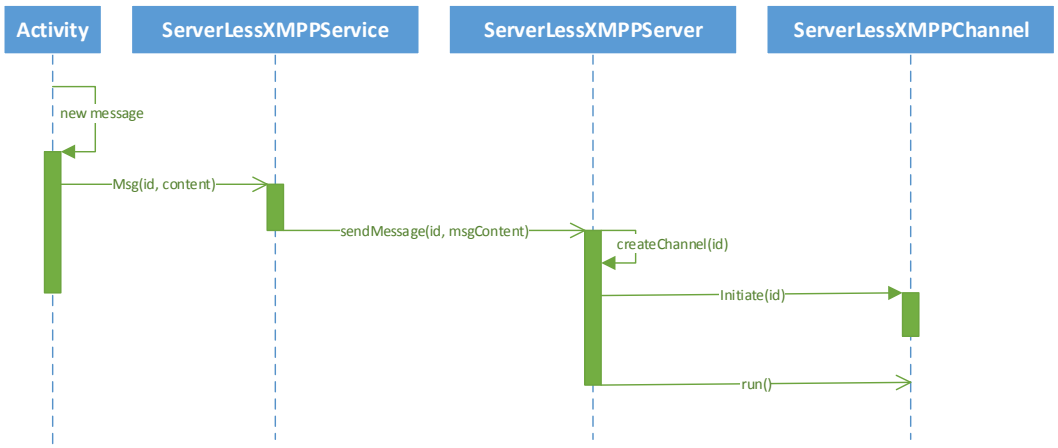

Figure 11- Session start from activity

If the chat session if started by another user, the ServerLessXMPPChannel will process the session start and reply accordingly.

When it receives a message, it sends the message to the CommsApplication class, which will store it in the conversations list in the CommsCore class and notify any activity that has registered with it of the new message. The CommsApplication class is a singleton that is responsible for handling messages directed to it and to the UI and for manipulating the CommsCore singleton that contains the contacts list, index and conversations list.

Each ServerLessXMPPChannel instance starts by handling the stream initiation tag, and them it keeps listening to other tags until the termination tag is received.

While listening it can receive two types of tags, message or query. A message tag is stateless and represents a message received, but a query tag can have different types that need to be handled differently. In our application, we only handle query tags of the “set” type for sending files. When it receives such a tag, the user will be notified of the file transfer request, which will contain the file details such as name and size as well as who it requesting the file transfer. He can then accept or reject the file transfer. If he rejects, a rejection tag will be sent and the instance will return to listening for new tags. If the file transfer is accepted, then it starts the stream host negotiation.

When negotiating stream hosts the sender application will provide with a list of host with the transfer protocols, IP addresses and ports available, the receiver will then select the stream host to use for downloading the file based on what protocols it supports. The specification [34] defines Socks5 Bytestreams and In-Band Bytestream as mandatory and preferred in that order. Our application provides support only for Socks5 Bytestreams.

If no compatible stream host is found, then the file transfer is canceled and the application will return to waiting for more messages.

When a stream host is selected, then the file transfer is started. Figure 12 is a state diagram with the states for the ServerLessXMPPChannel logic for receiving messages and files.

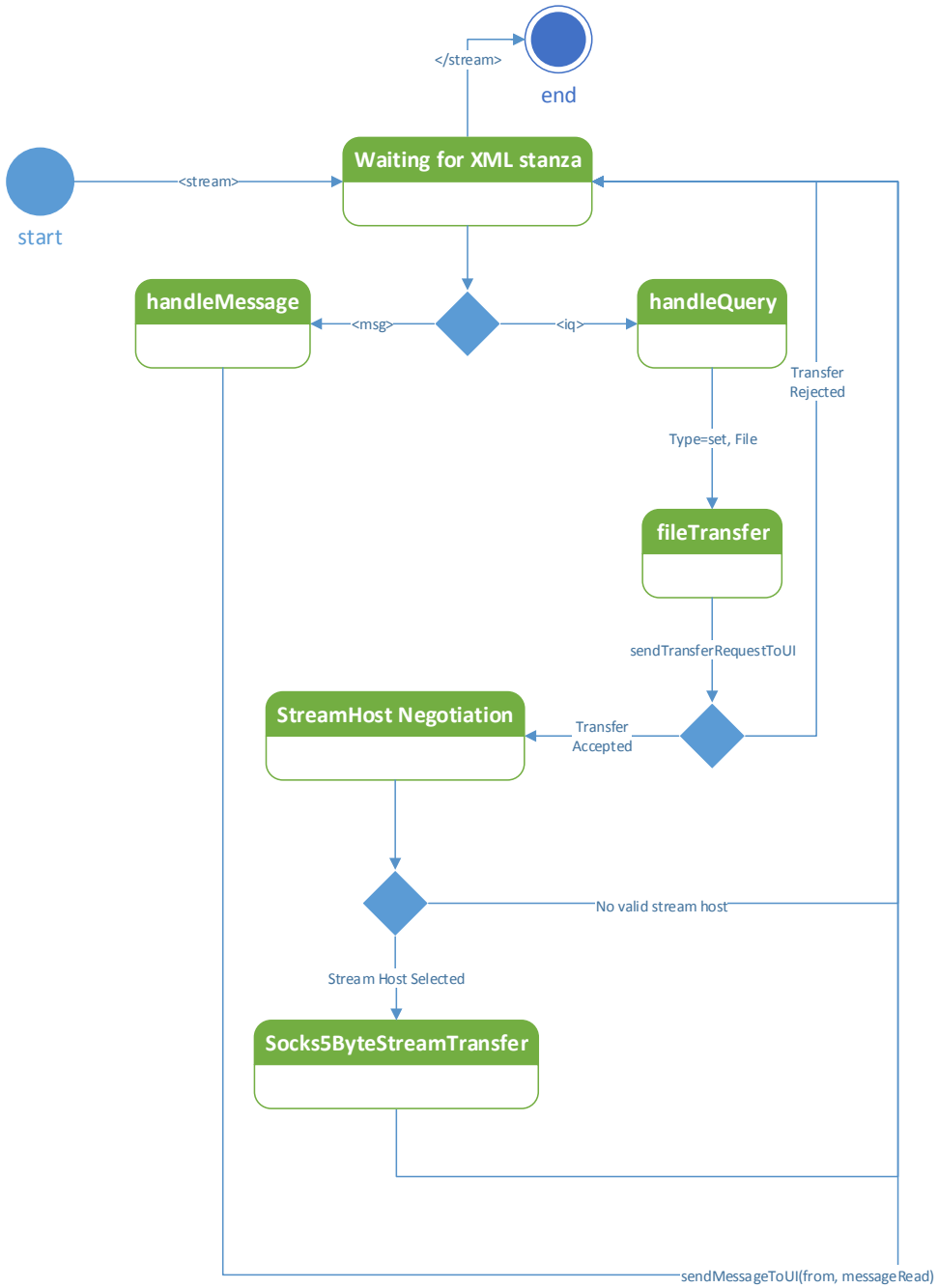

Figure 12- ServerLessXMPPChannel states when receiving messages and files

When a stream host is selected, ServerLessXMPPChannel will create an instance of the S5B_Channel class and run its logic on a different thread. The S5B_Channel implements the Socks5 Bytestream protocol [35].

After creating an instance of that class, the ServerLessXMPPChannel will set whether it is sending of receiving a file, when receiving it will pass the stream host details, when sending it will set the path to the file that is being sent.

The S5B_Channel class will then appropriately send or receive a file, notifying the user through a system notification of the progress of the file transfer.

4.4 Summary

The AndroidMesh application was designed to simplify the steps needed to put and Android device in ad-hoc mode, it is a simple utility application.

The Ad-Hoc Communicator application is a more complex application, it implements the Serverless Messaging extension [36] to the XMPP protocol specification and required some design considerations due to the nature of the application lifecycle Android applications are subject to.

Chapter 5

Results

With the work developed, we have created two applications, the first one, AndroidMesh allows users to enable ad-hoc networking on Android devices running versions above 4.0.

The other application Ad-Hoc Communicator is a distributed instant messaging application that can operate in both infrastructure and ad-hoc mode.

5.1 AndroidMesh

The AndroidMesh application presents an interface what allows the user to turn on ad-hoc mode and lists nearby networks. (See Figure 13).

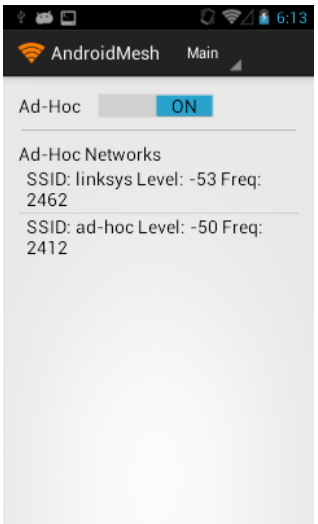

Figure 13- AndroidMesh main interface (at present time the list of networks show all networks, not just ad-hoc networks)

The application provides two views. A main view described above and a view that shows the log of operations, allowing for the quick diagnose of problems without having to connect the device to a computer, see Figure 14.

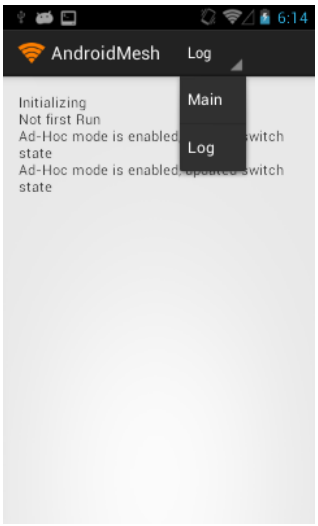

Figure 14- AndroidMesh log view

The application also provides a settings panel so that the user can edit network settings. All parameters that have been auto configured by the application, host IP address, WPA configurations file path, WPA control interface and routing daemon are under the manual configuration header and are not editable by default, but their edition can be enabled for testing purposes. (See Figure 15 and Figure 16)

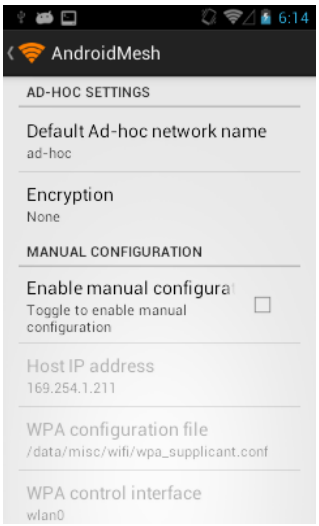

Figure 15- AndroidMesh settings view (1 of 2)

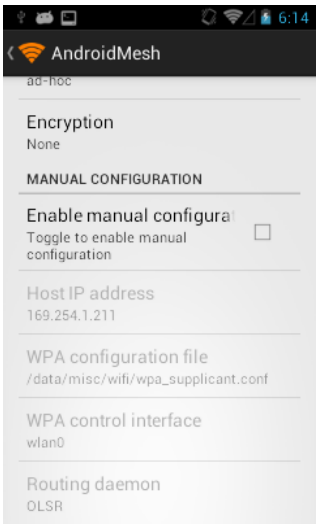

Figure 16 - AndroidMesh settings view (2 of 2)

After enabling ad-hoc mode using the switch in the interface and waiting until the device gains connectivity again, we can see that the network entry created by the AndroidMesh application is now shown in the network manager interface as can be seen in Figure 17 and in Figure 18.

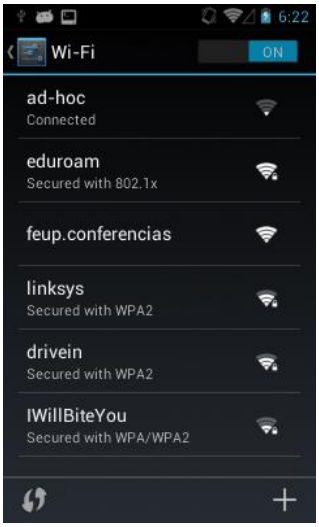

Figure 17- Android network manager, network list

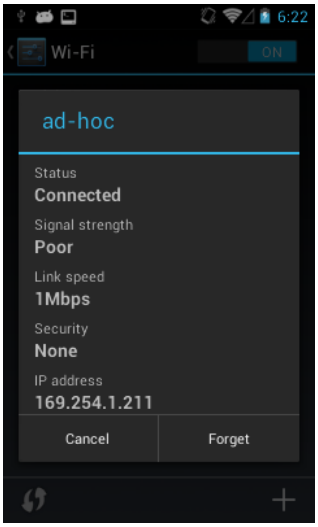

Figure 18- Android network manager, network details

5.2 Ad-Hoc Communicator

After the device is in ad-hoc mode the Ad-Hoc Communicator application can be launched to allow for instant messaging over the ad-hoc network. The application presents the interface see in Figure 19, where it shows the user name, a drop down menu with the user status and a list of nearby contacts. In the image, the first contact is from a Pidgin instance running in a laptop connected to the ad-hoc network, and the second is another android device.

Figure 19- Ad-Hoc Communicator main interface

Using Avahi service discovery browser on the laptop we can see what devices are present in the network and announcing their services. Figure 20 and Figure 21 show the service details for each one of the android phones running an instance of Ad-Hoc communicator.

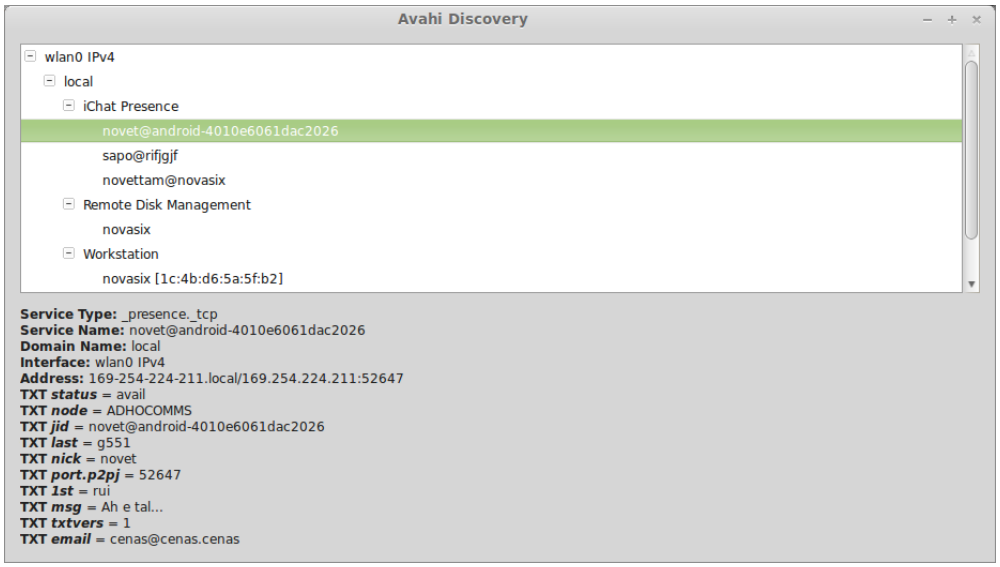

Figure 20- Avahi Presence (1 of 2)

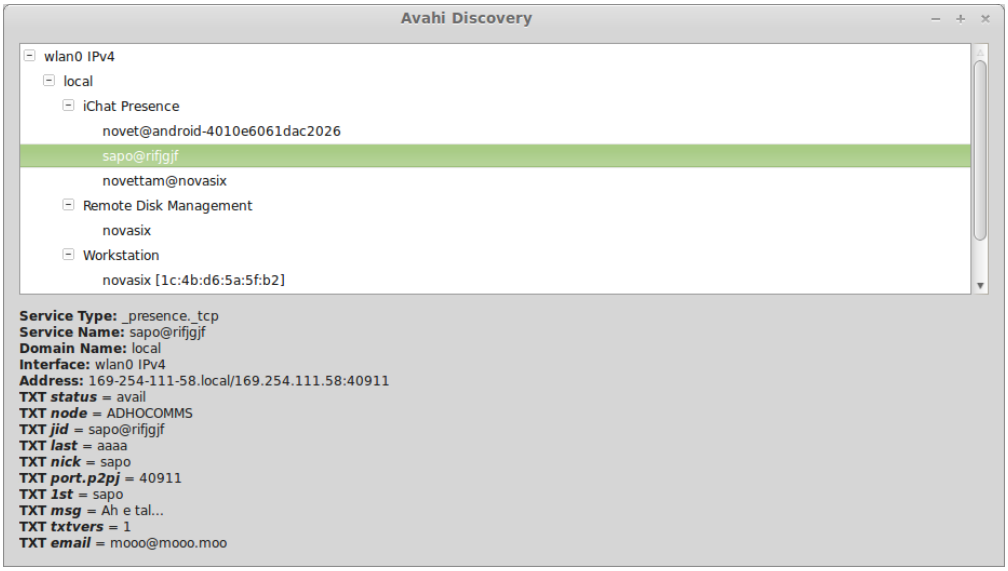

Figure 21- Avahi Presence (2 of 2)

Pidgin as well displays both devices on the Buddy list, acknowledging the service announcements made by the devices as shown in Figure 22.

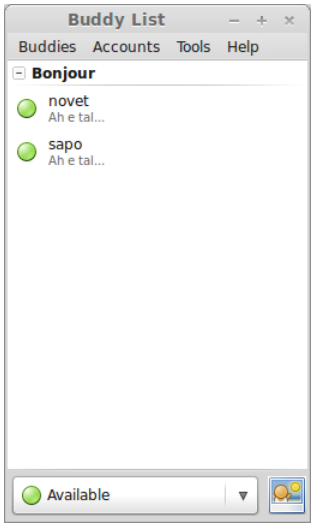

Figure 22- Pidgin buddy list

Update of the presence status via update of the TXT record also works as expected with Pidgin acknowledging the change in availability; see Figure 23 and Figure 24. Figure 19 shows the interface of one of the devices in the state after both devices are away as shown in Figure 24, with the other device appearing as away and the user in the laptop showing as online.

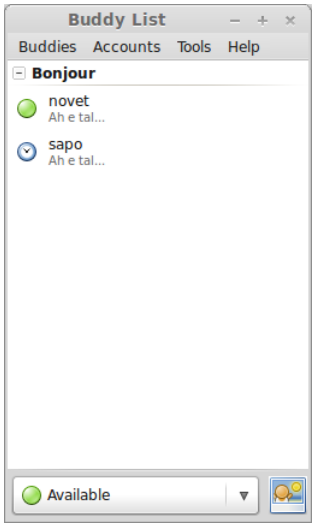

Figure 23- Pidgin, one contact away

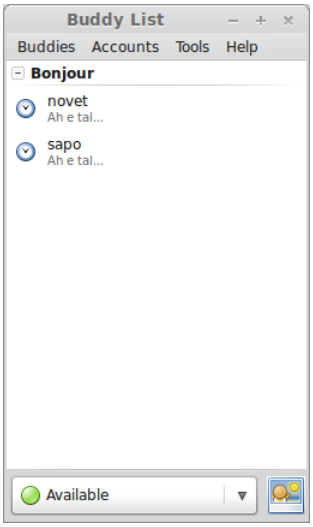

Figure 24- Pidgin, two contacts away

Likewise, when a device exits the application and it unregisters all the services, that contact is removed from the buddy list as Figure 25 shows.

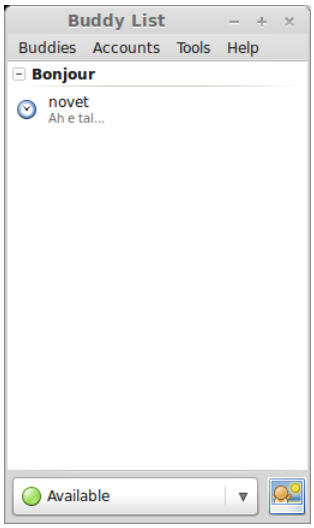

Figure 25- Pidgin, buddy list contact offline

By clicking on a contacts name in the contact list, the user opens the chat interface that can be seen in Figure 26, this interface provides a text input box and a button to send the message and a list of the last messages, as seen in Figure 29 and in Figure 30.

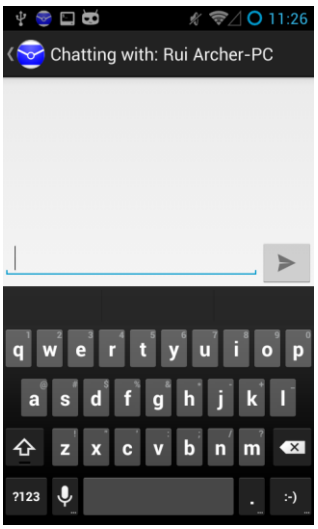

Figure 26- Ad-Hoc Communicator chat interface

When the main application interface is not in focus, the application uses notifications to let the user know when new contacts are available (Figure 27) or new messages have been received (Figure 28).

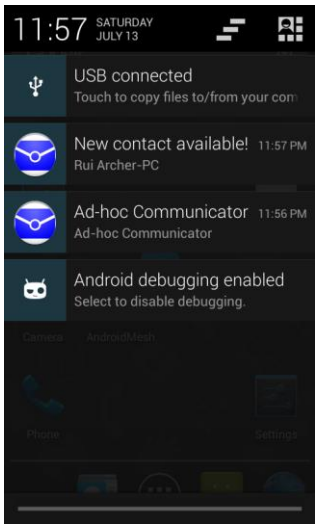

Figure 27- New contact notification

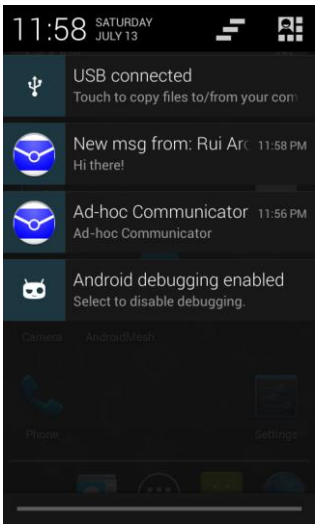

Figure 28- New message notification

Figure 29 and Figure 30 show a conversation from the point of view of the Ad-Hoc Communicator (the contact named sapo), the same conversation is shown from the point of views of the contact using Pidgin (the contact named Rui Archer-PC) on Figure 31.

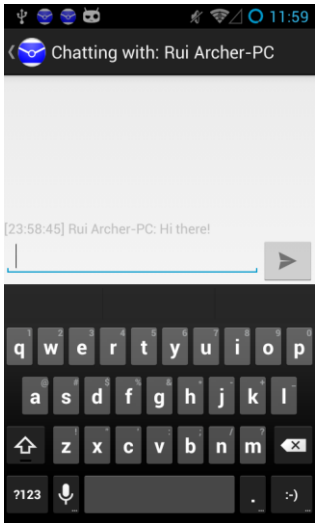

Figure 29- Ad-Hoc Communicator message

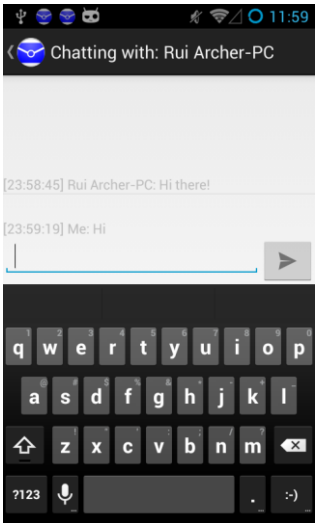

Figure 30- Ad-Hoc Communicator response

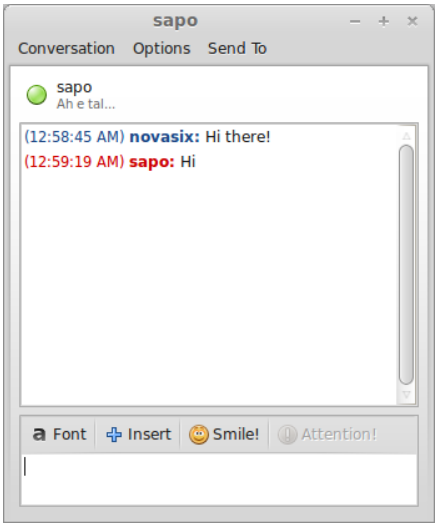

Figure 31- Pidgin conversation

The Ad-Hoc Communicator also supports file transfers, Figure 32 shows pidgin offering to transfer a file, and Figure 33 shows Ad-Hoc Communicator asking the user if he wants to accept or reject the file transfer. When the user accepts the file transfer, the application creates a notification showing the file download progress as seen in Figure 34. Figure 35 and Figure 36 show the files transfer progress and transfer complete dialogs of Pidgin, while Figure 37 shows Ad-Hoc Communicator notifying the user that the files transfer is complete.

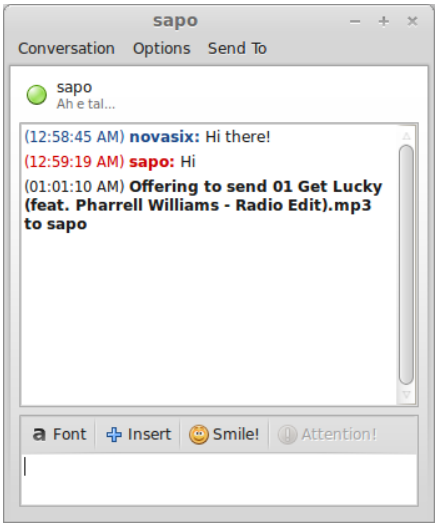

Figure 32- Pidgin file offer

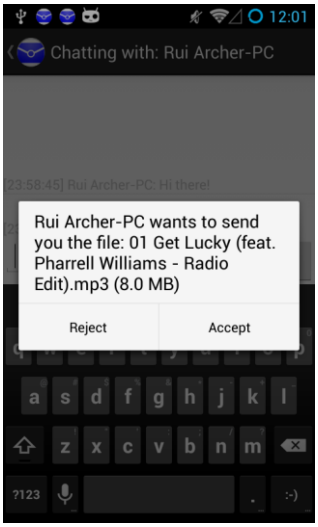

Figure 33- Ad-Hoc Communicator file request

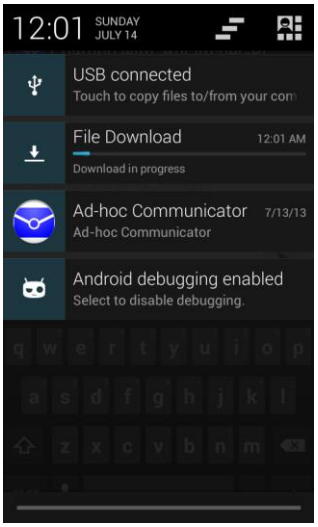

Figure 34- Notification file download progress

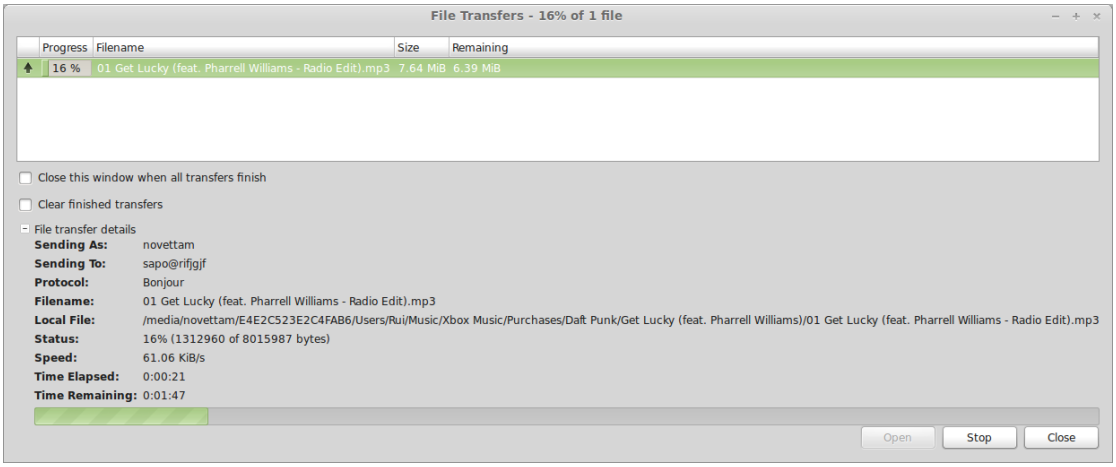

Figure 35- Pidgin file download in progress

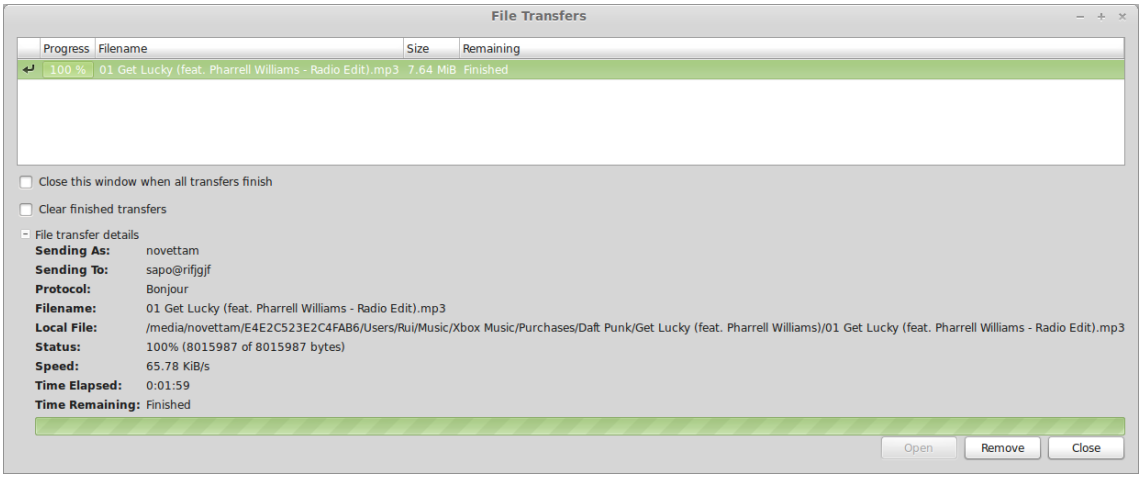

Figure 36- Pidgin file download complete

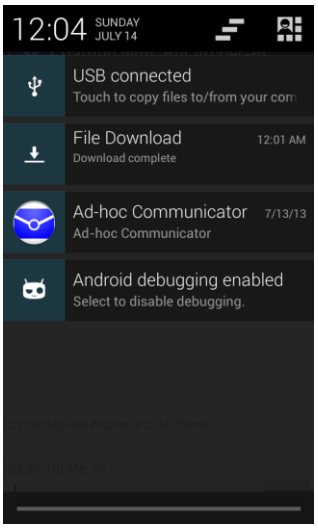

Figure 37- Ad-Hoc Communicator file download complete

5.3 Summary

Both applications present the expected behavior, but not everything is perfect as there are some unknowns, at the time of the test one of the Android devices was not able to receive multicast traffic but was able to broadcast and announce its service. This behavior had been previously observed as well in some devices, further investigation is needed to determine the cause of this behavior.

Chapter 6

Conclusion and Future Work

6.1 Contributions

This thesis provides technological solutions to the creation of wireless mesh networks of mobile devices, and de-centralized service discovery within those networks. The major contributions of this thesis are tools to create and automatically configure a mesh network of Android devices and an implementation of a distributed instant messaging application.

With the AndroidMesh application, the creation of a network composed of Android devices, is possible, without the user needing do manually configure the network or perform other tasks.

Although the usefulness of the application is limited to the devices with Wi-Fi chipset drivers that support ad-hoc mode, the application embodies the knowledge we acquired about what takes to put and Android device in Ad-Hoc mode.

The Ad-Hoc Communicator application covers the use case we set out to develop. A user can launch the application and easily send messages and files to those nearby even in places without network infrastructure. But even in places where that infrastructure is available the application can be useful as well, due to its decentralized nature it does not require a server, many other applications do require a server and it is not located in the local network.

The work conducted in this thesis, although very technical due to the numerous challenges we met, represents a significant step in making ad-hoc networking among Android devices and any other IP capable devices a reality to a wider public beyond researchers and developers.

The use of well-known standards for service discovery and communication protocols, makes our solutions interoperable with other existing applications, one such example of an application is the Pidgin [37] instant messaging client against which we tested by running it on a computer connected to the same ad-hoc network the Android devices were connected to.

This means that future work to advance the use of ad-hoc networking in Android devices, in the future is now easier through our contributions.

6.2 Future Work

Regarding the self-configuration of the Android devices, while we were developing our work, Bruno Randolf [23] published his work on bringing ad-hoc mode right into the core Android OS and submitted patches to the CyanogenMod project, featuring ad-hoc capabilities detection and providing an interface to enable ad-hoc mode integrated into the native network manager interface. With ad-hoc networks recognized as such and being managed appropriately by the network manager this solution is more robust that having an external application.

His solution is still lacking a feature, as the users still need to get into the network manager interface and manually configure the network's IP address. It is needed to add an option to use link local addresses to the network manager, along with the DCHP and Static options, similar to how the gnome network manager presents that option in Linux systems.

The information present on this report on how to generate an IP address for an Android device should be of help, the integration of that code into the Android system would also allow for access to the ARPeer class, and its use for address claiming and conflict detection without the need to install an external tool.

Regarding service discovery, we hope Google completes the work already done and exposes the TXT record for resolved services, allowing for full service discovery capabilities and mitigating the need for an external library, as JmDNS has not been properly maintained for some while now.

With those two aspects solved and properly integrated into the Android OS, we would end up right where we are now, just with a bit more tidy and integrated experience.

In the end we think that for a truly flexible networking experience it is needed that the transition between network modes to be seamless, concepts like redundant connections over different technologies allowing for seamless transition as described in the Haggle [10] architecture are worth exploring. Other concepts like asynchronous networking can also increase the flexibility of networks removing the requirement of uninterrupted connections and allowing for exploitation of opportunistic connectivity.

There are many areas to explore but a solution where a hybrid network architecture combining synchronous and asynchronous networking in a seamless fashion might be worth exploring.

References

[1] J. Wang, B. Xie, and D. P. Agrawal, “Guide to Wireless Mesh Networks,” 2009.

[2] R. Bruno, M. Conti, and E. Gregori, “Mesh networks: commodity multihop ad hoc networks,” *Communications Magazine, IEEE*, no. March, pp. 123–131, 2005.

[3] A. Lindgren and P. Hui, “The quest for a killer app for opportunistic and delay tolerant networks,” *Proceedings of the 4th ACM workshop on Challenged networks - CHANTS ’09*, p. 59, 2009.

[4] A. Sathiaseelan and J. Crowcroft, “Internet on the move,” *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 1, p. 51, Jan. 2012.

[5] “Open Garden - You Are the Network.” [Online]. Available: http://opengarden.com/. [Accessed: 07-Feb-2013].

[6] A. Matsumoto, K. Fujikawa, Y. Okabe, and M. Ohta, “Multihoming support based on mobile node protocol LIN6,” *2003 Symposium on Applications and the Internet Workshops 2003 Proceedings*, 2003.

[7] H. Han, S. Shakkottai, C. V Hollot, R. Srikant, and D. Towsley, “Multi-Path TCP: A Joint Congestion Control and Routing Scheme to Exploit Path Diversity in the Internet,” *IEEE/ACM Transactions on Networking*, vol. 14, pp. 1260–1271, 2006.

[8] Wai-Hong Tarn and Yu-Chee Tseng, “Joint Multi-Channel Link Layer and Multi-Path Routing Design for Wireless Mesh Networks,” in *Proceedings of the 26th IEEE International Conference on Computer Communications INFOCOM 07*, 2007, pp. 2081–2089.

[9] P. Hui, A. Chaintreau, J. Scott, R. Gass, J. Crowcroft, and C. Diot, “Pocket switched networks and human mobility in conference environments,” *Proceeding of the 2005 ACM SIGCOMM workshop on Delaytolerant networking WDTN 05*, pp. 244–251, 2005.

[10] J. Scott, J. Crowcroft, P. Hui, and C. Diot, “Haggle: A networking architecture designed around mobile users,” *WONS 2006: Third Annual …*, 2006.

[11] T.-Y. Huang, C.-M. Lin, J.-R. Jiang, W. T. Ooi, M. Abdallah, and K. Boussetta, “SYMA: A Synchronous Multihop Architecture for Wireless Ad Hoc Multiplayer Games,” in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, 2011, pp. 793–798.

[12] “Proximity-based Peer-to-Peer Mobile Application Development Framework - AllJoyn.” [Online]. Available: https://www.alljoyn.org/. [Accessed: 07-Feb-2013].

[13] “Issue 33666 - android - Download manager doesn’t download over Ethernet or VPN - Android - An Open Handset Alliance Project - Google Project Hosting.” [Online]. Available: https://code.google.com/p/android/issues/detail?id=33666. [Accessed: 13-Jul-2013].

[14] T. Xu, Y. Chen, X. Fu, and P. Hui, “Twittering by cuckoo: decentralized and socio-aware online microblogging services,” *ACM SIGCOMM Computer Communication …*, no. August 2009, pp. 2009–2010, 2010.

[15] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” *Middleware 2001*, no. November 2001, 2001.

[16] “The Serval Project: Practical Wireless Ad-Hoc Mobile Telecommunications.” [Online]. Available: http://developer.servalproject.org/site/docs/2011/Serval_Introduction.html. [Accessed: 07-Feb-2013].

[17] “Serval Mesh - The Serval Project Wiki.” [Online]. Available: http://developer.servalproject.org/dokuwiki/doku.php?id=content:servalmesh:main_page. [Accessed: 13-Jul-2013].

[18] “Samsung Chord | SAMSUNG Developers.” [Online]. Available: http://developer.samsung.com/chord. [Accessed: 13-Jul-2013].

[19] B. Aboba, E. Guttman, and S. Cheshire, “Dynamic Configuration of IPv4 Link-Local Addresses.”

[20] T. Clausen and P. Jacquet, “RFC 3626 - Optimized Link State Routing Protocol (OLSR),” *IETF RFC3626*, 2003. .

[21] S. Cheshire and M. Krochmal, “Multicast DNS.” [Online]. Available: https://datatracker.ietf.org/doc/draft-cheshire-dnsext-multicastdns/. [Accessed: 07-Feb-2013].

[22] S. Cheshire and M. Krochmal, “DNS-Based Service Discovery.” [Online]. Available: https://datatracker.ietf.org/doc/draft-cheshire-dnsext-dns-sd/. [Accessed: 07-Feb-2013].

[23] “Thinktube - Android IBSS.” [Online]. Available: http://www.thinktube.com/android-tech/46-android-wifi-ibss. [Accessed: 13-Jul-2013].

[24] “Android, Windows Phone, and Windows Mobile Development News, Information, and Howtos - XDA Developers.” [Online]. Available: http://www.xda-developers.com/. [Accessed: 13-Jul-2013].

[25] “ClockworkMod.” [Online]. Available: http://www.clockworkmod.com/. [Accessed: 13-Jul-2013].

[26] “BusyBox - Aplicações Android no Google Play.” [Online]. Available: https://play.google.com/store/apps/details?id=stericson.busybox. [Accessed: 13-Jul-2013].

[27] “BusyBox Installer - Aplicações Android no Google Play.” [Online]. Available: https://play.google.com/store/apps/details?id=com.jrummy.busybox.installer. [Accessed: 13-Jul-2013].

[28] T. Bray, “Identifying App Installations | Android Developers Blog.” [Online]. Available: http://android-developers.blogspot.pt/2011/03/identifying-app-installations.html. [Accessed: 22-Mar-2013].

[29] “A DNS RR for specifying the location of services (DNS SRV).” [Online]. Available: http://www.ietf.org/rfc/rfc2782.txt. [Accessed: 26-Jul-2013].

[30] “DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION.” [Online]. Available: http://www.ietf.org/rfc/rfc1035.txt. [Accessed: 26-Jul-2013].

[31] “JmDNS.” [Online]. Available: http://jmdns.sourceforge.net/. [Accessed: 13-Jul-2013].

[32] C. Perkins, E. Belding-Royer, and S. Das, “Ad hoc On-Demand Distance Vector (AODV) Routing,” *Internet RFCs*, vol. 285. pp. 1–38, 2003.

[33] “roottools - RootTools gives Rooted developers easy access to common rooted tools... - Google Project Hosting.” [Online]. Available: https://code.google.com/p/roottools/. [Accessed: 14-Jul-2013].

[34] T. Muldowney, M. Miller, R. Eatmon, and P. Saint-Andre, “SI File Transfer.” XMPP Standards Foundation, 13-Apr-2004.

[35] D. Smith, M. Miller, P. Saint-Andre, and J. Karneges, “SOCKS5 Bytestreams.” XMPP Standards Foundation, 20-Apr-2011.

[36] P. Saint-Andre, “Serverless Messaging.” XMPP Standards Foundation, 26-Nov-2008.

[37] “Pidgin, the universal chat client.” [Online]. Available: http://www.pidgin.im/. [Accessed: 13-Jul-2013].