TÉCNICO
LISBOA

Master Thesis

P2P Communication in Android devices

Tomás Falcato Costa

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Advisor(s)/Supervisor(s):	Prof. António Grilo
	Prof. Paulo Pereira

Examination Committee

Chairperson:	Prof./Dr. Lorem Ipsum
Advisor:	Prof./Dr. Lorem Ipsum
Members of the Committee:	Prof./Dr. Lorem Ipsum
	Prof./Dr. Lorem Ipsum

October 2017

Acknowledgments

Abstract

Your abstract goes here.

Keywords: my keywords

Resumo

Keywords: Comunicação descentralizada, Wi-Fi Direct, Wi-Fi P2P, Redes Ad hoc, Topologia em Malha, Android.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

Nowadays the demand for better mobile devices is higher than ever. Mobile phones are an indispensable gadget in today's society. Increasingly demanding application and connectivity requirements bring the need for devices with more capabilities, *e.g.* battery life, memory, persistent storage, Internet access speeds, *etc.* With this evolution of equipment, inevitably, comes an evolution of communication technologies.

Mobile phones, usually communicate between themselves in different ways: via mobile cellular networks, via Internet access, via Bluetooth, *etc.* New communication technologies are appearing at a fast pace and the possibilities for using them to provide new services for the users are endless.

The main communication methods use a limited number of central points, that coordinate the communication process between devices, acting as mediators in the communication channel. However, from this dependence, a question arises: if there is a limited number of central points what happens if a partial or total failure from their part occurs. This question has an answer in device to device communications.

There are many devices available, usually more than one per person, see [6], making the creation of an ad hoc network a big possibility to overcome possible failures with central points or even if one is not within reach of any central point. Despite that, this answer is not a substitute to the existing communication methods. It aims to add more range and robustness to the network and possibly reduce the workload of the infrastructured network, which has a limited capacity.

Due to the reasons just stated, ad hoc communication between devices has been lately a hot topic, with several applications being currently offered to mobile users *e.g.* FireChat and Ueppa!, see section 2.3. This thesis offers a framework to create applications of this nature. To realise this framework a new application of this kind was implemented which allows users to access web pages using ad hoc links when they are not within range of an access point or base station.

1.2 Problem Statement

Given the context above, the main question is: where can the creation of an ad hoc network be of use to the everyday tasks people perform on their mobile devices. There are many answers to this question, thus the difficulty of choosing a relevant topic. Much of the work currently being done focus on chat applications, where messages are transmitted via an ad hoc network. Bearing this in mind, this thesis takes a further step and creates a framework to form an ad hoc network capable of transmitting packets between devices.

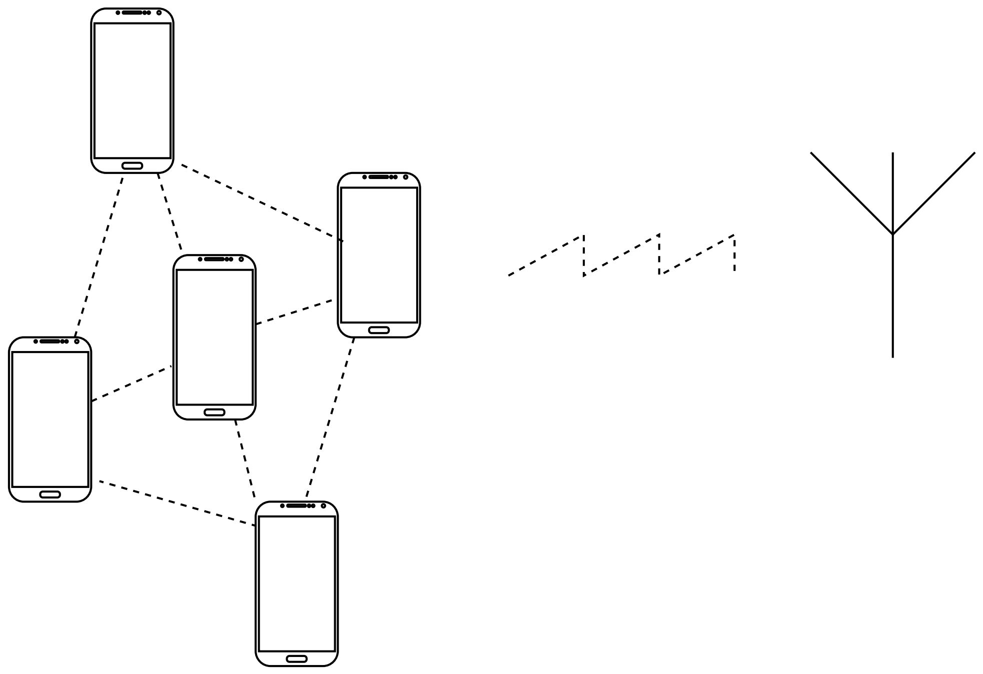


Figure 1.1: Example of a created ad hoc network

A solution is proposed to solve the inability of devices to access web pages, through a peer-to-peer application, even where there is the possibility of indirect Internet access, via communication with other devices and not directly with the infrastructure. The Android hotspot is an existing service that allows devices to share their Internet with the surrounding peers. However, this only works if a device is within immediate range of the hotspotting device, reducing the size and reach of the created network. In the solution proposed in this thesis, the devices are able to reach the Internet by sending their requests to their immediate peers, who will forward them to their neighbors until the destination is reached, travelling more than one hop[1], see the example of figure 1.1.

It is important to note that this work does not have the purpose of replacing the existing communication infrastructure, but is, in fact, trying to complement it.

1.3 Objectives

This thesis will pursue two main objectives:

1. Developing a framework to create of a decentralized ad hoc network, where packets are transmitted between devices. Proving that, with the current unmodified Android versions and the Bluetooth technology, the creation of a network of that nature is feasible.

   To materialize the framework an application that implements this framework and exchanges web pages between devices is created. The application will create a solid ad hoc network. After the creation is complete the application will provide the logic to correctly manage the web pages request throughout the network, as well as their correct delivery. This application will then be submitted to a series of tests to comprehend where it is more vulnerable and where it is more robust.

[1]One hop is the distance of the communication between a device and its immediate peers. Each time the message travels between a device and its peers a hop is added to the message path.

2. The second objective will be to assess the advantages of migrating this application from Bluetooth to Wi-Fi Direct and what changes need to be made to the current Android versions to accommodate this migration. The advantages and disadvantages of Wi-Fi Direct in comparison to Bluetooth will be compared in the scope of the created framework. Conclusions will be drawn from a series of tests to compare both technologies. Also, a description of the obstacles, present in the current Android devices, preventing the development of this application using Wi-Fi Direct instead will be presented.

This thesis will not provide a market product, thus it disregards some aspects of what would be to expect from a full consumer ready application. Security is not developed in this solution, although some ideas are given on how it can be provided.

1.4 Structure of the Thesis

The rest of this thesis is organized as follows:

- Chapter 2 will begin with a theoretical introduction of the different wireless communication technologies, analysing their features, advantages and disadvantages. This aims to provide the needed background to understand the technologies used in this thesis and the choices made along its developments. There will be an analysis on the Android's implementation of Wi-Fi Direct and some of the possibilities it may present, such as ad hoc networking and multi-hop routing. Finally, an overview of some ad hoc networking applications in Android will be given, describing their features and technologies used.
- Chapter 3 will contain the implementation of both framework and application. It will begin by a description of the steps taken to decide important parts of the framework, such as technologies and routing protocols used. The implemented network creation and communication protocols are explained, providing a better understanding of the framework. Lastly, the materialization of the framework, the peer-to-peer application to exchange web page is described, along with its features and overall packet exchanges.

Chapter 2

State of the Art

2.1 Communication Technologies Supported by Mobile Devices

2.1.1 Mobile Networks Technologies

The first form of communication on mobile devices where the mobile cellular telecommunications provided by the Public Land Mobile Network. At first they did not have so many features as we know them now, they were limited to basic voice calls and short text messages. As devices became more sophisticated so did mobile networks, including new features, such as Internet connections and device to device communication.

Mobile networks have become common place, *i.e.*, people make millions of phone calls and text messages everyday, using the service providers' Base Stations (BSs) to enter a network, where their message/phone call is being routed to its destination. This said, it is important in the scope of this work to have some understanding on how devices communicate with each other using these mobile communication standards.

In this subsection we will briefly present the existing standards for mobile cellular networks and their evolution, passing from 2G, 3G and 4G, emphasizing this last one.

2.1.1.1 2G: GSM

Global System for Mobile Communications (GSM) is a standard, created by European Telecommunication Standards Institute, to describe second generation cellular networks. These networks differ from the first generation due to the fact that they were no longer analog, as in 1G, and became digital, allowing for voice as well as text transfer.

GSM's architecture can be seen as hierarchical, with components ranging from Mobile Stations (MSs) to Mobile Switching Centers (MSCs). MSs, the devices, have a unique number, with which a BS can identify each one of the MSs it controls. A Base Station Controller (BSC) controls multiple BSs to allocate radio channels, manage call handover between BSs and control their power levels, in order to avoid muffling the transmission of other MSs. Finally, a MSC, in charge of multiple BSs connects to a Gateway MSC where mobile registration and authentication are made.

GSM uses the air interface to transfer information, being a wireless way of communication, specifically, it uses Frequency Division Duplex (FDD), to separate the uplink and downlink frequencies, 890-915MHz and 935-960MHz, respectively. Then divides each block of frequencies into smaller channels, 125 channels of 200kHz each, using Frequency Division Multiple Access (FDMA). In each FDMA channel it's given a time slot for each MS to use, using Time Division Multiple Access (TDMA). Using this methodology for medium access, GSM allows for a data rate of 9.6kbps per user, after enctryption and error control overhead.

GSM's main technologies are voice communications, Subscriber Identity Module (SIM) authentication, encryption and accounting information, handover, enabling MSs to move and connect to a different BS maintaining the service and SMS (Short Message Service), allowing for text transfer up to 160 characters, sent to one or multiple destinations.

In order to improve GSM, General Packet Radio Service (GPRS) was introduced, also known as the 2.5G networks, adding two new elements to the previous GSM architecture, a service support node for security, mobility and access control, a gateway support node for establishing connections to external packet switched networks. Although not much improvement on data rate was made on GPRS, soon came Enhanced Data Rates for GSM (EDGE), which combined GPRS with different modulations, improving the spectral efficiency of each channel and allowing for data rates up to 384kbps.

GSM requires heavy resource planning, *i.e.*, frequency and time planning and slot assignment, meaning each user has a dedicated time and frequency and thus the number of users in a cell does not influence the cell size.

2.1.1.2 3G: UMTS

Universal Mobile Telecommunications System (UMTS) was the natural 3G evolution of the GSM/GPRS netowrk. It used the previously created GPRS architecture and improved it using different Medium Access Control (MAC) techniques to improve even further spectral efficiency. The architecture of UMTS is divided into radio access network, UMTS Terrestrial Radio Access Network (UTRAN), in charge of managing cell-level mobility, and Radio Network Subsystem (RNS) and air interface, UMTS Terrestrial Radio Access (UTRA), similar to GPRS. Now UTRAN controls multiple RNSs, who is responsible for handover decisions. The UMTS network operates in parallel with the previously established GSM/GPRS network.

In UMTS transmission is made over two 5MHz FDD channels, using Direct Sequence Spread Spectrum (DSSS), improving both the data rate and security of transmissions. Wireless Code Division Multiple Access (W-CDMA) is now used instead of FDMA and TDMA, each user has a chipping sequence with which messages are encoded, in the destination, with the same chipping sequence the reverse process is made and the message is transmitted, allowing for similar data rates as EDGE and users to transmit simultaneously with little interference, depending on the number of users.

UMTS requires heavy power control, because the source can distinguish each user via their chipping sequences, but if one user muffles another user only one message is received in the destination, thus it is needed to control the power with which each user will transmit. This means the more users transmit simultaneously, more interference is created, assuming non ideal conditions, thus having to reduce cell size to compensate for this interference, leading to more complex cell planning.

In order to enchance the data rates of UMTS, High Speed Packet Access (HSPA) was introduced, which is an evolution of W-CDMA, 3.5G networks. This standard improved upling and downlink speeds, by adding higher-order modulation, *e.g.*, 16QAM or 64QAM, and a more efficient retransmission mechanism in the downlink channel and by allowing parallel transmissions of multiple users, also known as Multiple Input Multiple Output (MIMO), reaching rates up to 168Mbps and 22Mbps, respectively.

2.1.1.3 4G: LTE/LTE-A

Long-Term Evolution (LTE), came to meet the specified requirements in International Mobile Telecommunications-Advanced, issued by ITU-R. But since it did not meet all the requirements to be considered a 4G network, it was considered a 3.9G network. It introduced an exclusively IP-based packet-switching core network, denominated Evolved Packet Core (EPC), and it targets the increase of quality of service, spectrum efficiency and reduced cost.

EPC introduced new elements to the existing network, a Packet Data Network Gateway, serving as the termination of EPC towards Internet, providing Internet Protocol (IP) services, address allocation, packet inspection and policy enforcement, a Mobility Management Entity, responsible for location tracking, paging, roaming and handover, and a Policy Charging Rules Function to manage the quality of service provided.

LTE uses multiple frequency bands from 700MHz to 2600MHz, with a flexible bandwidth ranging from 1.4MHz to 20MHz, using both FDD, Time Division Duplex (TDD) and a combination of these two methods. This combined with Orthogonal Frequency Division Multiplexing (OFDM) and MIMO, for MAC, allows LTE to reach data rates of 326Mbps for downlink. In uplink a Single Carrier Frequency Division Multiple Access (SC-FDMA) is used allowing for data rates up to 86Mbps. These data rates are considerably higher than the ones reached by UMTS.

In order to further improve data rates on LTE, LTE-Advanced was introduced. This new network meets the requirements to be considered a 4G network, thus it is where 4G networks were actually introduced. Reaching up to 3Gbps for downlink and 1.5Gbps for uplink, Release 10, LTE-Advance immensely improves data rates by using a much wider channel frequency and higher-order MIMO, up to 100MHz, also improving on spectral efficiency.

A new type of networks is also introduced, the Heterogeneous Networks (HetNets), created by deploying a low-power BS at cell edges to enhance network perfomance. Three types of cells are created with this network: micro or pico cells, where a relay node is used to extend the service to other devices. Femto cells, for indoor coverage at home, offices or malls, where a Home eNodeB serves as relay node for devices inside the femto cell.

Considered for 4G LTE-Advanced was the concept of D2D communications, creating direct links between devices within a small area. This technology would enable the linking of devices by using the cellular spectrum, allowing for data to be transferred from one to the other over short distances, but using a direct connection. This form of device to device has a lot of applications, *e.g.*, in disaster scenarios, where the access to the infrastructure is denied, or when the infrastructure is overloaded in *e.g.*, large public events.

4G LTE-Advanced D2D was a feature in Release 12 and brought some benefits, such as reliable and persistent communication, meaning it persists if the LTE network is disrupted, data rates, when the distance to an BS is considerable and interference reduction, by not having to communicate directly with the BS overloading the network. There are of course some issues to be addressed with this communication, such as the authorization and authentication of MSs and the fact that inter-operator communication may not be approved by the different operators, limiting the possible links.

2.1.2 Wi-Fi (IEEE 802.11)

A Wireless Local Area Network (WLAN) is a wireless network that can connect multiple devices to each other within a limited range. WLANs have become very popular in the day to day life of people. Most households have a WLAN deployed so that devices inside and around the premises can have Internet access. The popularity of WLANs is mainly due to the fact that devices do not need to be physically connected to the Access Point (AP) to access the Internet, which removes the costs of cables and the associated infrastructures.

IEEE 802.11 is the *de facto* standard for WLANs and it is commonly known as Wi-Fi. IEEE 802.11 is composed by MAC layer and Physical layer specifications for WLAN implementations, various versions have been released, but the most common are IEEE 802.11a/b/g/n/ac, mainly because they were adopted by mobile and computer manufacturers as the standard to be used in radio communication, *i.e.* wireless.

Unfortunately, the connection speeds are not as high as in wired networks, since the environment plays a big role, *i.e.* if there are obstacles between the AP and devices. Also, the number of devices in the network will affect the data speeds, since the protocol specified for the IEEE 802.11 standard is Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). Although these are disadvantages over wired technologies, there has been done a lot of improvement on the data rates, during the development of wireless networks, reaching up to 6.93Gbps, using MIMO, high-order OFDM and enhanced MAC techniques, see [7] for more information on this topic.

In CSMA/CA the devices check the medium for clearance, *i.e.* if there is no other device transmitting at that time. Nodes attempt to avoid collisions by transmitting the full data only when the medium is clear. Hidden node is still a problem with this MAC method, meaning a node can be transmitting but its transmission is not detected by other devices, creating collisions.

In order to overcome the hidden node problem, Request to Send (RTS)/Clear to Send (CTS) can be used to reserve the access to the shared medium. A control packet RTS is sent by the transmitter, to which the receiver will answer with a CTS. If the channel is clear the packet will be sent immediately, *i.e.* if a CTS was received, else the device will wait a random period of time, named backoff time, before checking if the medium is clear again. When the backoff timer reaches zero, the device will perform the check, if the medium is clear the device will send the packet, otherwise the backoff time is set again. Due to the exponential factor of this backoff time the connections' speeds are limited when multiple users use exhaustively the network channel, whereas in wired connections, like switched Ethernet, traffic management is, typically, done through traffic flow prioritization.

Another main consequence of establishing a WLAN is the security of the communication. In wired networks there is a physical component to security, such as controlled access to the building. In WLANs networks this type of security is not relevant as, for instance, one can enter the network outside the building. Thus security protocols must be implemented to successfully prevent attacks on the communication between devices, such as WPA2 or IEEE 802.11i, see [8] for more information on security protocols.

A WLAN can have three different main modes of operation, infrastructure, ad hoc mode and Wi-Fi P2P. Besides these three main modes, IEEE 802.11s will also be briefly explained due to its characteristics and similarities with this work. In the next subsections these modes will be explained, and provided of pros and cons.

2.1.2.1 Infrastructure

Infrastructure is the most common method, usually deployed and made accessible by a local Internet Service Provider (ISP) or by a Local Area Network. The structure of the network is as follows, there is a wireless AP that manages the various devices on the network and provides the Internet access, this AP can be either wired, *e.g.* fiber, or wireless connected to network backbone. The AP is responsible for the creation and maintenance of the network, it generates an SSID with which the network will be identified, aswell as a security level, *e.g.* Wi-Fi Protected Access (WPA) or Wi-Fi Protected Setup (WPS).

MSs can then connect to the network via wireless or wired connections. In the wired connections no authentication is required as there is a physical connection and usually higher data throughput is achieved. In wireless connections there is the need of first identifying the network to which the MS wishes to connect and then to proceed according to the security level used by the AP. The access to the network is managed by the AP, the wired connections are, usually, granted priority over the network, whereas wireless connections compete for the usage of the air interface, typically, via a predefined MAC protocol, being the most used CSMA/CA.

An example of the infrastructure mode network layout can be seen in Figure 2.1.

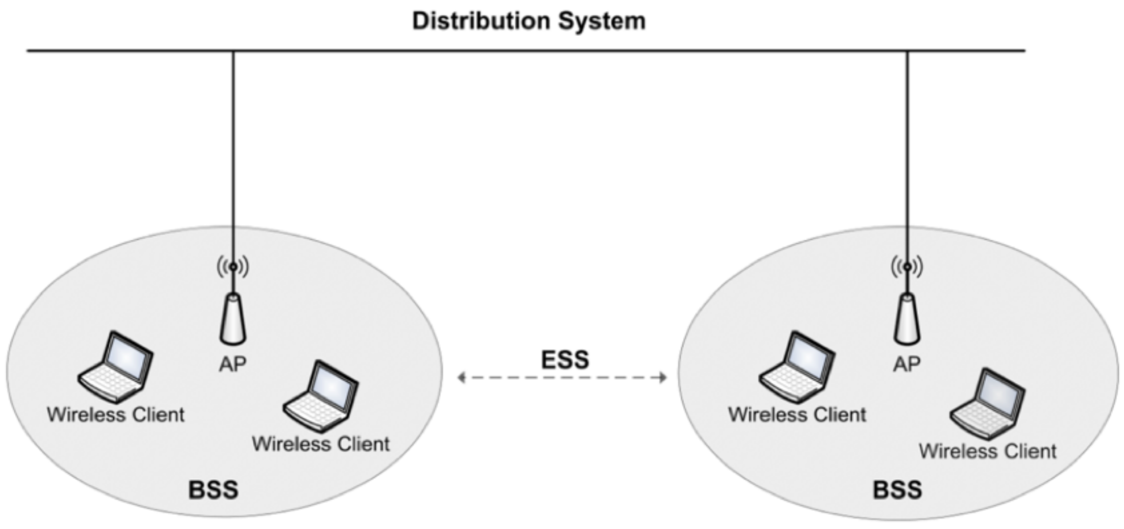


Figure 2.1: Infrastructure network layout (source: www.cse.wustl.edu)

Where each Base Station Subsystem (BSS) is a network and Extended Service Set (ESS) is a set of BSSs that form a single sub network.

The infrastructure mode is ideal if the network as a more permanent character, since the APs are, usually, developed to provide higher-power wireless radios and antennas so that the area covered by the APs is wider. Despite being the most widely deployed method there are some disadvantages associated with this method, for instance, two MSs will not be able to communicate directly even if they reside in the same network, and all their traffic is routed by the AP, which brings another problem: in case of AP failure due to, *e.g.* power failure, software failure, *etc.*, all the network will be compromised and to establish an Internet connection the MSs will have to either connect to another AP or to create the connection by themselves, which brings us to the next subsection.

2.1.2.2 Ad hoc Networks

In the ad hoc mode there is no need for a centralized AP, meaning all the devices can connect to each other if within range. An ad hoc network is slightly different from a Wi-Fi Direct network (WIFI P2P) that will be described in the next section. In ad hoc mode the network is meshed, and all the devices within it are peers, which brings some benefits, *e.g.* the direct communication between devices, without depending on a centralized point connected to the distribution system.

In ad hoc networks with a mesh topology there can be peer-to-peer exchange of traffic. This helps with the problem of having a centralized point of failure, such as the one present in infrastructure mode. Much like *torrents* files can be transferred by smaller parts and by different providers, by using this method to transfer files, packets, *etc.*, within the network higher data rates can be achieved, since multiple parts of the same file/packet can be sent simultaneously.

An example of the ad hoc mode network with a mesh topology can be seen in Figure 2.2.

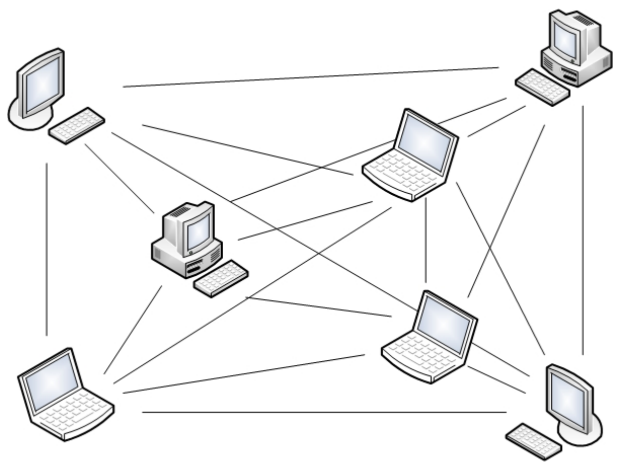


Figure 2.2: Ad hoc network layout (source: monet.postech.ac.kr)

As can be seen in Figure 2.2, one device can have multiple links connecting him to other devices in the network. Although this is a big plus if compared with infrastructure mode, it must be noted that, as said before, a device can connect to other devices if in range, otherwise they will not be able to establish a link and thus communicate. This limitation is due to the lack of a routing protocol in this network mode, so nodes cannot serve as relay for communication between two Stations (STAs), and although it is possible to implement a layer 3 (Network Layer) routing protocol with this mode, such as Ad hoc On-Demand Distance Vector (AODV), it is not intrinsic to this mode. In the next subsection, IEEE 802.11s will be introduced and we will see that it covers this limitation of this ad hoc mode.

Although there are advantages such as not having a centralized point of failure, peer-to-peer file/packet transfer described above, there is an easier setup of the network and the problem described in the infrastructure mode of not existing a direct connection between two STAs is now mitigated as every STA in the network is a peer, there are some disadvantages associated with this network mode, such as the network being more dynamic which brings a lot of changes in the network topology, the interference inherent to all the devices transmitting at the same time to different peers and there is always the scalability problem as more devices in the network mean more connections, which grow exponentially, whereas in the infrastructure mode the connections grow only linearly, so ad hoc networks don't scale well. Also, due to the lack of a routing protocol, the range of the the network will be significantly reduced, as devices do not know which route to forward the packets, in order to reach a certain destination.

Furthermore, the network will not be able to reach the Internet, since devices will communicate between themselves and not with the infrastructure, making the packet exchange limited to the local/cached information stored in the devices, unless if the infrastructure and ad hoc networks are connected through a common device. Finally, the mobility of the devices can make the maintenance of stability of the network a difficult task as links may have to be created and destroyed regularly.

2.1.2.3 IEEE 802.11s (Meshed Network)

IEEE 802.11s is a standard introduced in 2011 which aimed to provide both broadcast and unicast delivery of information. The main difference between the previously described ad hoc mode is that IEEE 802.11s supports multi-hop and implements a layer 2 routing protocol named Hybrid Wireless Mesh Protocol (HWMP).

In this standard, four main types of devices exist: Mesh Points (MPs) who establishes peer links with other MPs, Mesh APs (MAPs) that is a characteristic of MPs which provides BSS services to support communication with STAs, STAs which are devices outside the meshed WLAN and connect to the network via MAP, finally, Mesh Portal (MPP) that is the point at which devices enter and exit the network.

MPs discover potential neighbors based on beacon and probe messages, containing the WLAN Mesh Capability Element, a summary of active protocols and other channel information, and the Mesh ID, that identifies the mesh. The devices are considered to be members of the network upon the establishment of a secure peer link with neighbors within the network. In Figure 2.3, taken from [1], it is possible to visualize an example of the network:

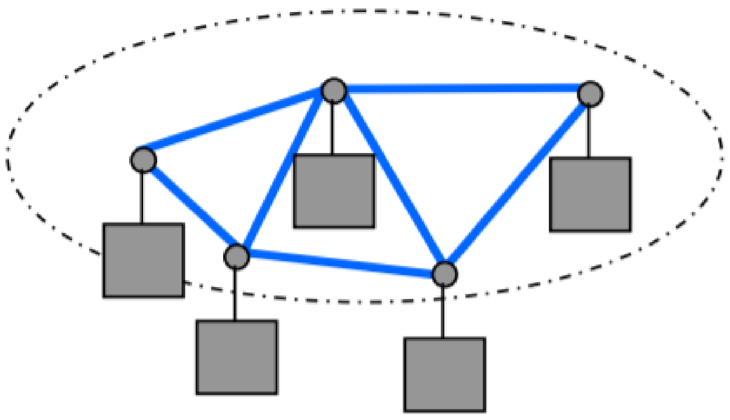


Figure 2.3: IEEE 802.11s network layout (source: [1])

Depending on the number of radio interfaces the devices have, IEEE 802.11s allows for multi-group formation, where each radio interface of each device is assigned to a different group, called Unified Channel Graph.

The routing protocol used by IEEE 802.11s, HWMP, is based on a combination of Radio Metric AODV and tree-based routing, which provides great flexibility in changing environments, great efficiency in fixed mesh deployments, and possible extensibility to metrics other than simple number of hops, such as quality of service, load balancing and power-aware. With this features, HWMP extinguished many of the ad hoc mode flaws, such as lack of routing protocol and thus inability to perform multi-hop transfer of packets and network range.

Although IEEE 802.11s comes with some benefits, pure ad hoc mode still predominates as the peer-to-peer mode, due to its simplicity. There are still some products that make use of the IEEE 802.11s, such as Linux operating system, FreeBSD operating system and Google WiFi routers.

2.1.2.4 Wi-Fi Direct

Wi-Fi Direct is a Wi-Fi standard created by Wi-Fi Alliance. The previously called Wi-Fi P2P, now Wi-Fi Direct, is an innovative way of mobile communication without the dependence of a physical AP. It can be used for different purposes, such as file transfer, Internet browsing, device communication, *etc.*. Wi-Fi Direct assumes an ad hoc topology, meaning the devices are not dependent on one another, but form a network where all devices share information, hence called peer-to-peer. In figure 2.4, it is possible to see the difference between traditional infrastructured Wi-Fi (to the right) and Wi-Fi Direct (to the left).

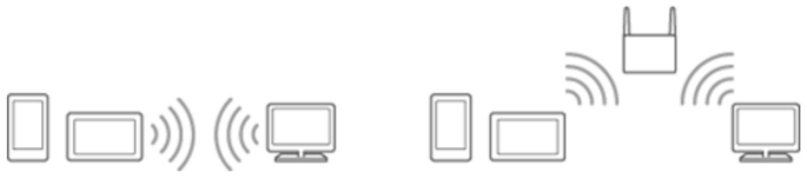


Figure 2.4: Wi-Fi Direct (left) and traditional Wi-Fi (right) network layouts (source: info.tvsideview.sony.net)

Wi-Fi Direct is not dependent on an infrastructure, meaning even without access to a Wi-Fi network it is possible for devices to connect with each other, this because the Wi-Fi Direct enables devices to emit a signal to other devices in the vicinity announcing the possibility of making a connection. Users in the vicinity of the sending device receive an invitation to join a network (Wi-Fi Direct group).

The process of group creation and administration is the most important topic to this work, regarding this technology. Devices can either join existing groups or create new groups, where they will be the administrators, *a.k.a.* Group Owners (GOs) of that particular network. This type of creation forces the Wi-Fi Direct to shape its topology as a star, as is evidenced in figure 2.4, where there is a central soft AP. It is important to make clear the distinction between a soft and a physical AP: the physical AP usually refers to a physical router, that administrates a network with wired and/or wireless devices, whereas the soft AP can be set up with a Wi-Fi adapter, present in many devices, such as mobile phones, computers, *etc.*.

After the creation of a group, the GO announces to all nearby devices its group, via the Service Discovery protocol, that sends a beacon packet with an Service Set Identifier (SSID), that will be the identifier of the network. Then, the receiving devices can connect to the desired network, by sending information about the device and what type of services it supports. Along with the unique identifier of that device, when received by the GO, the devices become Group Members (GMs) of that network, much like slaves in Bluetooth, see 2.1.3.

In traditional Wi-Fi Direct, the connections are one-to-one or one-to-many, limiting the topology to a star topology, the purpose of this work is to migrate from that star topology to a more dense meshed topology where many-to-many connections are established, and the transfer of data is made faster and without as many relay nodes as in star topology.

The speeds of Wi-Fi Direct are similar to the ones in other Wi-Fi operating modes, reaching up to 250 Mbps. This is the main advantage of Wi-Fi Direct to its direct competitors, such as Bluetooth. As in other wireless technologies, the speed is affected by the environment where the network is inserted, the physical characteristics of the devices and the Wi-Fi physical layer they support, *e.g.*, 802.11a, g or n.

2.1.3 Bluetooth

A Wireless Personal Area Network (WPAN) is type of network where devices are connected wirelessly to each other, based on the standard IEEE 802.15. This definition seems to be quite similar to the one of WLAN, but there is a considerable number of differences between the two. The term personal area network derives from the use that is to be given to such networks, in other words, WPANs are to be deployed in order to connect multiple devices of one's personal area, such as home, office, *etc.*.

Bluetooth is one of the main technologies to implement a WPAN, described above. It uses the 2.4GHz Industrial, Scientific and Medical (ISM) band, it was invented by phone company Ericsson, and is used to connect devices in a short range network.

Devices connect to each other forming *piconets*, which is the term assigned to designate an ad hoc network formed by devices using Bluetooth. Each *piconet* has a master, the device that controls the network, similarly to an AP without providing access to the infrastructured network. Associated to each master there can be up to seven slaves, which are devices that take part in the same *piconet*. Multiple *piconets* can connect between themselves and form a *scatternet*, as seen in figure 2.5:

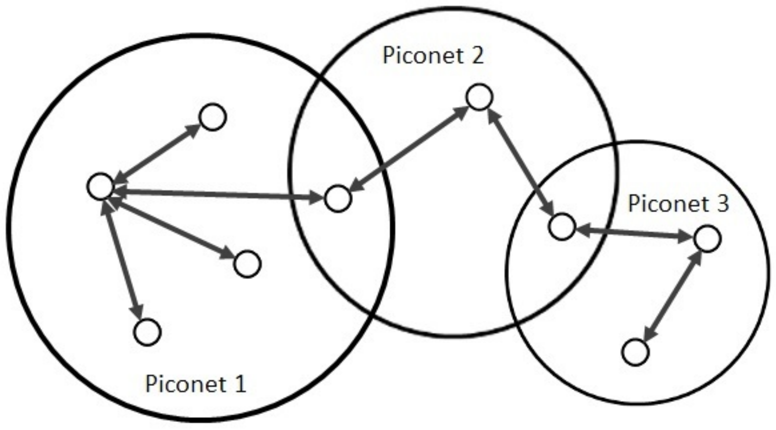


Figure 2.5: Scatternet Layout (adapted from: www.summitdata.com)

Each master is associated with a certain number of slaves, that can participate in more than one piconet, as seen above. These slaves are responsible for coordinating both *piconets*, so that no interference exists.

Bluetooth uses slow Frequency Hop Spread Spectrum (FHSS) to control the frequency of transmission of each slave, this is done by creating a hopping sequence partially based on the master device's MAC address, and then distributing the sequence to each slave on the *piconet*. Devices connect to the *piconet* by pairing with the master, forming a secure link, the master then controls the access to the medium by deciding which slave will transmit at a certain moment in time. In the *scatternet* case, the data to be transmitted from *piconet* to *piconet* is relayed by the node participating in both networks. The pairing of both *piconets* is similar to the pairing of a master and a new member of the network.

Although several advantages of Bluetooth are clear, such as low power wireless protocol, low transmission headers, ease of set up, multi group information transfer, this protocol still has a lot of downsides. The main current problems with Bluetooth consist in the low range of connections, due to the week signal power being a feature of the technology, the limited number of users that can form a *piconet*, due to the narrow band used by Bluetooth and, finally, the low data rates of the protocol, which are heavily surpassed by the Wi-Fi standard.

WPAN uses, typically, technologies that allow communication between devices within a certain specific range, usually around 10 meters, making this type of network much smaller than the ones created by WLAN. The most common technology is Bluetooth, although there are several other technologies that are beginning to raise awareness, due to the crescent interest taken in Internet of Things (IoT), such as ZigBee. The used radio band is the 2.4 GHz ISM band, due to its general availability worldwide and its lower cost.

Bluetooth Low Energy (BLE) is the power-version of Bluetooth developed for IoT. The natural power-efficiency of Bluetooth combined with lower energy consumption provides the key factors for devices running for long periods of time without recharging. BLE's key features include: standard wireless protocol, allowing for interoperability across platforms, low idle power consumption and data encryption for security of communications, among others.

BLE achieves data rates similar to classic Bluetooth over the same distance, although the application throughputs are much smaller: 0.27Mbps compared to 0.7-2.1Mbps for classic Bluetooth. The spectrum range of BLE is the same as in classic Bluetooth, but the channels are two times wider than in Classic, consequently, there is half the number of channels. The main difference between the two lies in the power consumption and peak current consumption, 0.01-0.5W and less than 15mA for BLE, 1W and less than 30mA for classic Bluetooth. The number of slaves in BLE is implementation dependent as opposed to the fixed seven in classic Bluetooth.

BLE is appropriate for networks that rely on the longevity of the battery of the devices, application throughputs are smaller but the information to be sent is, usually, also much less than in classic Bluetooth. One of the biggest limitation of BLE is the inability to transmit voice, whereas classic Bluetooth is able. Despite this and other disadvantages of BLE there are many areas that can benefit from technologies such as this one, *e.g.* healthcare applications, logistic sensors, sports, among others, so it is not a technology that should be overlooked.

2.1.4 NFC (Near Field Communication)

Near Field Communication (NFC) is a short-range high frequency wireless communication technology that allows data transfer between devices over small distances, first introduced by three manufacturers, Sony, Nokia and Philips. Communications maintain interoperability between other different communication methods, such as Bluetooth.

NFC can be used to connect mobile applications with the physical world, *e.g.* home appliances, connect devices through physical proximity, forming a peer-to-peer network, and card emulation, creating a connection to a common infrastructure and allowing some actions on the infrastructure, such as making payments.

Evolved from Radio Frequency Identification (RFID) technology, an NFC chip operates as one part of a wireless link. Once it is activated by another chip, small amounts of data can be exchanged between the pair if within a few centimeters from each other. One of the advantages, compared to other wireless communication technologies, is that NFC does not requires a setup to pair two devices, thus reducing the time and packets exchanged in the transaction, allowing for times up to 1ms. Also, NFC chips run on low amounts of power, making this technology much more power-efficient than other technologies.

The short range of the NFC technology is a disadvantage due to its spacial limitation, but in terms of security, this spacial requirement provides a higher degree of security than Bluetooth. For instance, making NFC relatively secure to use in crowded areas, where other wireless technologies could be impossible to use to transfer sensitive data, such as credit card data.

NFC operates at 13.56MHz using Amplitude Shift Keying as the modulation scheme and TDD for simultaneous receive and transfer of data, achieving data rates up to 424kbps, and although these rates are not impressive, for the amount of data that is sought to exchange using this technology and the lack of necessity for communication setup, NFC provides relatively fast transfer times.

2.1.5 Conclusion

Mobile network technologies have been improving at a fast pace, since the demand for higher speeds is constant. With the evolution of modulation methods to higher-orders and MAC protocols improving spectrum efficiency and number of users in the network without interference, the demand for higher speeds has been successfully answered. 5G networks should focus further on resource optimization and a massively distributed MIMO system.

Wi-Fi technology has many different standards, some being the natural evolution of the others, some serving different purposes, such as IEEE 802.11s. The Wi-Fi infrastructure mode has been constantly updated with better MAC and modulation techniques, allowing for higher data rates and more users on the network. Ad hoc networks have also been evolving being the best candidate to offload some of the traffic in the infrastructure mode, also to achieve smaller, independent networks. Wi-Fi Direct has appeared as a possible method to implement ad hoc networks. Its support is still limited in devices, only allowing for some of the features it can provide. Progresses must be made in order to utilize this technology to its full capabilities.

Bluetooth has had a similar development to the IEEE 802.11 standard, evolving to faster data rates from version to version. Used for smaller networks than Wi-Fi, Bluetooth is widely used to deploy WPANs, now with the concurrence of Wi-Fi Direct, although they can both be used simultaneously. BLE was also a big development in low energy networks, allowing for fast data transfer with low power consumption. Each Bluetooth technology has its utility, and the future focus should be in expanding the number of allowed users and range of the networks.

Finally, NFC provides technology for yet another type of network. This time with a range even smaller than WPANs. It has a lot of applications but it's widely known for its easy and secure usability in transactions. It is being researched by industry giants like Amazon and AliBaba, but there is still much room for improvement, in security, network range, data rates, *etc.*.

Overall we can say that most technologies have met huge improvements in short periods of time, and the tendency is to continue that way. Data rates will continue to grow as higher-level modulation techniques are discovered and new MAC protocols are proposed. More emphasis has been given to smaller device to device networks in later years, has a way to take some load from the infrastructure, or even to for networks relevant to day to day tasks (IoT).

2.2 Wi-Fi Direct in Android

For the scope of this work we are specifically interested in the way Wi-Fi Direct is implemented in Android devices. There are small differences depending on the operating systems in which Wi-Fi Direct is being implemented, thus it is not possible to universally describe Wi-Fi Direct with more detail than the one used in 2.1.2.4.

In the next section the details intrinsic to the Android operating system will be introduced and explained, followed by an in-depth description of various works on how to improve and expand the functionalities of this implementation.

2.2.1 Wi-Fi Direct Star Formation

As previously mentioned in 2.1.2.4, a peer-to-peer network is implemented in Wi-Fi Direct, by using a protocol for discovery and connection of the GO with the GMs. The GO functions as a typical Wi-Fi AP, managing the different communications of the GMs.

GOs are not predefined. It is during the group creation that the actual GO is chosen, according to the specified parameters of the protocol, *e.g.* battery percentage. This feature is relevant to manage the vitality of the network as a mesh of groups, since the GOs can be chosen dynamically extending the life of the network.

After the process of the group creation, the GO periodically sends a beacon to advertise the group, enabling other devices to discover and join the group. This advertisement is made in two different ways, either via Wi-Fi Direct, or via typical Wi-Fi. In the first way, devices discover the network via the Wi-Fi Direct discovery protocol, and join using the described set of actions. In the second way, the GO announces the SSID of the network and other devices, also known as legacy clients, connect via infrastructure mode Wi-Fi, using the SSID and password, if set, to identify the GO's network. This leaves us with two different types of clients: legacy and normal. This differentiation will be the key to overcome the lack of multi-group interaction.

In Android devices, IP addresses are predefined according to the function the device performs within the network. The GOs are automatically assigned the following IP address: 192.168.49.1/24, using Dynamic Host Configuration Protocol (DHCP). Whenever a P2P or legacy client connects to the group, DHCP is run again and the clients take an IP address ranging from 192.168.49.2/24 to 192.168.49.254/24, chosen randomly to minimize the chance of conflicts. The GOs are always assigned the same IP addresses, unless they participate in another group as a GM taking the same IP as a regular client - see [2] for a more detailed explanation on this topic.

This technology is still not implemented to the best of its capacity in Android devices. The lack of a routing protocol that can establish multi-group communication, establishing a meshed network is an essential tool to achieve ad-hoc networks. The current state of this technology in Android devices is a single group network with one-to-many links established from the GO to the GMs, which limits both the range and scalability of the network. Wi-Fi Alliance states that it is possible to overcome these limitations using stock devices, by creating software to allow for multi-group formation, although it is not standardized in Android's current version 7.0 ”Nougat”.

In the next section, a collection of developed work will be presented where the different authors propose different methodologies to overcome the lack of this feature in Android devices.

2.2.2 Ad hoc Networking

C. Casetti *et al.* propose in [2] a process to successfully form a meshed network, by allowing multiple groups to communicate. This proposition is based on stock Android, not requiring any ”root” to be made to the devices, meaning all the actions will be performed in application layer, not envolving any changes in IP addresses or MAC interfaces.

The authors state that multi-group formation can be implemented by taking advantage of both virtual network interfaces of a device, *i.e.*, the Wi-Fi or legacy interface and the Wi-Fi P2P interface, using each one to act as bridge in each group.

This said, upon experimentation, the following scenarios are not feasible in stock Android, due to the inability of creating a custom virtual network:

- a device is the GO of one group and GM in another,
- a devices is the GO of two or more groups,
- a devices is a GM in two or more groups (non-legacy).

Due to these limitations, the authors propose that a GO be a legacy client in a different group, seen in 2.6:

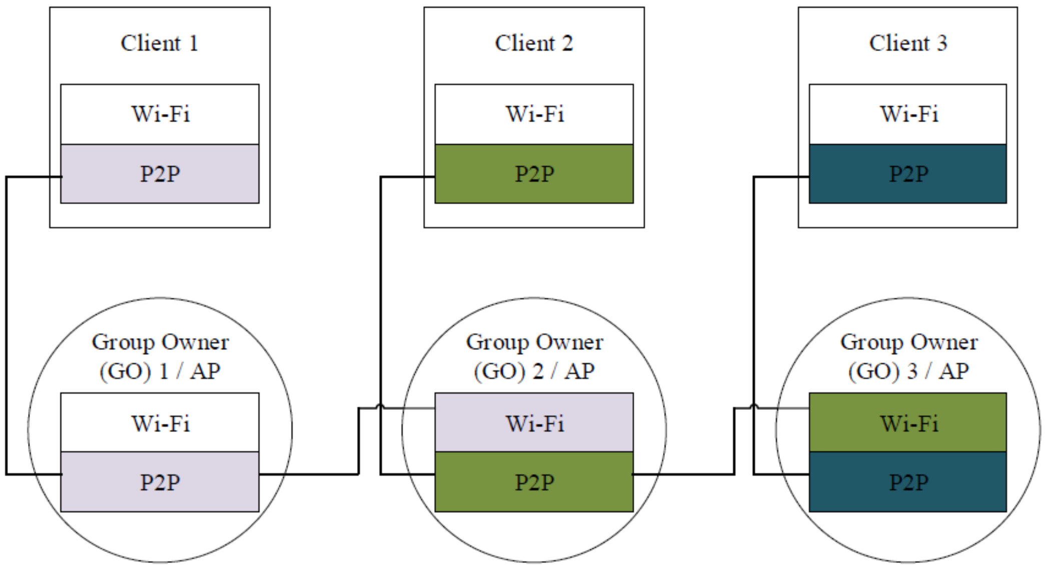


Figure 2.6: Multi-group physical topology with six devices (source: [2])

So, for each GO two network interfaces are enabled, one is the conventional Wi-Fi and the other used for Wi-Fi Direct connection. The IP addresses are assigned according to the previous description.

Two cases are distinguished by the authors: the GO is not connected to any other group as a legacy client, which is the default topology of the network. In this case all connections are feasible, as Wi-Fi Direct has been implemented in order to provide full connectivity among all devices of a single group.

In the second case, the GO is connected to another group as a legacy client as depicted in Figure 2.6 Groups 2 and 3, limiting data transfer to only a subset of D2D data. These limitation are due to two reasons, first the IP conflict of both GOs, who share the same address, 192.168.49.1, making the communication between two adjacent GOs impossible. Secondly, when a GO wants to send a unicast packet to any client, the packet is sent through the GO's Wi-Fi interface, due to Android's implementation of routing table entries in the GO.

So in this case, client-to-GO communication is allowed since client routing tables list only one interface and there are no conflicts, in GO-to-client direction, bidirectional unicast communication is not allowed. Broadcast communication on the other hand is possible, since it is always sent through the GO's P2P interface. Although when they reach the GO acting as a legacy client they are dropped due to the IP address conflict mentioned above. Finally, client-to-client communication is bidirectional and sent through the client's P2P interface.

It is known from the first case that full connectivity among devices in a single group is allowed, even if one of the GMs is a legacy client and GO of another group. Based on this, the authors introduce the term relay node, which is used to describe this legacy client. The relay node is used to connect two groups. This node is chosen at random, upon the sending of a message from the GO to one of its clients chosen at random among those who do not act as GO in another group. It is important to note that the authors state that this message must be broadcasted to avoid sending it through the Wi-Fi interface, the problem described above in the second case. These clients provide the communication backbone and provide connectivity to all other clients in the group, except from the ones acting as GOs in another group.

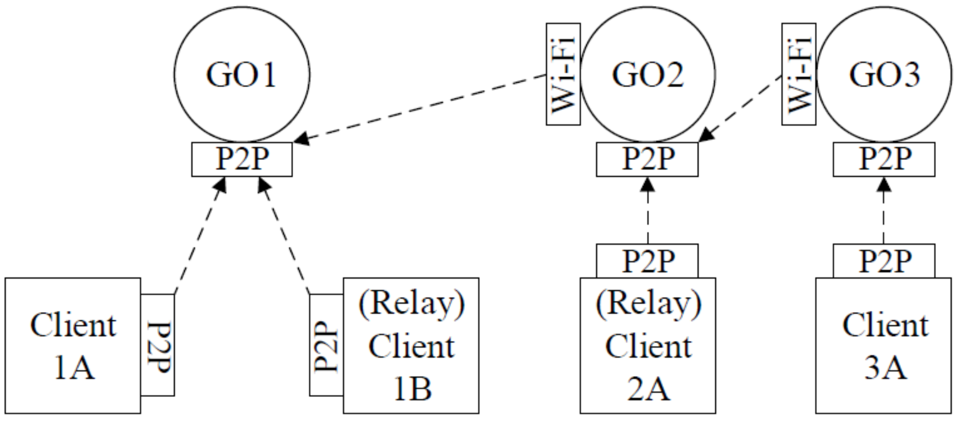


Figure 2.7: Example network topology with 3 Wi-Fi Direct groups. (source: [2])

Take Figure 2.7, where only Wi-Fi Direct connections are represented, for instance. The procedure for Client 1A to send a packet to Client 3A is as follows: Client 1A encapsulates the data in the payload of a unicast User Datagram Protocol (UDP) packet and sends it to the relay Client 1B. The packet will be forwarded by GO1 to Client 1B, at the MAC layer.

The packet is processed at the application layer and the payload is duplicated into a new UDP packet. Sent directly to GO2's standard Wi-Fi interface IP address. At the MAC layer, the packet is sent to GO1, which sends the packet to GO2, via Wi-Fi direct.

The same process is repeated by GO2, but the packet is sent as a broadcast IP packet through GO2's Wi-Fi Direct interface, to relay Client 2A. This client replicates the process of relay Client 1B, sending it to the IP address of GO3's Wi-Fi interface.

Finally, GO3 processes the received packet and sends it to its destination with the correct payload, broadcasting it, similarly to the procedure of GO2.

This mechanism is used following the second case, where the GO is connected to another group as a legacy client. With this mechanism the packet is successfully sent from group to group until its destination is reached, using a mix of unicast and broadcast communication.

C. Funai *et al.* propose a similar approach in [3]. Their approach is to test two distinct possibilities for multi-group formation, as seen in Figures 2.8 and 2.9, where ”LC” stands for legacy client and refers to clients that use the classic Wi-Fi interface, instead of the P2P interface to connect to a group.

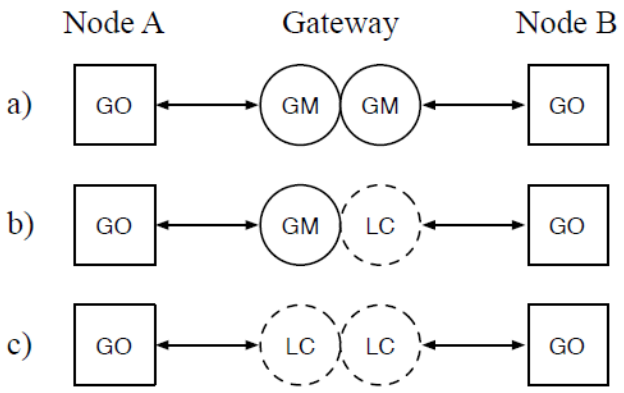


Figure 2.8: Multi-group communication scenarios where the gateway node acts as a client in two groups (source: [3])

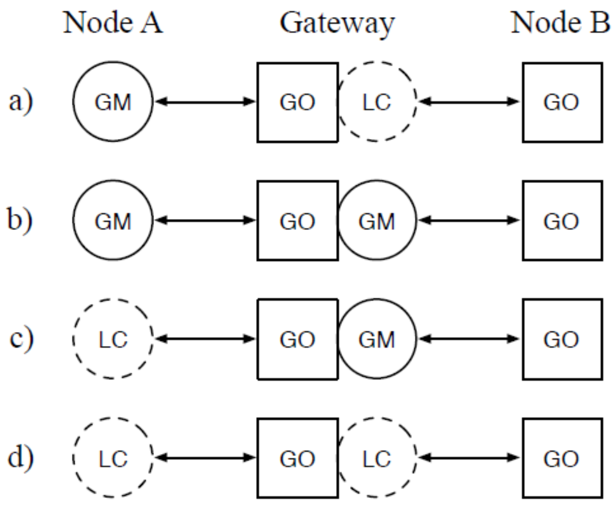


Figure 2.9: Multi-group communication scenarios where the gateway node acts as the GO in one group and as a client in the other (source: [3])

The term gateway node is introduced, referring to the device that connects multiple groups. First by iteratively switching between the different P2P groups, relaying data between them. Secondly by using UDP-based broadcast and a UDP/Transmission Control Protocol (TCP) hybrid solution to achieve multi-group communication. And finally by modifying the source code of the Android operating system.

The authors describe the limitations of stock Android, one of which the multi-group communication must be handled at the application layer and not at the network layer. Actions such as setting IP addresses and managing routing tables cannot be performed without reprogramming the operating system of the device. Another limitation, already referred in [2], is the inability to create virtual network interfaces or multiple virtual MAC entities.

These limitation result in the failure of direct implementation of the test scenarios shown in the figures above. Despite this, the authors state that it is possible to use Wi-Fi Direct functionalities simultaneously with an infrastructure wireless network, reaching the conclusion that the operating system is creating a virtual network interface. But the same interface cannot be connected simultaneously to multiple groups. Thus, scenarios *a)* and *c)* from Figure 2.8 and scenarios *b)* and *c)* from Figure 2.9 are not feasible using simple application layer procedures.

Although it is possible to use both interfaces concurrently and the connection between the groups is successfully established, the experiments from the authors suggest that it is not possible to create a unicast communication to and from the gateway node. For scenario *b)* of Figure 2.8 the gateway node was able to receive data from both groups but was not able to send. For scenario *a)* and *d)* of Figure 2.9 the gateway node was able to communicate with node A but there was no communication with node B. The authors believe this is due to the fact that the DHCP protocol assigns the same IP address to multiple GOs, creating a routing problem, also referred by C. Casetti *et al.* in [2].

Three solutions to this problem are presented: the first is time sharing, which will allow the implementation of any scenario on the figures above. In this solution, the gateway node is alternatively connecting between groups, *i.e.* disconnecting from the current group, scanning for active devices and request to connect to the new group. In the scenarios of Figure 2.8 the gateway node acts as client in both groups, thus neither group has to be destroyed for this switch to occur. Alternatively, in the scenarios of Figure 2.9 one of the groups has to be destroyed, since the gateway node acts as owner for one group and client for the other. The main difference between the different scenarios within each figure is the protocols used to connect and disconnect to a group, *i.e.* Wi-Fi Direct, classic Wi-Fi or a hybrid combination of the two.

A different solution proposed by the authors would allow simultaneous connections between groups. This can only be achieved by using a hybrid combination of the protocol, as already discussed, so only some scenarios will be tested. The authors tested the different topologies with different network sockets, *e.g.* stream, datagram and multicast sockets. These tests showed that when combining a LC/GM, or *vice-versa*, with a multicast socket, the gateway node is able to communicate with both groups simultaneously. Although the multicast socket only encapsulates one-to-many unicast communication, underutilizing the bandwidth of both protocols.

From the authors' experiments, the gateway node is able to receive and send data over the standard Wi-Fi link, while being connected to both groups with a unicast socket. But no data can be routed with the unicast socket over the P2P link. The reason for this is that Android prioritizes standard Wi-Fi links over P2P links. So the *Hybrid* protocol is proposed, where the multicast socket acts as a control channel to change the configuration of the gateway node. The gateway node receives a control message from a GM, it then verifies which type of link it established with the group. In case of a standard Wi-Fi link, the node starts the reception of the data and disconnects from the same group after the reception is finished, creating a TCP connection with the other group to send the data. In the second case, the node is not allowed to receive data, so a notification message is dispatched and the node disconnects from both groups re-connecting with the correct configuration, *i.e.* inverting the link types.

Finally, the authors modified the source code of Android 4.4.2, altering the current implementation of Wi-Fi Direct to assign a unique IP address to each GO, mitigating the routing problem. This allows for a gateway node that uses both interfaces, and acts as GO in one group and as a legacy client in the other.

A. Shahin *et al.* propose an Efficient Multi-group formation and Communication (EMC) protocol for Wi-Fi Direct, in [4]. This protocol allows multi-group formation as the solutions presented before, only this time the protocol has some significant improvements. EMC exploits the battery conditions of the devices in the network to select the GOs of each group and enables the dynamic formation of Wi-Fi

Direct groups. With these features EMC is a very efficient protocol if battery is an essential resource, *e.g.* during an emergency period.

The authors utilized Wi-Fi Direct's service discovery feature to allow devices wishing to form a group to share information on their battery status. The algorithm will then choose the device with a richer energy reserve and elect it as the GO. The rest of the devices can then connect to the created group as GM. Some of these GMs are referred as Proxy Members (PMs) and are the GMs that link a group to another, similar to the definition of bridge and gateway nodes introduced by the other authors. PMs use their standard Wi-Fi interface to join another group, as a legacy client, forming a network topology similar to the one in Figure 2.10.

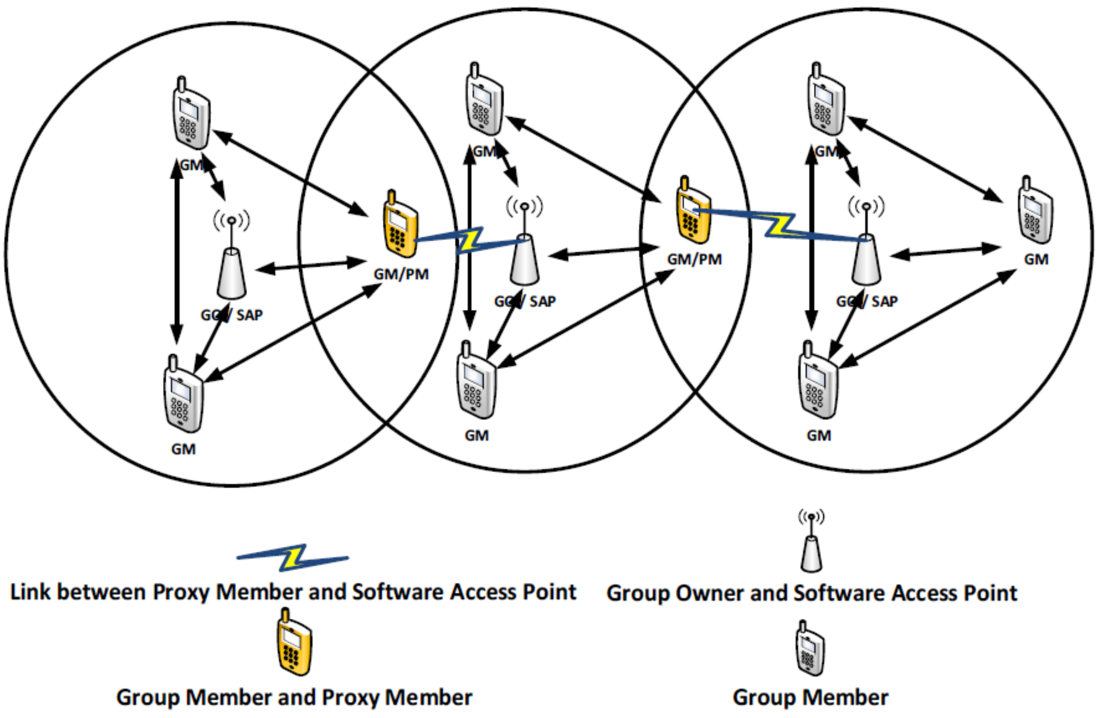

GM
GM/PM
GO / SAP
GM
GM
GM/PM
GO / SAP
GM
GM
GO / SAP
GM
GM
Link between Proxy Member and Software Access Point
Group Owner and Software Access Point
Group Member and Proxy Member
Group Member

Figure 2.10: Example of a network topology after running EMC (source: [4])

After the group is created and PM selected, the GO waits for a period of time before tearing down the group and restarting the EMC protocol. This is done to ensure that the battery drain of the GO is minimal and that groups can be established with all the devices for the maximum period of time. Upon restart, the new GO is elected and the process is repeated.

In order to overcome the limitations of stock Android already discussed, the authors decided to modify Android's source code. By doing this, multi-group bidirectional unicast communication is allowed. Also, the issue of IP addresses assignment is mitigated by giving GOs different addresses.

Finally, in order to validate the protocol the authors created a chat application, which runs autonomously without any user interaction, apart from the messages to be sent. Manual override buttons are also present in the application for users to manually control the group creation and teardown.

K. Liu *et al.* propose a new implementation of Mobile Ad hoc Network (MANET) using Wi-Fi Direct in stock Android, in [5]. It is the authors' belief that MANETs using this technology can be used in LTE offloading systems. This implementation has the following properties:

- All devices must have the same setup, *i.e.* same functionalities.

- The devices must be ready to be discovered, connected and to transmit
- The MANET is dynamic so devices must be able to join different groups in the network
- All devices are able to leave or join the network, making the MANET not depend on any device

This solution follows a different approach than the previously presented. According to the authors, in order to achieve all of the properties listed above, all devices must become GOs when there is no data transmissions, creating a topology similar as the one in Figure 2.11.

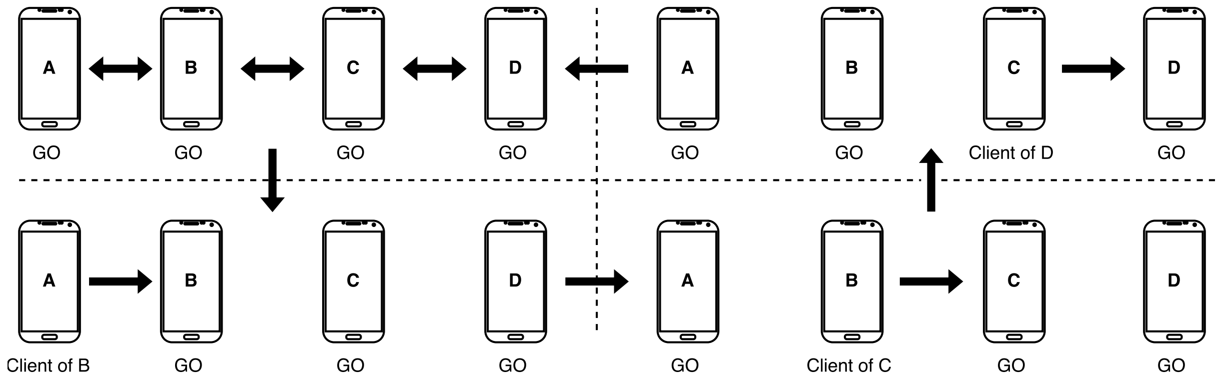


Figure 2.11: Wi-Fi Direct MANET topology (adapted from: [5])

The transmission cycle is as follows: the device with data to transmit must first remove its GO status and connect to the destination as GM, via the Wi-Fi Direct interface. Once the connection is established, the devices may communicate between them. After the transmission, the device acting as a GM disconnects from the group and becomes a GO again. This cycle is repeated whenever there is data to transmit.

This MANET topology is implemented in stock Android devices, as previously mentioned, and presents a distinct solution from the other works, since the devices in the network are constantly changing roles and groups inside the MANET. One disadvantage of this solution is that the status change of the devices are triggered by human interaction with the application, making the network somewhat dependent on human interaction.

2.2.3 Multi-Hop Routing

In the previous section some methods proposed by different authors were presented, in order to create a multi-group networks using Wi-Fi Direct. In this section the focus will be the routing algorithms implemented over some of these methods.

In [2], C. Casetti *et al.* propose a content-centric routing algorithm on top of the network topology proposed. Meaning each node knows what is the next hop to which it has to send the request for a specific content - see [9] for a detailed explanation on content-centric networks.

The authors introduce two data structures responsible for storing the information for content routing: Content Routing Tables (CRTs), providing the next hops to reach a certain content. These tables function similarly to standard IP routing tables. They store the MD5 hash of the IP address of the next hop.

There are three possible scenarios for filling the CRT:

- The simplest one where the content item is available within the group of the content requester, where the next hops of the all GMs is the IP address of the content provider.

- The second scenario is when the content is available in a different group, reachable through the group's relay node. This means the GO of the second group is connected as a legacy client to the first group and, according to the authors' scheme, all the GMs of the first group will have as next hop the group's relay node, except for the GO acting as a client. The next hop for the relay client is the IP address of the GO of the second group. Finally, the next hop of the latter is the IP address of the P2P interface of the content provider.

- The other scenario is when the content is available in a second group, reachable through the GO of the first group, which acts as a legacy client in the second group. Following the authors' scheme the next hop for all members of the first group is the IP address of the P2P interface of the GO acting as a legacy client. The GO of the first group will have as next hop the IP address of the second group's relay node. The relay client will then follow the steps from the first scenario.

The other data structure are the Pending Interest Tables (PITs), where the information about the destination of the content item is stored, they are the next hops from the reverse path of the content requests. When forwarding a content request, the nodes store the IP address of the node interface from where the request was received. With this mechanism, a "memory" of the content request path is created and utilized to then forward the content item to its requester. Upon receiving the content, the intermediate nodes forward the packet and remove the corresponding entry from the PIT. When multiple PIT entries requested the same content, the sending node replicates the packet and sends it to all the requesting devices, deleting all the corresponding entries. A content received by an intermediate node without any correspondent entry in the PIT is discarded.

The content registration, advertisement and request is done in two phases: the initial phase when a client advertises that new content is available, sending a message to the GO, which returns an Acknowledge Packet (ACK) as confirmation. The GO will then advertise within the group the availability of the new content, by sending a broadcast message to all GMs and waits confirmation from the relay node. This broadcast message is discarded at the IP layer by the legacy clients that are GOs of other groups. Thus, in order to create a multi-group advertisement, the relay client will send an advertisement to these clients and waits for the ACK.

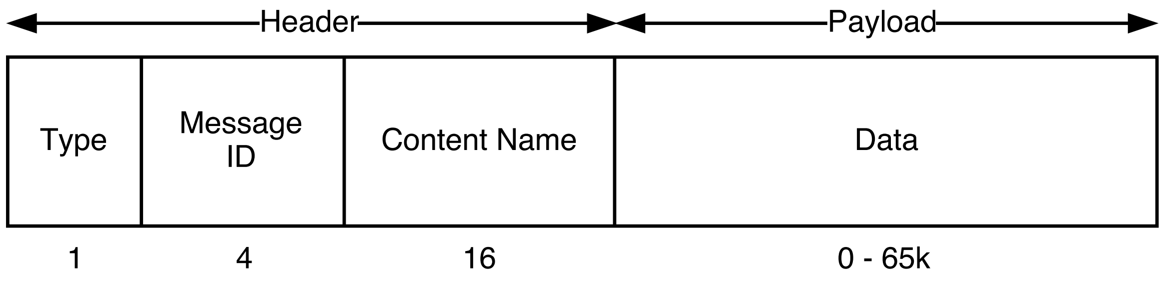


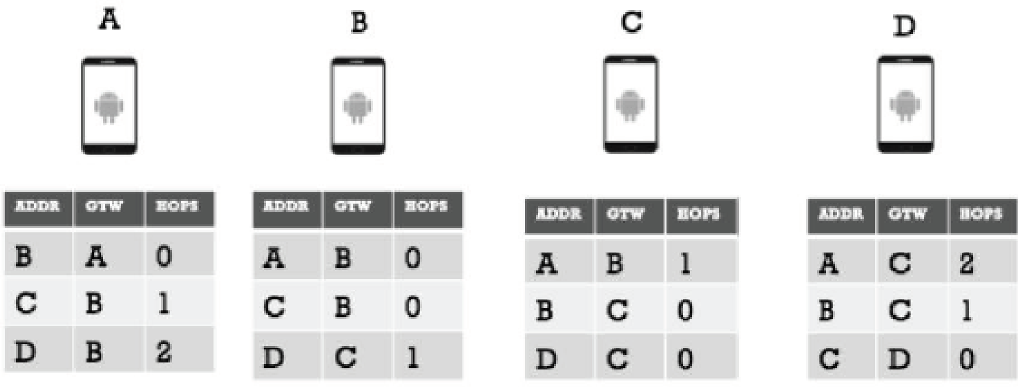
Figure 2.12: Application-layer message for content registration, advertisement, request and delivery. (adapted from: [2])

The message format can be seen in Figure 2.12, where "Type" refers to the purpose of the message, *i.e.*, Content registration, Content advertisement, Content data, Content request, Relay election or notification of GO role in another group and corresponding ACKs. "Message identifier" is used to identify what message is an ACK referring to. Content name is the MD5 hash of the content name. "Data" is the payload and can carry control or content data.

K. Liu *et. al* also provide a multi-hop routing implementation in [5]. The authors create a routing table composed by two sub-tables, one containing all peers that are directly discovered by the node and the other composed by peers that are accessible via other peers, *i.e.*, multi-hop peers.

The first sub-table is created with a simple broadcast of the peer discovery signal and it will receive the responses from the devices in the vicinity, since in the authors' topology all devices are GOs if no data is being transmitted. The response messages should contain the MAC address of the destination devices, the gateway to reach the devices, in this case the node itself, since all nodes are neighbors and the number of hops to reach them, in this case zero.

After the first sub-table is complete all the nodes in the network have knowledge on who are their immediate peers. The nodes will then exchange routing table information between themselves to get a knowledge of all the multi-hop peers that may exist in the network. The second sub-table is filled proactively when nodes receive routing messages. The message-processing scheme is as follows: the device receives a message and checks its destination. If the destination MAC is not the device's MAC the device will look at its routing table and verify if an entry exists with that destination and where should the message be sent to, incrementing the number of hops, otherwise there is no route to the destination. If the destination MAC is the device's MAC, it will check the message type and infer if it is a routing message. If so, the device's routing table will be updated with the MAC address of the source of the message, the MAC address from where the node received the message, *i.e.*, the gateway, and the number of hops necessary to reach the source.



A

ADDR	GTW	HOPS
B	A	0
C	B	1
D	B	2

B

ADDR	GTW	HOPS
A	B	0
C	B	0
D	C	1

C

ADDR	GTW	HOPS
A	B	1
B	C	0
D	C	0

D

ADDR	GTW	HOPS
A	C	2
B	C	1
C	D	0

Figure 2.13: Wi-Fi P2P MANET routing table. (source: [5])

In Figure 2.13 it is possible to see a complete table for four different nodes. Each node is now capable of sending a packet to its destination, without having to use a broadcast mechanism.

Two different routing mechanisms have been introduced in this section. Each mechanism is supported by a specific network scheme. This is the main problem of routing in Wi-Fi Direct, since the routing algorithm is developed at the application layer thus being always dependent on implementation, making most of the algorithms not compatible among themselves. This problem also brings some limitations in the number of users that will integrate a specific network with a specific routing scheme.

2.3 Ad hoc Networking Applications in Android

In this section some Android applications that make use of ad hoc networking will be presented, as well as the toolkits on which their development was based, if applicable. It is important in the scope of this work to know what are the offers on today's market on the ad hoc networking, to understand what has been done and what remains to be done and improved. These applications specifically make use of Wi-Fi Direct technology to establish communications or discover peers. It is worth mention that many other applications exist that create ad hoc networks but we are interested in the ones that use this technology in particular, since it will be the base to this work.

2.3.1 FireChat

FireChat is a cross-platform application that allows users to live chat with each other without access to the cellular or infrastructured network. It uses Bluetooth, Wi-Fi Direct or Apple Multipeer Connectivity to provide peer communication between devices.

FireChat is useful when there is a large concentration of users creating traffic that can, possibly, create some congestion in the infrastructured network. By using FireChat users are able to communicate between themselves with minimum delay, offloading the network and being independent on any service provider.

Users can create or join groups, *a.k.a.* chat rooms, to communicate with other users in the same group. Messages can be sent to multiple or single destinations, also messages can be either private or public: in the first case only the selected receiver/receivers will be able to see the message. In the second case the message is broadcasted to all GMs that form the chat room.

When a user sends a private message to another device, a private group is automatically created. However, senders might want to reach devices outside the groups where they are inserted, in that case the message is routed across the group as a private message to a GM that is connected to the infrastructure and can send the message to the destination, that will receive it upon establishing Internet connection.

FireChat also allows for other features, such as blocking the communication with specific users, sending of photos, following other users' FireChat activity, *etc.*.

This application is built with MeshKit, a Software Developer Kit (SDK) module with methods allowing for P2P mesh connectivity within devices using this module. It is built over Bluetooth, BLE, Wi-Fi Direct, ANT and other wireless protocols.

It works on three different mechanisms: cloud to mesh where devices connected to the Internet receive data from this link and share it with the other devices within its group, using one of the technologies mentioned above. The packets hop from device to device until it reaches the destination. This mechanism uses broadcast to diffuse the packets through the network.

The second mechanism is mesh to cloud where devices connected to the Internet forward data from devices without Internet connection but with a P2P connection established with the gateway devices, *i.e.* the ones connected to the Internet.

Finally, the peer-to-peer mechanism, where the Internet is not accessed and devices transmit data between themselves using only the P2P link. Data is routed within the groups to reach its destination. Multicast or unicast transmissions can be used.

2.3.2 Uepaa!

Uepaa! is one of the fastest growing P2P software companies, alongside with OpenGarden, responsible for FireChat. Uepaa! created Uepaa! Safety App, an application that aims to provide a security service for users who spend time outdoors and where cellular or Internet connection might not be available, such as mountaineers.

The app allows users to connect directly to the companies emergency call center even if no mobile network coverage is available in the area. This is achieved by forwarding the rescue message to users in the area using Uepaa!'s application. Similarly to FireChat's peer-to-cloud mechanism, in Ueppa! Safety App the message is routed through devices in the are until a device has a connection with either a mobile network or the Internet, and the rescue message, with the user's location is then sent to Uepaa!'s call center where the user can get assistance.

Uepaa! also developed the p2pkit a cross-platform SDK that allows for P2P transmission of beacons, with configurable information. P2pKit also focuses on the battery consumption, since it can be used in battery sensitive applications such as Uepaa! Safety App. It provides P2P communication with all the associated processes, such as discovery and formation of groups. Range estimation on who are the closest peers and how close are they is also included in p2pkit modules, which can be useful to efficiently manage the battery of the device by sending only the beacons to the closest peers.

P2pkit is the base for different applications. Nearby devices using p2pkit are notified when a proximity event takes place and, although the principle is the same, applications are tailored to respond differently to certain types of events, *e.g.* Uepaa! Safety App responds to the event of receiving a beacon by checking the device's Internet or mobile signal and tries to forward the message to the call center. other applications might trigger other responses, such as notifications to the current user, not forwarding the data but storing it and displaying it in the device.

2.3.3 Murmur

Murmur is an open-source, anonymous messaging application. Users can communicate with one another without Internet or mobile connection, similarly to the applications described above. However, Murmur uses Wi-Fi Direct and Bluetooth technologies to establish communications between the devices, creating a Delay Tolerant Network (DTN) in which the P2P transmission of messages occurs.

Much like every other P2P network, the more users Murmur has, the faster transmission of data is. Messages are broadcasted over the network and if no device is in the vicinities of the sender, the message is queued for posterior transmission, thus the delay tolerant network designation. Murmur allows users to post messages to the network, a system to upvote, store, share and search for posts.

The discovery and connection between devices is done using both Wi-Fi Direct and Bluetooth: each user advertises it is running Murmur via the Wi-Fi Direct interface with a specific name, "MURMUR:Bluetooth MAC", constantly searching for peers using a similar device name, *i.e.*, starting by "MURMUR". If a discovery is made, the application filters the Bluetooth MAC address from the device's name and creates a Bluetooth connection, exchanging the message. The actual exchange starts with a handshake where each device sends its contact list and the amount of shared contacts between the two is calculated. Secondly, the number of expected messages to be exchanged is sent and, upon agreement, each device transmits until that number is reached. Finally, when the exchange is complete, the received messages are saved to a local database and the devices set a backoff time before establishing other connections, avoiding redundant exchanges and unnecessary battery consumption.

Chapter 3

Developed Peer-to-Peer Communication Prototype

In this chapter the implementation of the developed peer-to-peer communication prototype will be analysed from the decisions

This chapter will begin with a brief summary of the steps taken to reach the developed application's features, functionalities and architecture. Which obstacles led to this solution and which were the fundamental drivers to decide upon this development and architecture to solve the problems previously stated. After the overall application's objectives and high-level architecture is described, a section on the lower-level architecture and components of the application will be presented, also containing flowcharts and diagrams on its normal functioning. A summary on the limitations and future work of this application will be given, as well as some possible solutions to its current limitations, outside the scope of this work. Finally, a conclusion on the difficulties and drawbacks of the development process and overall project appreciation will be given, along with some final thoughts on the work process and methods used.

In section 2.2, developed work on Android applications using Wi-Fi Direct has a mean of communication was introduced. Having the guidelines of these works in mind, the development of this application started. The main structure would always be: the application will have a routing algorithm that controls the destinations of the packets in each hop. A communication technology, *e.g.* Bluetooth or Wi-Fi Direct, handler for both discovering nearby devices and establishing communication sockets with peers. A set of functions capable of analyzing incoming messages and deciding which is the next step to perform, in order to complete the requests/advertises. Finally, a user interface where the incoming messages can be seen and analysed (for debug purposes), also providing a text area for the user to enter the requested data.

The choosing of what would be the transferred data between devices was one of the major steps of this work. Many ideas could be pursued and each one of them could be a great work to develop. The main contenders were text messages, beacon messages, geolocation messages and web pages. After some research was made, see section 2.3, some of these contender could be eliminated, due to the existence of works already revolving around them. The main topic was text messages with many of the works developed focusing on recreating popular applications, such as Messenger and WhatsApp, using a peer-to-peer network to route the messages to their destinations. Beacon and geolocation messages were also not a completely innovative choice, since applications such as Uepaa!, see subsection 2.3.2, are already implementing systems similar to the one being proposed. Given this reasoning, the most innovative choice would be to transfer web pages between devices, creating a multi-hop hot spot in a peer-to-peer network.

After being established that web pages were the data to be requested, the next step was to choose the communication technology to be used, followed by the routing algorithm that would manage the next hops to take. The first and most obvious choice would be to use Wi-Fi Direct in both advertising and communication between devices, since this technology offers the best features to transfer files around 1Kbyte, in both range and data rate of transmission. However, during the development it proved to be impossible to continue with this approach, as Wi-Fi Direct's current Android implementation does not allow for devices to transfer files without user consent.

Given this drawback a shift to Bluetooth was made, and a hybrid version of the application was created, where the advertisement would be done via Wi-Fi Direct and the actual transfer of the web pages would be made via Bluetooth. This method also proved to be infeasible, due to security issues, since the the devices would have to display their Bluetooth MAC addresses in their Bluetooth name, making them much more vulnerable to possible attacks.

So, finally, the application was developed using Bluetooth for both advertisement of the devices and data transfer, although this is not the best technology for communication in a peer-to-peer network, it is the only one that, currently, meets all the requirements for this work. In the end of this chapter a brief discussion on the changes that need to be made to Wi-Fi Direct's implementation in Android will be presented, and why developers could benefit from these changes.

The last step would be to create/modify a routing algorithm to control the destination of each incoming message. The requirements are simple, the algorithm simply needs to save the next hop of the shortest path leading to a device with Internet connection.

AODV is a known routing scheme used in ad hoc mobile networks, precisely what this application will establish with the devices. However, it establishes dynamic paths, *i.e.* only when a device wants to retrieve a web page does this protocol searches for a path in which to send the packet, see [10] for the full specification of this scheme. This scheme is not the best for the applications reliability, since the requesting device should know before hand if it is capable of reaching the Internet, otherwise users will experience long periods of path requesting and advertising every time they request a web page.

Destination-Sequenced Distance Vector (DSDV) is also known for its use in ad hoc mobile networks. This protocol uses a routing table where it stores in each entry: the destination, next hop, number of hops to reach the destination and a sequence number - avoiding routing loops. The devices exchange full routing tables, creating a network where every device has total knowledge of the topology, being a powerful advantage. However, DSDV requires the exchange of routing tables and their regular updates, using unnecessary bandwidth, see [11] for more information on DSDV.

Since the routing of this work does not have a specific destination as a goal, in other words, device A does not want to transmit to B, it only wants to reach a node with network access with the minimum amount of hops, so A does not have knowledge on where its message will be sent after the immediate next hop. Thus, in order to accommodate such requirements some modifications were made to DSDV. Firstly, to reduce the bandwidth used by DSDV, this adaptation will simply exchange the best estimate of the current device, only including the number of hops and the device's MAC address. Secondly, instead of a periodic exchange of tables, devices will update their tables every time a new advertise message is received and they only exchange routing information when a new best path is received, with the contents described above.

In the next section a more detailed and technical description of the work will be presented, along with the explanation of the application's code and some diagrams to give a more comprehensive overview of the developed work.

3.1 Architecture

This section is divided into four subsections: Bluetooth connections, where a detailed explanation on the Bluetooth Service and application Handler used to manage the Bluetooth connections and data transfer between devices, will be presented. An overview of the discovery and advertising code functions, as well as a description on how this was implemented and how the discovery and advertising data is analysed and stored. The same analysis for the exchange of web pages between the devices, from the request to the received response. Finally, a demonstration of the execution of the application with a linear topology with three devices, where one is connected to the Internet.

3.1.1 Bluetooth connections

Since the communication technology is Bluetooth the application must have a thread[1] that handles the listening and acceptance of connections and the data transfer between devices. This corresponds to the *BluetoothService.java* class, from the application. It creates and listens to insecure communication sockets to allow for devices to exchange data without user approval.

The service has three different threads within it: a thread for the acceptance of incoming communications - *AcceptThread*, for the initial exchange of vital information before the connection - *ConnectThread* - and a thread for the actual exchange of data between devices - *ConnectedThread*. The connection of devices goes through each of this stage, allowing the main thread to get information on what is the connection status of the device at a given time.

There were two different approaches to the connections of devices: either the devices stayed connected as long as possible until a new connection was requested or the devices stayed connected for the minimal amount of time for data to be transferred. The second approached was used, since it is the one that drains less amount of battery from the devices.

This class, seen in appendix A.4, starts with the definition of several variables that will be the core of the service. The variables *NAME_INSECURE* and *MY_UUID_INSECURE* are set so that Bluetooth communications is only accepted if both devices have matching ”credentials”, they provide a minimal amount of security to prevent against possible attacks. *MY_UUID_INSECURE* is a unique identifier, generated randomly that represents this application, no other application should have the same identifier.

In the next section, variables are defined to represent the threads, discussed above, the Bluetooth adapter of the device and the handler, creating the bridge between the main activity and the service. Two other variables are defined, one for retrieving the status of the connection and other for assessing if the device is ready to receive the desired file. We will discuss this variables later on this subsection.

Finally, a set of four constants is presented, giving a textual representation on the possible connection states: *STATE_NONE*, *STATE_LISTENING*, *STATE_CONNECTING* and *STATE_CONNECTED*, where each one represents the current thread of the connection and *STATE_NONE* represents the nonexistence of one of these threads.

I will now analyze the code functions deemed most important, as some are trivial and do not require mention. The constructor[2] matches the Bluetooth adapter of the main activity to the Bluetooth adapter that this service will use and the handler with the one created in the main activity, in order to pass information from the service to the activity, *e.g.* messages sent/received or connection states.

[1]Thread is a sequence of instructions managed independently, usually by a scheduler from the operating system, see [12] for more documentation on threads.

[2]Constructor is invoked in a class to create objects, based on the class blueprint, see [13] for more documentation on constructors.

3.1.1.1 Listening

The *start()* function is, probably, the most important function of this service. It is called every time the service is initialized, creating the threads necessary for the well functioning of the service. We can see that before creating an instance of an *AcceptThread*, this function makes sure any running threads are shut down, avoiding any conflicts between connections. As said before, this application works based on short communication times, so the process of restarting the service in order to close ongoing connections is called more often than it would be usual in long duration communications. Thus, it is essential that there are no memory leaks or communication sockets left open. It also sets the communication status to *STATE_LISTEN*.

Jumping now to the *AcceptThread* class, we can see it extends another class, the *Thread* Java class, previously mentioned. This class contains a constructor and two methods: the constructor simply creates a communication socket with which the device listens to incoming communications. The *run()* method handles the listening, it has a cycle whose purpose is to block the thread until a connection is received or an error occurs. Upon successfully accepting a connection, the method assesses the status of the connection and decides upon it. If everything goes as planned, the status should be *STATE_CONNECTING* and a call to the *connected()* function is performed, indicating that we will now handle the accepted communication and start receiving/transferring data. The second method is simply a closing of the established socket, to avoid memory leaks. It is called whenever we want to force stop the thread.

3.1.1.2 Connecting

In the side of the communication requester, the *connect()* method is called, in order to try and establish a successful socket between both devices. It has a similar structure as the *start()* method, since it cancels any ongoing thread and simply creates an instance of the *ConnectThread* class, altering the communication status to *STATE_CONNECTING*.

The *ConnectThread* class is also similar to the *AcceptThread*. It has a constructor, and two methods. The constructor receives a BluetoothDevice as parameter and tries to establish a communication socket with it. The *run()* method starts by shutting down any device discovery process as it would slow down the connection time. It then waits on the acceptance of the requested device, in case of failure, the *connectionFailed()* method is called, where a notification is sent to the main activity, in order to notify the application that the connection was not successful, the service is restarted. In case of success this thread is shut down and the *connected()* method is called, with the respective socket and requested device. The *cancel()* method serves the same purpose as in the *AcceptThread* class.

3.1.1.3 Connected

The *connected()* method is called in both requested and requester devices, it is called upon concluding the preliminary works of the connection. It starts by canceling any ongoing threads, again in order to avoid conflicts between the device's Bluetooth communications. It then creates an instance of the *ConnectedThread* and sends back to the main activity the name of the device with which the connection was just established. Finally, it sets the communication status to *STATE_CONNECTED*.

The *ConnectedThread* class follows the same pattern as the other two thread classes. It has a constructor but it has four methods: the constructor receives the established communication socket and creates an *InputStream*[3] and *OutputStream*[4], in order to read and write bytes to the socket shared with

[3]An InputStream is an abstract object created to read bytes from a source of data, see [14] for full documentation.

[4]An OutputStream is an abstract object created to writes bytes to a source, *e.g.*, a file or a socket, see [15] for full documentation.

the other device, respectively. The *run()* method has a cycle that keeps reading from the socket to a buffer while the connection is not closed. Since we send two different types of data from device to device, strings and web pages, logic was created in order to distinguish what is the device expected to receive at any given time. The variable *fileReady*, as mentioned previously, is used to assess if the device is ready for the reception of a web page, by checking the status of this variable the application infers if it should treat the received data as a string or a file. Should the variable return false, meaning the device is not ready for file reception, the service will send the received string to the main activity, via the handler, so that the message can be analysed and the next step can be decided, otherwise, the device is ready to receive the web page. In order to overcome the size limitation of the socket buffer, approximately 1Kb, the web pages are disassembled in smaller chunks of 990 bytes. A auxiliary buffer is created to store the received chunks, orderly, and, by judging the size of the chunks, the receiver can assess if it is receiving the last chunk, *i.e.*, by comparing the received number of bytes and 990. After the reassembly of the chunks is done, the bytes are sent to the main activity to be processed. In case the connection is suddenly broken, the *connectionLost()* method is called, notifying the main activity of this event and restarting the service.

The next two methods, *write()* and *writeFile()* are called by the sender device to write bytes to the *OutputStream*. It is important to make the differentiation between writing a simple string and a file because of the above mentioned disassemble and reassemble of the file chunks, which is not necessary when working with strings. So if the message to send is a string, containing information on an advertise, request, response or failure, it will be handled by the *write()* method, simply writing the bytes to the socket and notifying the main activity of that action. However, if the data to be sent is a file, it will have to be divided into chunks of 990 bytes as mentioned before, also, since the service needs to notify the stream on the exact amount of bytes to be transferred, extra logic is needed to get the number of bytes that the last chunk of the file contains. Both these methods can be called asynchronously, *i.e.*, by user request, via two functions of the service, *write()* and *writeFile()*, whose functionality is simply to call the *ConnectedThread* class's functions, respectively.

Finally, the *cancel()* method serves a similar purpose to the previous classes, it forces the close of the socket and nullifies the thread instance.

3.1.2 Discovery and advertising

Now that the basic concepts on how the communication between devices are implemented are explained, it is possible to dig deeper into the intrinsics of the application itself, and what is the logic behind the routing of web pages.

The first thing in order for each device to know what is the next hop of a certain request will be to fill the routing tables. This process is done by a series of discoveries[5] and advertises[6]. Upon starting the main activity each device will advertise its best estimate to reach the Internet. This advertise will be received by peers and analysed, in case the estimate of relaying the requests through that node improves the current estimate, the receiving peer will begin another discovery and advertising process with the newly discovered path. Otherwise, the peer will add or update an entry in the routing table, regarding the advertiser.

In figure 3.1 the format of an advertise message is shown. The type is a feature to common all messages exchanged in this application, it servers the purpose of identifying which type of message is being received. It can take four different values: *ADV*, *RQT*, *RSP* and *FAIL* for an advertise, request, response and fail message, respectively, in this case it will take the value *ADV*. The delimiters of the

[5]Discovery is the act of finding nearby devices with Bluetooth on.
[6]Advertise is the act of notifying peer devices of the cost of relaying a request to the device issuing the advertisement.

Type	Bluetooth MAC Identifier	Estimate number of hops

Figure 3.1: Advertising message format

different message parts - vertical lines in the figure - are represented by the dot comma character (;), in order to be able to separate the different parts of the message. The Bluetooth MAC identifier is always the MAC address of the Bluetooth adapter of the sender, it is then used by the receiver to populate its routing table. Finally, the estimate number of hops is the smallest number of hops the sender needs to be able to reach the Internet plus one, corresponding to the hop from the receiver to the sender.

Each device has two different routing tables: one that is accountable for routing the requests to the final device and another one that is accountable for routing the responses to the original sender and it will be analysed in the next subsection. For simplicity the first routing table will remain with this name, while the second will be referred to as response table, as it handles the routing of responses.

Next hop's MAC	**Number of hops**
Own MAC	0 or 16
Device A's MAC	Estimate through A
Device B's MAC	Estimate through B
...	...
Device Z's MAC	Estimate through Z

Table 3.1: Routing table example and format

In table 3.1, an example of a routing table is presented, where the first entry is always populated with the device's own MAC address and its estimate, which is either 0 or 16[7], meaning the device has an Internet connection or not, respectively. Every time a device receives an advertise message, it populates this table, either by adding a new row or updating an existing one, with the information contained in the message, see figure 3.1. When the routing table is done being populated, each node knows its immediate peers and the best estimate through each peer, giving it a full knowledge on its vicinity, not on the overall network.

In the code, this process was implemented, mainly, in *RoutingApp.java* class and in the main activity, or *BtActivity.java*. Let's start by understanding how the routing tables are being created and, posteriorly, it will be explained how and when they are populated. It is important to note that *RoutingApp* class extends the Java class *Application*, meaning that, with the exception of the private methods and variables, every part of this code can be accessed at any time in the application, in contrast with other activities that are only accessible when active, such as *BtActivity* and *InitActivity*.

In appendix A.3 it is possible to see the definition of two tables *routeTable* and *rspTable*, corresponding to the routing table and response table, respectively. In this subsection's scope a closer look will

[7] 16 was chosen to be the representation of infinity or inability to reach a destination, since it has been represented by this number in various protocols, for instance in Routing Information Protocol (RIP), see [16]

be taken on the first table. It is represented as an instance of the Java interface Map[8] where the keys correspond to the MAC address of the next hop, the string, and the keys to the respective estimated number of hops, the integer. So it maps a string to an integer, which is the same to say that given a string the table contains, it will return an integer.

There are three important features that the application needs from this table: to get the absolute minimum value, meaning retrieving the lowest possible value for any given key, in order to retrieve the device's best path. To get the corresponding key to the minimum value, in order to retrieve the receiver to whom this device should send the message. Finally, to update or add a row with a new key-value pair, in order to add newly received advertising information.

The first feature is completed by method *getMinHop()*. This function starts by creating a variable *minHop*, that will be assigned to the minimum value of the table. It then makes use of the preexisting Java method *Collections.min()*, that returns the minimum value from a set of values, and assigns the variable *minHop* to this value. To get the desired value, it is given the set of values of the routing table, by calling *routeTable.values()*. Should this succeed, the function will return the desired value, otherwise it will return 16, for the reasons explained above.

Having the minimum value from the routing table, it is now necessary to retrieve the MAC address from the device that provides this estimate. This is done by function *getKeyFromValue()*, that receives the routing table itself and the desired value. It iterates through every key and, in every iteration, it compares the value of the current key to the expected value, since it already knows the minimum value it is useless to compare the values of the different keys. If, in a given iteration, it finds a key that maps to the desired value, this key will be returning, corresponding to MAC address of a device that gives the smaller estimated number of hops, otherwise it returns null. It is important to note that several devices may provide the shortest path, but the application will only chose the one that was found first, since they all provide the same estimate.

Finally, the addition and update of rows within the table is done by the method *updateRouteTable()*, that takes as arguments the newly received advertising message. It begins by fragmenting the message in its parts, knowing that, according to figure 3.1, this message will contain the type in the first position[9], the sender's MAC address in the second position and the estimated number of hops in the third, using dot commas as delimiters. Using Java method *String.split()*, the function separates the string by dot commas, creating an array of smaller strings, it then matches the destination to the second position, represented by variable *dest*, and the number of hops to the third position, represented by variable *hops*. Having this information, the Java method *Map.put()* is used, it inserts the given key-value pair into the existing table, in case this key already maps to a value, the value will be overwritten with the new one, otherwise it will simply insert the new pair as a new row, see [17] for full documentation on this method.

Note that, in case of a table update, the application does not compare values, *i.e.*, it does not check if the new estimate is better than the old one, since the sender device could have lost the communication with the best route and thus it needs to advertise a worse path. If this check was made, the receiver could be mislead into sending requests through this device, thinking it would provide the better path.

3.1.2.1 Discovering peers

Now that it is clear how the intern process of modifying the routing tables works, it is possible to discuss where and when these methods are being called. Shifting the focus to *BtActivity.java*, in appendix

[8]In Java, a Map is an object that relates keys to values. To a single key corresponds a single value, thus not creating duplicate entries, see [17] for full documentation.

[9]In Java, the first position of an array is given by the number 0, the second by the number 1 and so forth, see [18] for more information.

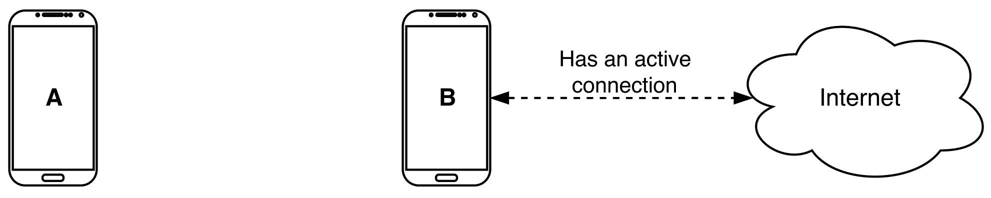
A.2, there are some variable definitions indispensable for the process of advertising and discovery. Starting by the *peers ArrayList*, it is a list of *BluetoothDevices* each corresponding to a peer found in the discovery process. This list is then used to retrieve information from these devices, in order to successfully establish communication sockets with them. The *mBluetoothAdapter* is no more than the device's Bluetooth adapter, without which there would be no Bluetooth communications. Finally, the variable *discoveryFinished* is a boolean value that serves the purpose of identifying if the discovery process is finished, assuring that this is not carried away for longer than it was intended to be, using processing power and damaging the performance of the rest of the application.

Method *onStart()* is used to denote the starting of the activity. It starts by verifying the necessary steps to ensure Bluetooth is properly set. Some configurations relative to an *WebView* object, that will be discussed later, are performed. The *ensureDiscoverable()* method is called, in which the application checks the Bluetooth visibility of this and, in case it is not discoverable, it sets the visibility to 0, that corresponds to "always visible", see [19].

Once the Bluetooth is set up, the initial update of the routing table is performed, a quick network check indicates if the device has an active Internet connection, by calling the method *getHasNet()*, from *RoutingApp.java* class, in appendix A.3. Should the device return positive, it will add a new row with its own MAC address, retrieved by method *getOwnMAC()* and 0 hops, since it's connection is immediate. In case the Internet check returns negative, meaning the device is currently unable to reach it, the same process is performed, however instead of 0 hops, the estimate will be of 16 hops, for reasons already explained.

The device is now finished with the first steps of assessing its position in the network and filling its routing table accordingly. After this logic is accomplished the function *doDiscovery()* is called, it sets the variable *discoveryFinished* to false, indicating the discovery is not yet finished and commands the Bluetooth adapter of the device to initiate the discovery process.

Figure 3.2 demonstrates how two devices, one without an Internet connection (left) and one with an active Internet connection (right). At this point both devices should have exactly one entry at their routing tables, since they have not communicated with any other device but they have established their position in the network, *i.e.*, if they have an Internet connection. Device A is unable to reach the Internet, so it adds to the routing table an entry with its own MAC address and an estimate of 16 hops. Device B, on the other hand, is able to reach the Internet, thus having an entry with its own MAC address and an estimate of 0 hops.



Next Hop's MAC	Number of Hops
Own MAC	16

Next Hop's MAC	Number of Hops
Own MAC	0

Figure 3.2: Example 1: State of the two devices after the initial step is over

Jumping now to the definition of the *BroadcastReceiver mReceiver*, responsible for the management of the Bluetooth discovery process and peers found, we can see that it is divided into two main sections: one for the management of a peer found and other for the management of the finished discovery. In the first segment, whenever a device is found, a new *BluetoothDevice* object is created, to designate the found peer, this peers is the added to the list *peers*, unless this device is already part of this list. In the second segment, corresponding to the finish of the discovery process, the receiver checks if the variable *discoveryFinished* has been toggled on, meaning it's the first time the device has received an *ACTION_DISCOVERY_FINISHED* action, which is self-explanatory, so the discovery should be canceled, by *cancelDiscovery()* and the variable *discoveryFinished* needs to be set to true.

In figure 3.3 a fluxogram of the discovery process is presented. By following the figure it is possible to understand the flow of the code and the hierarchy of instructions. It is a visual representation of the code running. Note that it ends with the instruction *advertisePeers()* that will be explained shortly.

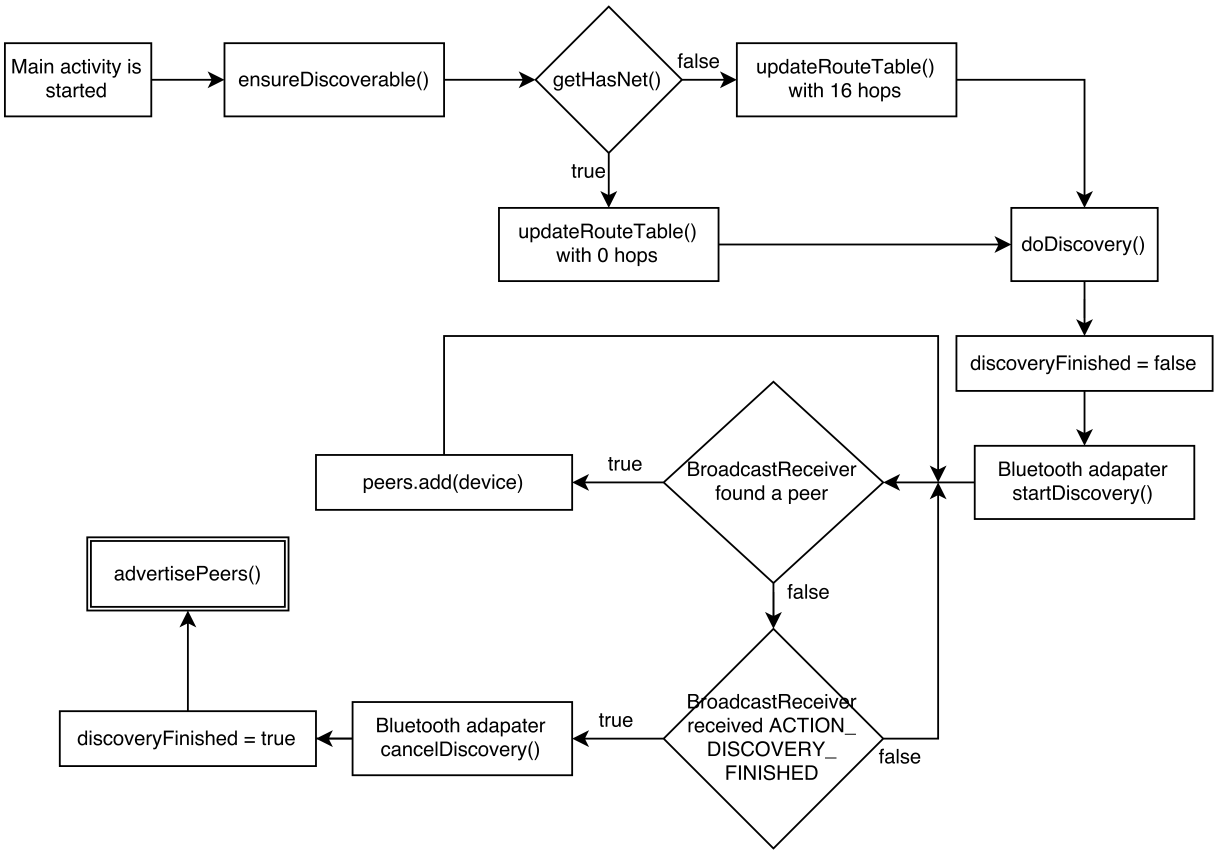

Main activity is started
ensureDiscoverable()
getHasNet()
false
updateRouteTable() with 16 hops
true
updateRouteTable() with 0 hops
doDiscovery()
discoveryFinished = false
Bluetooth adapater startDiscovery()
BroadcastReceiver found a peer
true
peers.add(device)
false
BroadcastReceiver received ACTION_ DISCOVERY_ FINISHED
false
true
Bluetooth adapater cancelDiscovery()
discoveryFinished = true
advertisePeers()

Figure 3.3: Fluxogram of the discovery process

3.1.2.2 Sending an advertising message

The device has now found its peers and it can start the advertising process, done by *advertisePeers()*. The method starts by iterating through each peer contained in the list *peers*, a first check is performed assessing if the device is a smart phone or a different type of device, a sensor or a headset, for instance, for this the *DeviceClass* is compared with 524, which is the value for smart phones, see [20]. In case the peer is a smart phone, a quick check to the name is performed, in order to see if it contains a dot comma at the beginning of the name. This is simply a test measure and can be deactivated, it is only to reduce the number of connections to the ones where the other device is, effectively, using the application.

After the necessary checks are processed, if the peer device is eligible for connection, the *Bluetooth-Service* method *connect* is called, see 3.1.1.2. To ensure the message is not sent before the connection is established, a small loop was created. This loop will be seen throughout the application, since it proved to be an efficient way to ensure the connection was established before the message was sent. It

”blocks” the method until the Bluetooth state is *STATE_CONNECTED*, see 3.1.1.3, inside the cycle there is a verification if the state does not change to *STATE_LISTEN*, as this would mean the device is not attempting to establish a connection anymore and it is simply listening for incoming requests. In case this is verified, the *BluetoothService* is restarted and the application assumes the connection has failed.

If the method breaks the cycle normally, meaning the connection was established successfully, the function *getMinHop()* is called returning the value of the minimum estimate of the device and 1 is added, symbolizing the hop this message will take from sender to receiver. The message is then sent, with the format seen in figure 3.1, where the device’s MAC is retrieved by method *getOwnMAC()*.

Finally, the method enters another cycle, this time for ensuring this device does not try to advertise to a new peer whilst it is sending a message to the previous one. For that, the *initTime* variable is set to the current time and the method is ”blocked” while the Bluetooth status is not *STATE_LISTEN*, see 3.1.1.1. In case the method is ”blocked” for more than 5 seconds, the application assumes something went wrong proceeding to break the cycle and continues the program normally. Usually, the cycle should be over way before the 5 seconds mark. Once the cycle is over or broken, the method iterates to the next peer and proceeds to do the same tasks.

In figure 3.4 the fluxogram of the advertising process is presented. It shows how the *advertisePeers()* method works in a simple way. In the end of this method’s execution the application’s *BluetoothService* keeps listening for incoming connections of its surroundings.

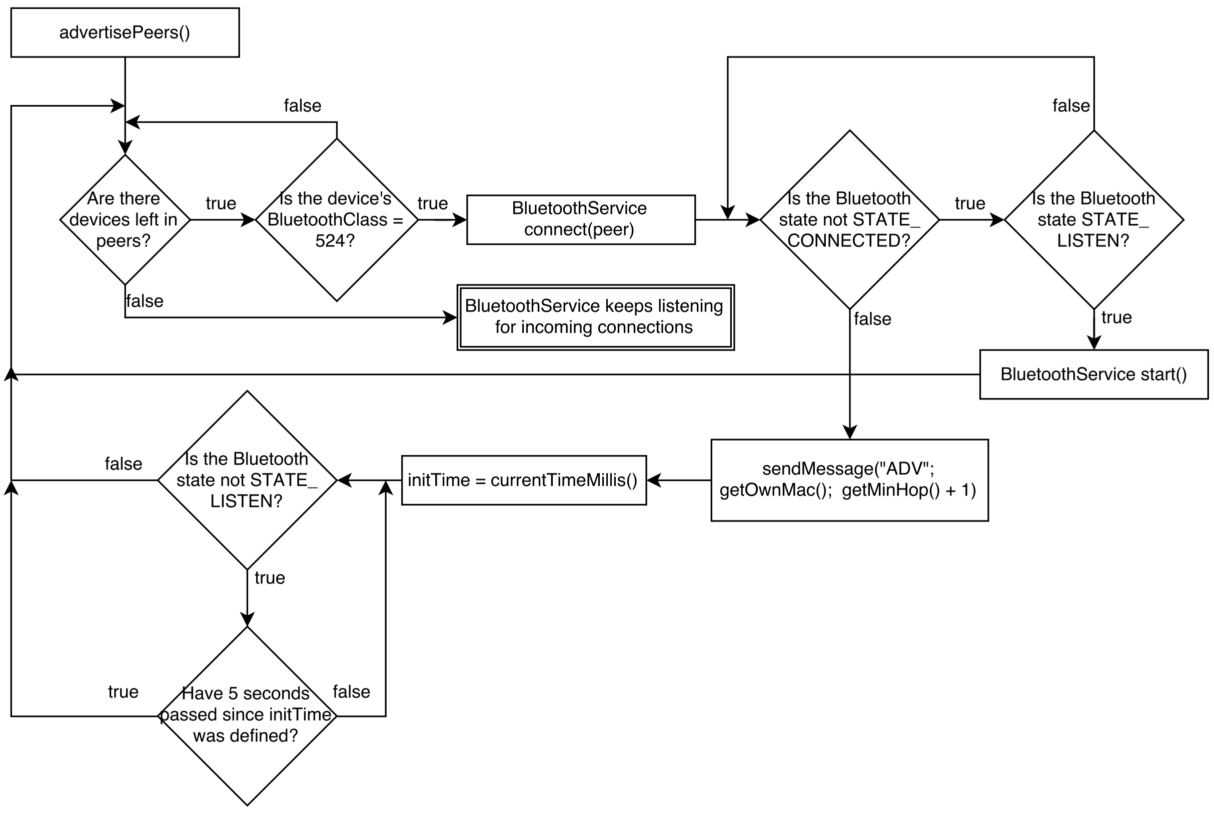

advertisePeers()
false
false
Are there devices left in peers?
true
Is the device's BluetoothClass = 524?
true
BluetoothService connect(peer)
Is the Bluetooth state not STATE_CONNECTED?
true
Is the Bluetooth state STATE_LISTEN?
false
BluetoothService keeps listening for incoming connections
false
true
BluetoothService start()
false
Is the Bluetooth state not STATE_LISTEN?
initTime = currentTimeMillis()
sendMessage("ADV"; getOwnMac(); getMinHop() + 1)
true
true
Have 5 seconds passed since initTime was defined?
false

Figure 3.4: Fluxogram of the advertising process from the point of a sender

3.1.2.3 Receiving an advertising message

It is now covered how the transmitter of an advertise message operates, however, to fully understand the advertising process it is necessary to analyse the receiver device and how it handles this message.

So, on the receiver device, after the reception of the advertising message, the handler receives the transferred message bytes from the *BluetoothService*. To understand how this is performed the *Handler mHandler* must be explained first.

The method *handleMessage()* is overwritten, in order to control what is done with the received *Message*, see [21] for full documentation on this Android class. There are seven possible *Message* types: *MESSAGE_STATE_CHANGE* used to notify of a Bluetooth state change, *MESSAGE_WRITE* received when this device has sent a message, *MESSAGE_READ* to notify that this device has just received a message, *MESSAGE_DEVICE_NAME* received when there is a new connection, gives the name of the connected device, *MESSAGE_TOAST* for when the user needs to be notified on something regarding the *BluetoothService*, *FILE_READ* to notify of the reception of a web page by the device and *FILE_WRITE* for debug purposes, received when this device sends a web page.

For the receiver's advertising process the focus will be on the *MESSAGE_READ Message* type. If the *Message* is identified as a *MESSAGE_READ* the *Handler* will construct a *String* from the received bytes. From this *String*, the device is now able to extract what type of message it is, *i.e.*, an *ADV*, a *RQT*, a *RSP* or a *FAIL*. In case the message is not a response the *BluetoothService* is restarted and will listen to incoming connections. However, if the message is a response, the service is not restarted, this will be analysed in the next subsection.

After the message is converted to a *String*, it is passed to the method *analyzeMessage()*, where the device will see how to deal with the received message. Here, the message type is assessed, in this case the focus will be the *ADV* type. If the message is identified as an advertising message the method will extract its estimate and compare it to the device's best estimate, retrieved from *getMinHop()*. If the comparison concludes the received estimate does not top the previous best, the routing table is updated with the received estimate and MAC address, through *updateRouteTable()*, previously explained. Otherwise, it means the device has found a better way to reach the Internet, so after the routing table is updated, the variable *discoveryFinished* is set to false and the discovery process is done again, through the method *doDiscovery()*.

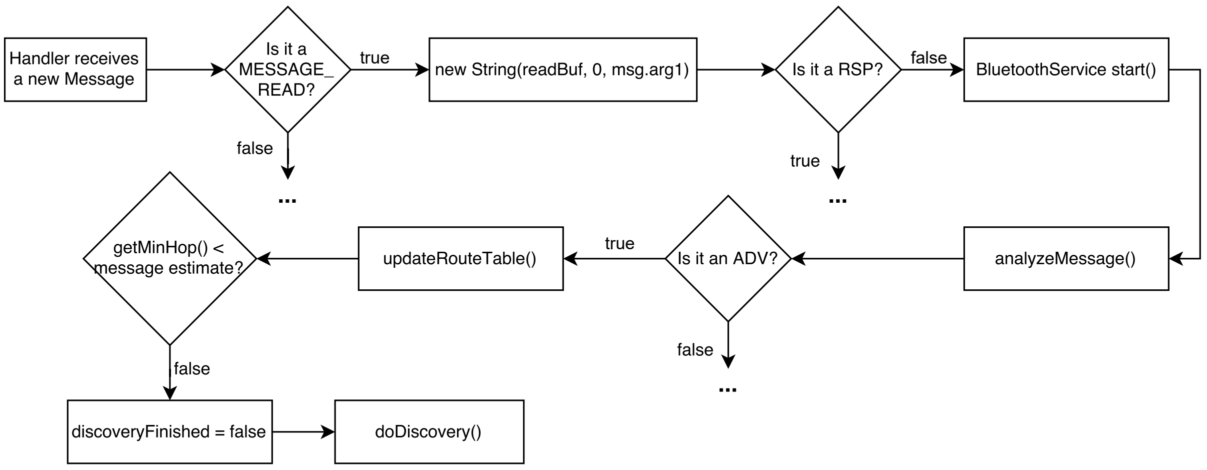
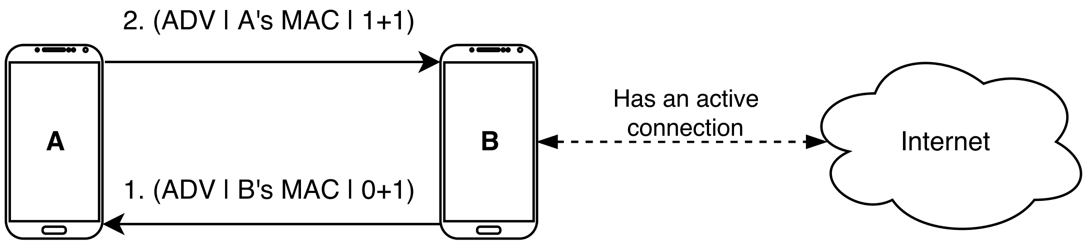


Figure 3.5: Fluxogram of the advertising process from the point of a receiver

In figure 3.5 it is shown a fluxogram for a receiver device and its order of instructions. The device ends this process either with a *doDiscovery()* method call or by "listening" to incoming connections. Some aspects of the fluxogram are simplified as they do not concern this process and will be discussed later.

To summarize, in figure 3.6 the example with devices A and B from figure 3.2 is extended. Now device B has advertised to its peers, which include device A. Upon receiving this message, device A proceeded to store the information in its routing table, followed by and advertise from itself, since the



Next Hop's MAC	Number of Hops
Own MAC	16
B's MAC	1

Next Hop's MAC	Number of Hops
Own MAC	0
A's MAC	2

Figure 3.6: Example 1: State of the two devices after the advertising process is done

new estimate tops the previous one. After this is concluded both devices have the shown routing tables, and A knows it routes to B, while B routes to itself, since A provides a poorer estimate.

If A advertises first the result would be the same, since when B is advertising A would still receive a better estimate and would advertise again. When B receives the second advertise from A it will overwrite the previous entry, maintaining the same values.

Now it should be clear how the discovery and advertising process is performed and what is the code executed at each time of the process. This is the basis of the next subsection, since without it the devices would not be able to reach the Internet unless they already had the connection.

By giving each device a full view of its vicinity it is possible to establish a network in which every device knows the best path instantly. This is done by what was described above, from discovery to advertisement. The next subsection will refer to the next step, where the devices already have their routing tables populated and are ready to exchange web pages.

3.1.3 Exchange of web pages

In this subsection the logic created and implemented to exchange the web pages will be explained, in detail, also an example, similar to the previous one will be shown to better demonstrate the processes. This analysis will be mainly focused in appendix A.2. However, before that it is necessary to explain the routing table related to the exchange of web pages, previously mentioned in subsection 3.1.2 and referred to as response table.

The main purpose of this table is to provide a destination for a response message. Once a device has received a request and it has an Internet connection, it should know where to send back the response. This would be easy if only this case applied, it would simply send back the response from where it received the request. But, in the case this device is a ”bridge” node, *i.e.*, it is simply a node that forwarded a request, it may have received requests from different devices and it still needs to be able to differentiate each message and decide which device to send back the response to.

The response table has a similar structure to the routing table, see figure 3.2, but it serves a different purpose. Similarly to routing table, the response table is defined in *RoutingApp* class, it is an object of the Java class *Map*.

Message ID	**Next hop's MAC**
Message ID #1	Device X's MAC
Message ID #2	Device Y's MAC
Message ID #3	Device Z's MAC
...	...
Message ID #9	Device X's MAC

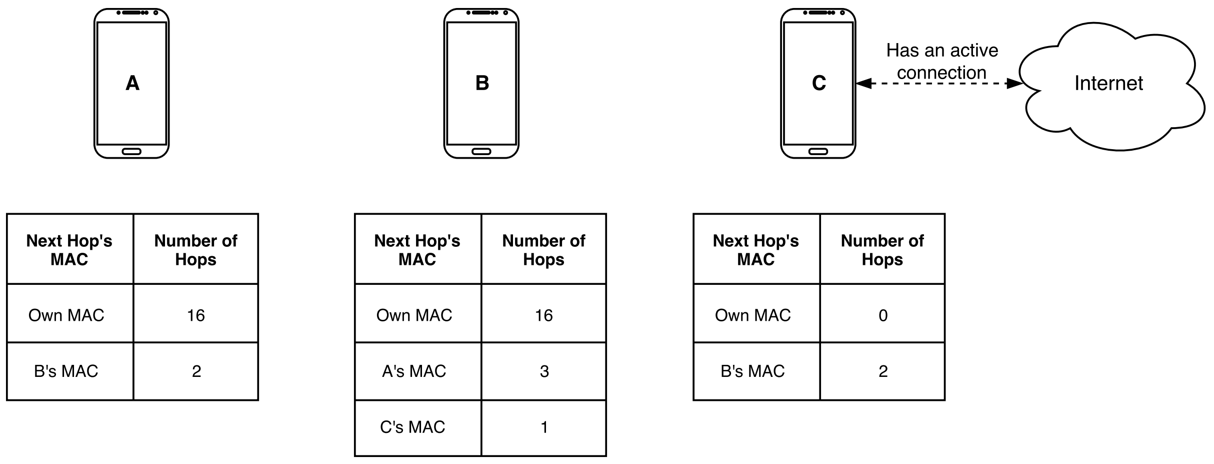
Table 3.2: Response table example and format

Despite the similarities, there is one difference between the two: routing table maps *Strings* to integers whereas response table maps integers to *Strings*. The latter is used to associate message identifiers, the integers, with the MAC address of the next destination of a response, the *String*.

The table is defined as *rspTable* and it has two associated methods: *updateRspTable()* and *getRspHop()*. The first method is used to add new rows to the response table, it is similar to routing table's method *updateRouteTable()*, and makes use of the same Java method *put()*, previously explained. Although it has a slight difference in logic, since this method will not update rows, it will always insert new rows, seeing the message identifiers are unique there should be no duplicate entries.

The second method, *getRspHop()* is used to retrieve the MAC address associated to a certain message identifier, that is as a parameter. In case a table entry is found for that specific identifier, the associated MAC address is returned, otherwise it will return null, notifying the application that no entry was found and thus, the response will not be routed.

For the example three devices are used: A, B and C. Where device C has an active Internet connection, as seen in figure 3.7. In this figure it is possible to see the routing tables of each device after this process is completed. Device A will choose B to send a request and B will choose C, since they provide the best estimates, respectively.



A:

Next Hop's MAC	Number of Hops
Own MAC	16
B's MAC	2

B:

Next Hop's MAC	Number of Hops
Own MAC	16
A's MAC	3
C's MAC	1

C:

Next Hop's MAC	Number of Hops
Own MAC	0
B's MAC	2

Figure 3.7: Example 2: State of the routing tables of the three devices

3.1.3.1 Sending a request message

Now that the logic and implementation behind the response table is understood it is possible to start explaining the actual process of web page exchanging. Starting by the definition of variables, in *BtActivity* class, it can be divided into three sections: the first being the definition of variables that will identify the different elements seen by the user, a *Button*, to submit requests, an *EditText*, for the user to input the requested Uniform Resource Locator (URL) and a *WebView*, to manage the actions related to web pages.

It is then possible to see the definition of the constants relative to the *BluetoothService* class, see 3.1.1. The possible *Message* types received from the service are defined: *MESSAGE_STATE_CHANGE*, *MESSAGE_READ*, *MESSAGE_WRITE*, *MESSAGE_DEVICE_NAME*, *MESSAGE_TOAST*, *FILE_READ* and *FILE_WRITE*. These types are explained in 3.1.2.3, and *MESSAGE_READ* was already mentioned, although only covering the advertising messages.

Finally, the last block of variables, containing various definitions relative to this class. *msgID* is the variable that concerns this process of exchange, it stores the message identifier of the last received request or response.

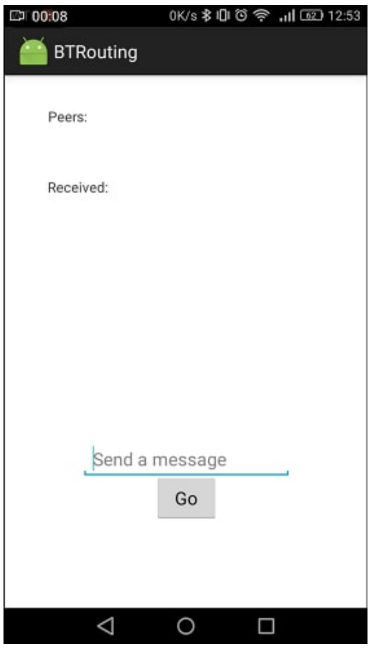


Figure 3.8: User interface for the sending of request messages

In figure 3.8 it can be seen the display of the main activity page, it has five main elements: a place where the *peers* list will be displayed, at the top. The received messages, this is optional and used for debugging as it would incur in severe security issues, so it should be removed if not for debugging. An *EditText*, see [22] for documentation on this Android feature, used to allow the user to input the web page he/she desires and to capture this input to be processed. A *Button*, to notify the activity it should now begin the process of requesting the web page. Finally, a *WebView*, see [23] for documentation on this feature, that is hidden from user view and will remain so until the user receives the response to its request.

Assuming the device already established the best route to reach the Internet, the journey of the request, from the user input to the display of the response, will be explained. Starting by the user input, has said above, a *Button* and an *EditView* are defined, so a user interface is created, in order to send requests based on the user input. This is defined in method *onStart()*, from appendix A.2, after the routing table is first populated and the discovery is started, discussed in 3.1.2. The variables previously defined, *goButton* and *mEdit* are matched to the elements in figure 3.8, so they can be identified by the application.

The *Button* class' method *onClick()* is then overwritten, meaning it is tailored for the specific needs of this application. This method has the purpose of responding to a click of the user in the specific button, to this click the method must associate an intent to send a request message. With these parameters the method *onClick()* is set to call *sendRequest()*.

The *sendRequest()* method receives as parameters: a boolean to identify if the intent is coming from the owner of the request or from a mediator node, an integer that corresponds to the message identifier and a *String* containing the requested URL.

Since this particular request comes directly from the user input, this method is initially called by this device with the following arguments: "true", "-1" and the user input text. The "true" value is used whenever the device is the one issuing the request, *i.e.*, whenever it is its origin. "-1" is used when the value of the message identifier is not important, in this case, since this device is the "owner" of the request, a random message identifier will be generated. Finally, to identify the web page requested, the user input URL is passed.

Type	Message ID	Bluetooth MAC Identifier	URL to request

Figure 3.9: Request message format

In figure 3.9 a generic request message is shown. Its format does not differ much from the one presented in figure 3.1, however there is a new field, the *Message ID*, a unique identifier representing each message that will be useful to keep track of what is each message's payload and sender. The *URL to request* is the web page's URL the user is trying to access, it is considered to be the payload of the message.

Continuing the analysis on method *sendRequest()*, the first thing that is done is to retrieve the next hop, to whom the device shall send its request, this is done by calling the method *getNextHop()*, defined in A.3. This method returns the MAC address of the device's peer who provides the best path. *getMinHop()* is called to retrieve the best estimate, to which is applied the function *getKeyFromValue()*, alongside with the routing table, see subsection 3.1.2.1 to get more a detailed explanation on how these two methods work. This combination should return the desired MAC address, however if the best estimate is 16, "null" is returned to notify the device that no path was found.

Once the MAC address of the next hop is attained, *sendRequest()* compares it with "null", to ensure there is an actual device to connect to. In case this comparison is successful, meaning there is no next hop, the method *sendFail()* is called. Since this device is the owner of the request, the *sendFail()* method will not send a failure notification to another device, it will simply notify the user of the failure. The method *sendFail()* will be discussed further on this subsection.

In the case of the device having a valid next hop, the connection is performed, similarly to the advertising connections, by the *BluetoothService connect()* method. The first cycle described in the advertising process is now repeated here, to ensure the connection is established before sending the message, with the single difference of in case of failure, *i.e.*, the Bluetooth state becoming *STATE_LISTEN*, *sendFail()* is called again, to notify possible predecessors of this request, however in this case it does not apply.

If the cycle is finished and everything is completed without any incidents, the method verifies the first argument, the boolean variable to identify if the device is the owner of the request. If the device is the owner a random message identifier is generated through the Java class *Random*, see [24]. Once the identifier is generated, the message is sent with the parameters: ”RQT” as *Type*, the value of the generated random as *Message ID*, the device’s MAC address gotten through *getOwnMAC()* as *Bluetooth MAC Identifier* and the received URL from the arguments as the *URL to request*, as seen in figure 3.9. The response table is also updated with the newly generated message identifier and device’s MAC address, through *updateRspTable()*.

Otherwise, if the device is not deemed as the owner, there is no random identifier generation, since the value will come from the received request. Also the response table is not updated, since it will be updated earlier in the receiving process, that will be the next focus. However, the message is sent as before, with the slight difference that the *Message ID* is now the one received as an argument.

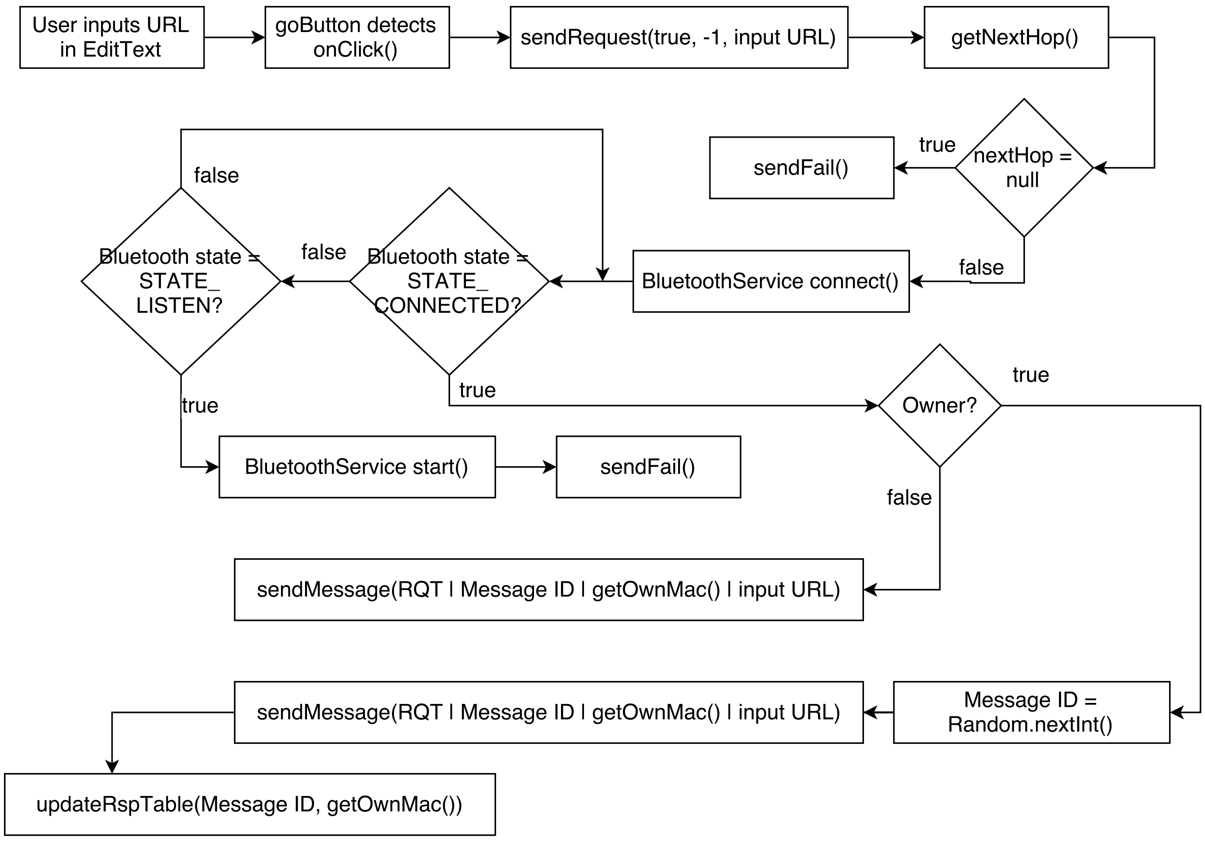

User inputs URL in EditText
goButton detects onClick()
sendRequest(true, -1, input URL)
getNextHop()
nextHop = null
true
sendFail()
false
BluetoothService connect()
Bluetooth state = STATE_ CONNECTED?
false
Bluetooth state = STATE_ LISTEN?
false
true
BluetoothService start()
sendFail()
true
Owner?
true
false
sendMessage(RQT | Message ID | getOwnMac() | input URL)
Message ID = Random.nextInt()
sendMessage(RQT | Message ID | getOwnMac() | input URL)
updateRspTable(Message ID, getOwnMac())

Figure 3.10: Fluxogram of the request sending process

In figure 3.10 a fluxogram representing the process of sending request message is presented. It provides a better overview of the code hierarchy, and when is each method called. Once the process is finished the device keeps listening to incoming connections, changing the Bluetooth state to *STATE_LISTEN*.

3.1.3.2 Receiving a request message

The request receiving process is similar to the advertise, the receiver connects to the sender and receives the bytes from the request sent. As before, the bytes are transmitted from the *BluetoothService* to the main activity, through a *MESSAGE_READ Message* and the *mHandler* receives them. After comparing the received *Message* to the possible types, explained in subsection 3.1.2.3, and concluding it is a *MESSAGE_READ* the *Handler* converts them to a *String*. Since the newly created *String* is not a response, the *BluetoothService* is restarted and *analyzeMessage()* is called taking as argument the *String*.

In *analyzeMessage()* the argument *String* is compared to the possible message types, *ADV*, *RQT*, *RSP* and *FAIL*. It will now be identified as a response, due to the analysis of the message type, see figure 3.9, and immediately the message identifier is saved in the variable *msgID*.

In the request sending process it was mentioned that, in case the device was not the owner of request, it would not update the response table. This was due to the fact that, for the devices that are not the owners of the request, this process is done here, in *analyzeMessage()*. So, the routing table is updated with the message identifier and MAC address of the sender, retrieved from the message.

Two different approaches can now be taken: one regards the devices with Internet connection and the other the devices without one. To define which approach is taken by the device a quick check of the Internet connection status is performed, via the method *getHasNet()*. If the result is false, meaning the device has no Internet connection, the request will be forwarded to the next hop to reach a device with Internet. To do this, the device calls *sendRequest()* with arguments: false, the message identifier, retrieved from the received message and the URL requested, also retrieved from the message. The method follows the same steps as before, described in 3.1.3.1.

If the result of the *getHasNet()* query returns true, it means the device has an Internet connection and, as such, it is a final destination. This means the device will now be the communication link with the Internet and it needs to retrieve the requested web page and send it backwards until it reaches the owner of the request. This process begins by calling the method *getPage()*, that receives the requested URL and the message identifier, both retrieved from the received message.

In *getPage()* the web page is downloaded via the *WebView*, mentioned in 3.1.3.1. A new *WebView-Client* is created, see [23] for documentation on *WebView* and its features, such as the *WebViewClient*. This client is different from the usual, since the device does not want to display the downloaded page.

The methodology to save the web page had several possibilities, such as saving the web page as an image, saving only the *HTML* content or saving each element of the page individually. The first two methods are not complete, meaning the web page could lose some of its features, *e.g.* dynamic images, search fields, *etc.*. The third method would fully download the web page, however it would require additional logic to save the different elements in the same directory and to arrange them to re-create the web page with its initial format.

The method *saveWebArchive()* provides a complete solution to this problem, it is a method specific to the *WebViewClient*'s class, see [23] and it solves the problem of third method, since it downloads each element but compiles them in a web archive, that can be decompressed easily by *WebView*. Thus, it proved to be the better solution for this problem.

To implement this methodology, the method *onPageFinished()*, from *WebViewClient*'s class, is overwritten. This method is used to do something after the web page is finished loading. So, a new *File* is created in the application's directory, with the name ”file.mht”. If the file creation succeeds, the web page is downloaded and saved in the created file, via *saveWebArchive()*. In order to guarantee the web page is downloaded and saved before the response is set, a *waitForWebPage* downloader is created and executed.

The *waitForWebPage* class can be seen in appendix A.2 and was created with the sole purpose of guaranteeing the web page transfer is completed before the response is sent. If this check is not performed, it is possible that, for large web pages, the response file will be incomplete and thus not displaying correctly the requested URL.

It extends an *AsyncTask*, used to perform background operations and send the results to the main thread, see [25] for the full documentation. It starts by creating a variable *file* that will contain the address where the web page will be saved. The method *doInBackground()* is overwritten, as it always has to be in an *AsyncTask*, since it contains what operations to be performed.

In this method it is continuously checked if the variable *file* has a size bigger than 0 bytes. Whenever this condition is verified, it means the web page was saved successfully. The *onPostExecute()* method, used to send the results to the main thread, is also overwritten and, once this thread reaches this point, it means the device is ready to send the response, so *sendResponse()* is called.

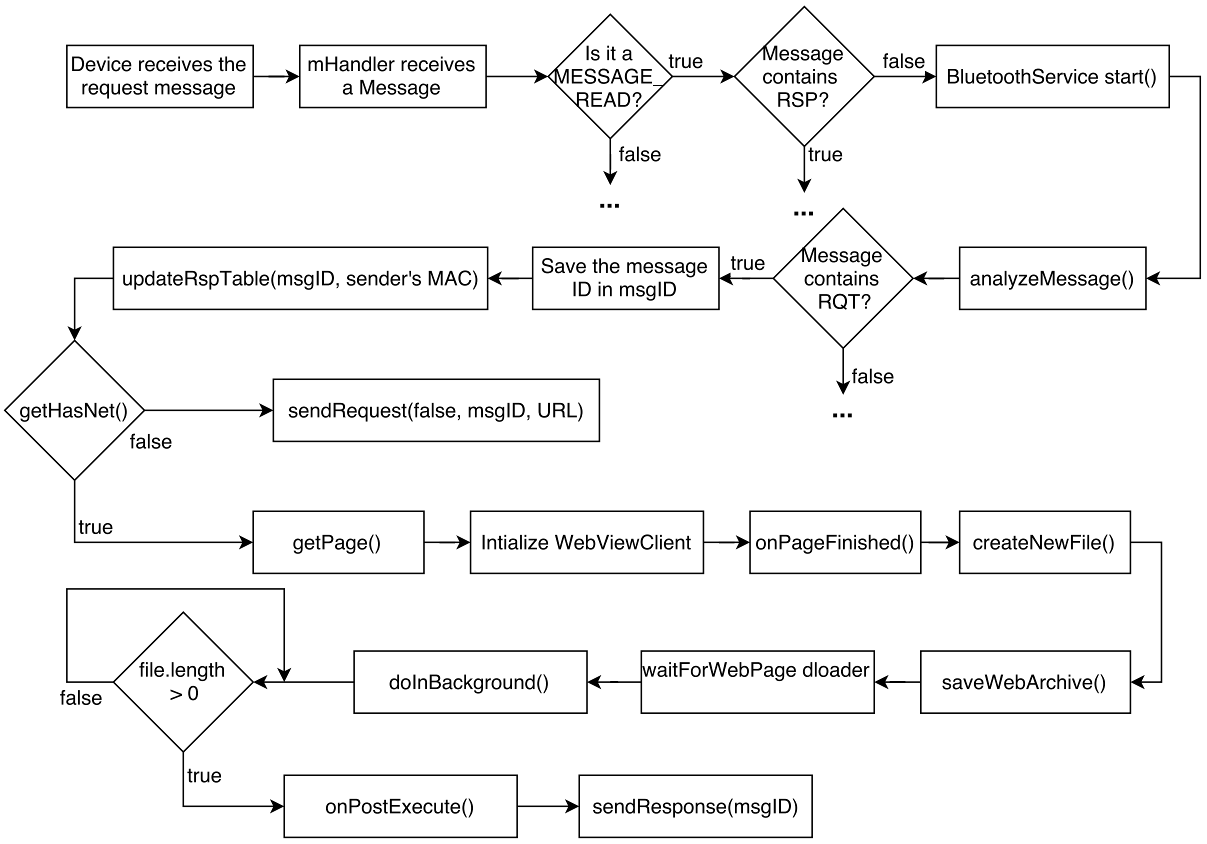


Figure 3.11: Fluxogram of the request receiving process

In figure 3.11 it is possible to see a fluxogram of this process. The process, as shown in the figure, ends either with sending a new response or a new request, depending on whether the device has Internet connection or not, respectively.

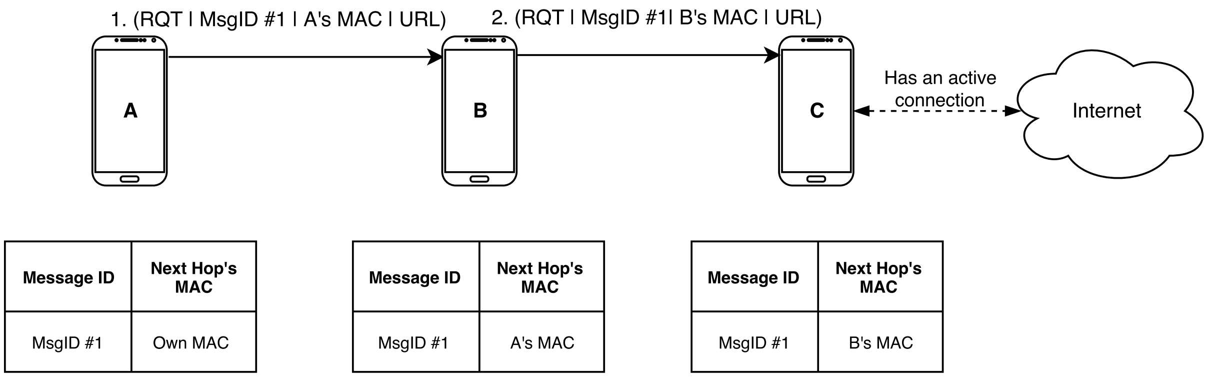


Figure 3.12: Example 2: Request sending and receiving process

Resuming the example in figure 3.7 and supposing the user from device A wants to request a URL and inputs it correctly, the device will follow the steps explained in subsection 3.1.3.1 and check the routing table to establish the next hop. Also its response table will be filled during this action, since it is the owner of the request.

Once that process is completed and the message reaches device B it will update its routing table with the newly received request. After that it checks its own routing table and assesses if it should forward the request or download the page. Since the device does not have an active connection the first option will chosen and the request forwarded to C.

Finally, upon receiving the request, device C will update its own response table with the received message identifier and B's MAC address. Having an Internet connection the device will download the web page.

In figure 3.12 it is shown the above described, as well as the three response tables from the devices once the requests are all sent and the web page downloaded.

3.1.3.3 Sending a response message and web page

Once the web page is successfully saved in the device it is possible to send the response back until the owner of the request receives the requested web page. As seen before, *sendResponse()* is called in *onPostExecute()* and it is used to send a response to the next device.

It receives as parameter the message identifier, so that the MAC address of the next device can be retrieved from the response table. Thus, having the message identifier and through the method *getRspHop()*, the device retrieves the next hop's MAC address. Should the result of this call be null, the response will not be sent. If the result returns a valid MAC address, the connection process is performed, along with the cycle to ensure the connection is successful, described in subsection 3.1.2.2.

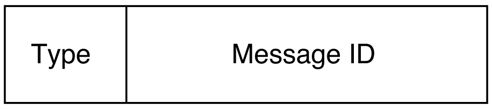


Figure 3.13: Format of a response message

Finally, if all the above goes as planned and succeeds, the response is sent with the format seen in figure 3.13, with "RSP" as *Type* and the received argument *msgID* as *Message ID*.

The next step is to send the actual web page to the next hop, but first the sender needs to ensure the response was correctly sent. This is done by the *Handler*, explained in subsection 3.1.2.2. The *Message* type *MESSAGE_WRITE* is used for this, since it returns the bytes sent from the device and convert them to a *String*. If the converted *String* is a response message the application concludes the process is completed correctly and proceeds to call the method *sendFile()*.

The *sendFile()* method is called when the device is ready to sent the web page bytes to the next hop. It begins by creating a variable *file*, pointing to the file in the application storage that contains the web page. The size of that file is retrieved through *file.length()* and an array of bytes is created with that same size. This array of bytes will be the place where the web page will be transferred, from the internal storage to the application.

In order to transfer the web page from the file, in the internal storage, to the application a *BufferedInputStream* is used, see [26] for more documentation on this class. This *BufferedInputStream* instance takes the bytes from the variable *file*, previously described and saves them in the array of bytes, it is then closed to avoid memory leaks.

Once the bytes are transferred to the application the device needs to send them to the next hop, since the connection is still opened, for reasons that will be explained in the next subsection, it is possible to send the bytes without the logic to connect to the next hop. The method *writeFile()* from *BluetoothService* is called and the file bytes is transferred to the next hop as described in subsection 3.1.1.3.

The method gets the number of bytes to be sent and divides that size by 990 bytes, getting the number of chunks that need to be sent to the next hop. It then proceeds to send each chunk with a length of 990 bytes, until it reaches the last chunk, sending the remaining bytes.

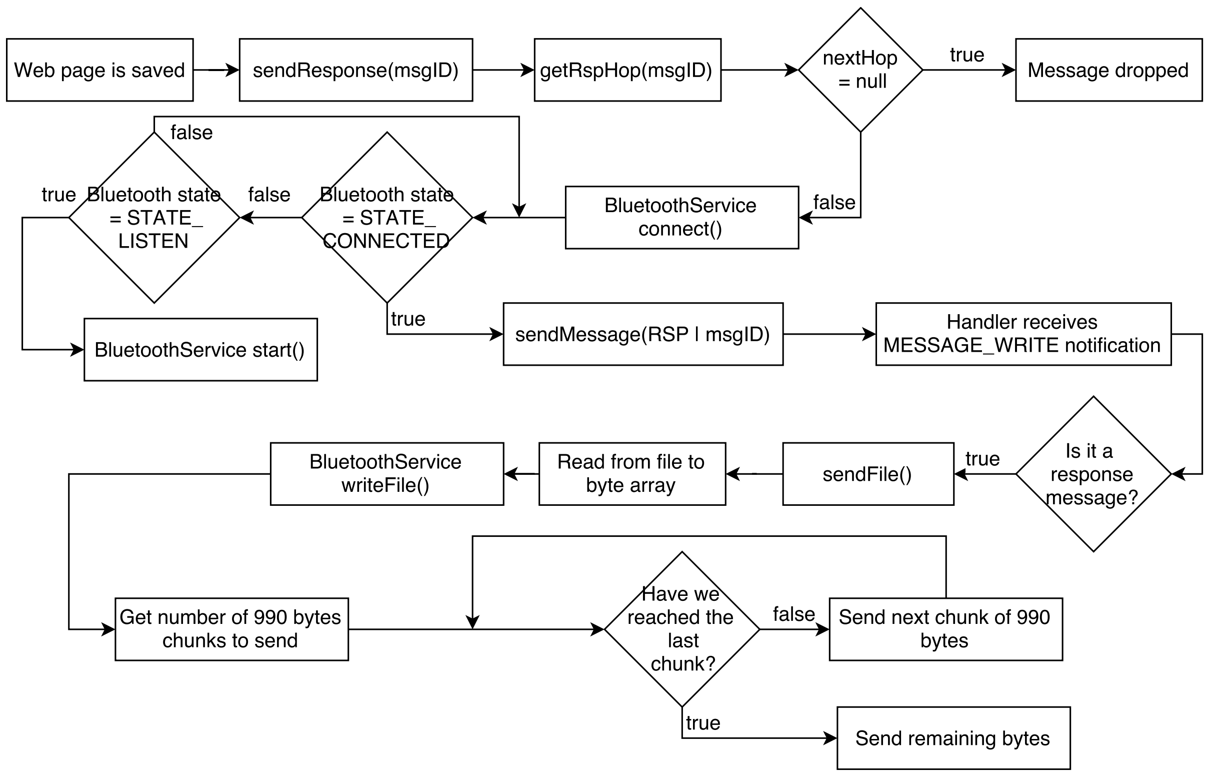

Web page is saved
sendResponse(msgID)
getRspHop(msgID)
nextHop = null
true
Message dropped
false
BluetoothService connect()
Bluetooth state = STATE_ CONNECTED
false
Bluetooth state = STATE_ LISTEN
false
true
BluetoothService start()
true
sendMessage(RSP | msgID)
Handler receives MESSAGE_WRITE notification
Is it a response message?
true
sendFile()
Read from file to byte array
BluetoothService writeFile()
Get number of 990 bytes chunks to send
Have we reached the last chunk?
false
Send next chunk of 990 bytes
true
Send remaining bytes

Figure 3.14: Fluxogram of the response and web page sending process

In figure 3.14 a fluxogram of the response sending response is presented. It shows the process from the saving of the web page to the sending of the same. As previously described the fluxogram ends by dropping the message if there is no valid next hop for that message identifier, by restarting the *BluetoothService* if the connection fails, or by sending the full web page as expected.

3.1.3.4 Receiving a response message and web page

In the receiver side, the web page and response are received and the device must assess if it is the final destination or if it is a relay node for a different device, in which case the web page will not be displayed, but the response and web page are forwarded.

This process begins in the *Handler*, where the notification is notified that a message as been received by a *MESSAGE_READ*. The received message is compared with the possible types and, since it is a response message, it is identified as such. In the case of a response, the *BluetoothService* is not restarted, since the device benefit from keeping the connection active to avoid further delays in re-connecting to receive the web page, also mentioned in subsection 3.1.3.3. Instead, the *Bluetooth-Service*'s variable *fileReady* is set to ”true”, meaning the device is has received a response message and is ready to receive the associated web page.

The message is then sent to method *analyzeMessage* as a *String*, where the message identifier will be saved to *msgID*, retrieved from the response message, see figure 3.13. Having the message identifier the method compares the device's MAC address to the one returned from *getRspHop()* for that same message identifier. If both addresses match the device is the destination and the owner of the request, so the web page will be displayed. Otherwise, the response will be forwarded to the next destination.

The sender will then proceed to transfer the web page bytes from one device to another. At this point the receiver is expecting a file, since the variable *fileReady* is set to ”true”. This shifts the *BluetoothService* reception logic from receiving a *String* to receiving a web page, see subsection 3.1.1.3 for more details on this logic.

A variable *output* is created, where each 990 bytes chunk is saved. When the transfer reaches the last chunk, identified by its size, which will be smaller than 990 bytes, the information contained in this variable is passed to a byte array, that will be sent back to main activity via a *FILE_READ Message*.

Once the main activity's *Handler* receives this notification it copies the bytes received from *BluetoothService* to a file in the application's directory. The *BluetoothService* is restarted and the variable *fileReady* is set to ”false”, as the device does not expect to receive anymore files for the time being. A new comparison between the device's own MAC and the result of *getRspHop* for that specific message identifier is made. If the result is false, meaning the device is not the destination, the response is forwarded through *sendResponse*, with the corresponding message identifier, see 3.1.3.3 for information on this process.

However, if the device is deemed the owner of the original request and thus the destination of the web page, the method *loadPage()* is called, were the logic to display the web page is performed. First the *WebView* instance is set to load web pages first from cache instead of directly trying to access the network, via *setCacheMode(WebSettings.LOAD_CACHE_ELSE_NETWORK)*, see [23] for more documentation. It is followed by setting the *WebView*'s visibility to *VISIBLE*.

There is one important aspect of this application that was not yet discussed. When the requester receives the web page, it might be needed to request multiple pages, for instance in a Google search, the user requests the front page and then inserts its query and a subsequent page is requested. This logic has to be reflected in the application for it to be deemed useful and usable. If the user had to go back and input each URL by itself the application would have no real application.

WebViewClient's class may have a solution to this problem. The class method *onReceivedError* is triggered whenever the *WebView* receives an error, meaning it is possible that, whenever a ”no Internet” error is received logic can be created in order to prevent the *WebView* to display that error and instead create a request with the URL that returned the error.

To do that, this method is overwritten, beginning by loading the page ”about:blank”, a page composed of a white screen, in order to avoid the user seeing the error page. A request message is then sent, via *sendRequest* with the arguments: ”true”, -1 and the URL that originated the error, see 3.1.3.1 for a reminder on how this process is carried. Alongside this, a new *WebChromeClient* is assigned, to handle the display of the web page, see [23] for more information on this class.

Finally, the web page is displayed via the *WebView* method *loadURL*, which for Android versions superior to 5.1 works perfectly. However if the device's Android version is prior to that the archive where the web page is saved has a different formatting and, as such, needs to be compiled again, via *loadArchive()* by calling *WebView*'s method *loadDataWithBaseURL()* with data retrieved from the saved archive with the methods *getStringFromFile()* and *convertStreamToString()*, taken from [27].

In figure 3.15 a simplified fluxogram of the response and web page receiving process is shown. It is possible to visualize the file receiving logic, as well as the different possibilities for the receiver of the response: forwarding the response or displaying the web page. It is also shown the logic behind the multi web pages request.

With this implementation the user is capable of seeing the request web pages and navigate through those pages without having to manually send each request.

To finalize the example from figures 3.7 and 3.12, device C, after finishing the download of the web page, will check its response table and send a response message to device B, which is the next hop retrieved for that specific message identifier *MsgID #1*. The response is followed by the web page archive, previously downloaded, as described in subsection 3.1.3.3.

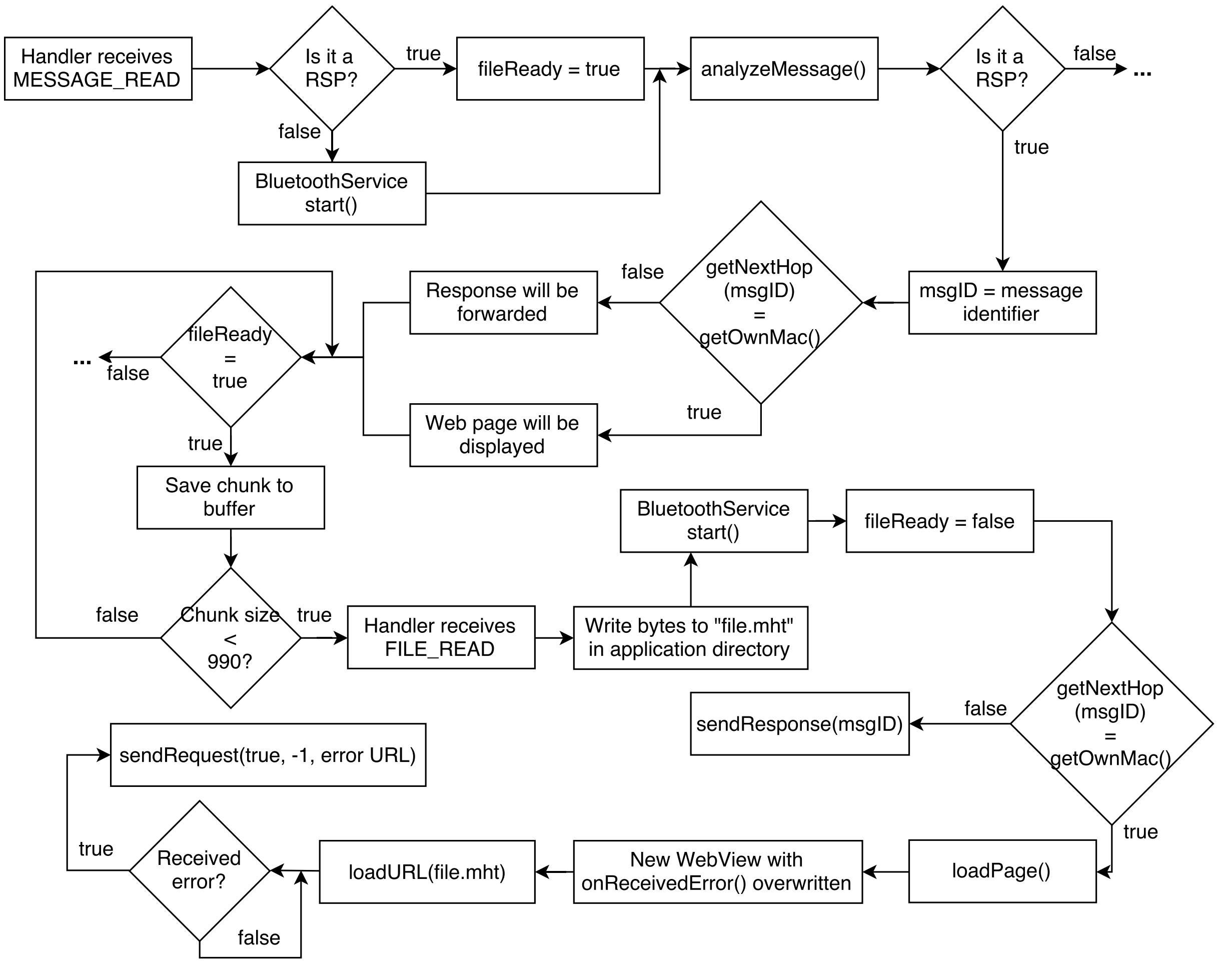
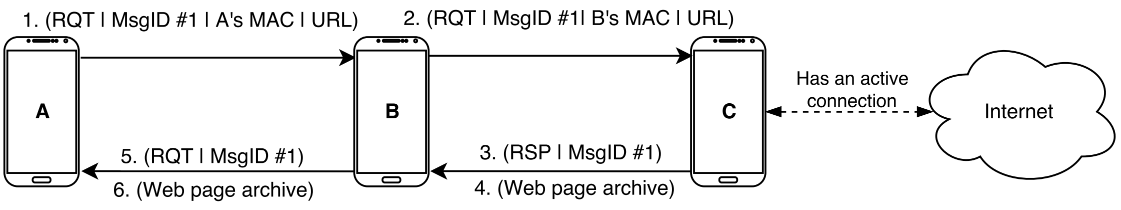

Handler receives MESSAGE_READ
Is it a RSP?
true
false
fileReady = true
BluetoothService start()
analyzeMessage()
Is it a RSP?
false
...
true
msgID = message identifier
getNextHop (msgID) = getOwnMac()
false
true
Response will be forwarded
Web page will be displayed
fileReady = true
...
false
true
Save chunk to buffer
Chunk size < 990?
false
true
Handler receives FILE_READ
Write bytes to "file.mht" in application directory
BluetoothService start()
fileReady = false
getNextHop (msgID) = getOwnMac()
false
true
sendResponse(msgID)
loadPage()
New WebView with onReceivedError() overwritten
loadURL(file.mht)
Received error?
true
false
sendRequest(true, -1, error URL)

Figure 3.15: Fluxogram of the response and web page receiving process

Device B receives the response and the web page following the steps previously explained in subsection 3.1.3.4 and checks its response table for that message identifier. Device A's MAC is returned from that query and that's the destination for B's response, so device B sends the response and web page to A.

A receives the response and web page following the same steps as B. However, when checking the next hop for that message identifier device A gets its own MAC address and concludes it is the final destination for that response, proceeding to display the web page requested by the user, as described this subsection.

Figure 3.16 illustrates this example and messages exchanged between the three devices, as well as the response tables, previously established in figure 3.12.



Message ID	Next Hop's MAC
MsgID #1	Own MAC

Message ID	Next Hop's MAC
MsgID #1	A's MAC

Message ID	Next Hop's MAC
MsgID #1	B's MAC

Figure 3.16: Example 2: Sending and receiving of response messages and web pages

Chapter 4

Tests and Results

In this chapter several experiments will be performed in order to assess the quality and features of technologies and developed application. The main goals are to obtain an empiric and realistic conclusion on which technology serves better the purpose of this thesis and to assess how the application fares when subject to different stress tests.

This chapter will be divided in two different sections: one regarding the comparison between Bluetooth and Wi-Fi Direct via different tests and other regarding the tests performed with the developed application, to better understand where it performs better and worse, proving or not if the theoretical choices made translate into actual performance gains.

4.1 Bluetooth vs. Wi-Fi Direct

This section will cover the differences, advantages and disadvantages of both Bluetooth and Wi-Fi Direct. A series of experiments will be conducted and their results analyzed, in order to justify which technology is best suited for this application and applications with a similar architecture and/or purpose.

The tests will range from battery consumptions to data rates and most of them will be backed up by both theoretical and empirical results, although in some of them, due to the inability of getting precise measures, the results will be taken from developed experiments.

4.1.1 Ranges of communication

The ranges of communication of both technologies are of extreme importance. They can reduce or increase greatly the number of hops a packet has to pass through, in order to reach the destination. If the range of communication is too short, the number of connections made will increase, this may cause an overload of the network, and the deterioration of the communication medium. On the other hand, if the communication range is long the devices are able to jump through bigger hops, creating less traffic in the network and establishing the least possible number of connections.

Both technologies share some similarities, they are both dependent on the environment of the communication, the elements that are surrounding the devices and possible obstacles in the way. The experiments were conducted, for the obstacle experiment, in a corridor of Torre Norte in the vicinities of Instituto Superior Técnico and, for the line of sight experiment in the outside area of the same. It is important to note that although this experiments were made with as little interference as possible, there are certain elements that are impossible to controls, such as wireless communications from other devices, metal objects, such as metal lockers and cars.

Bluetooth establishes four different classes for the devices that may use this technology, depending on the transmitting power. Mobile phones are inserted in class 2 and, for that class, the specified average range of transmission in order to have a reliable connection is 10 meters, from [28].

Wi-Fi Direct on the other hand offers, theoretically, ranges of communication up to, approximately, 200 meters, from [29], which poses for a much better solution, in terms of network off-loading and general depth.

In order to verify these claims from both technologies, two mobile devices were taken to an open space, although with some limitations as described above, and several searches were made, until the devices stopped being discovered by one another. After measuring the distance between both devices, the results were taken, and prove what was already to be expected, although with some twists.

Bluetooth was able to create a connection between devices from a distance up to 42 meters apart, see figure 4.1 for the overall scheme of this experiment. This value is a lot more than what was expected judging by the theoretical value of 10 meters, although the health of the connection was not verified, see 4.1.4 for these tests. Using Wi-Fi Direct the devices were able to communicate from a maximum distance of 77 meters, see figure 4.2 for the overall scheme of this experiment, which is, considerably, smaller than the theoretical value of 200 meters.

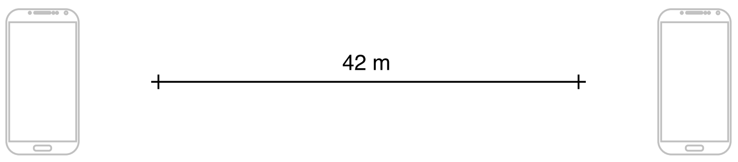


Figure 4.1: Max range of Bluetooth communication with line of sight between devices

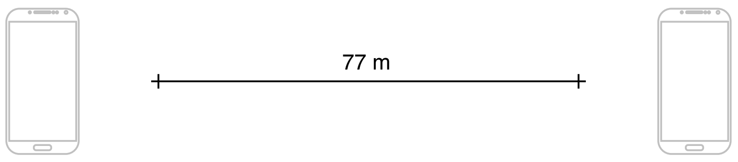


Figure 4.2: Max range of Wi-Fi Direct communication with line of sight between devices

Both tests were made with a direct line of sight between devices. For the next ones there will be obstacles in the way of communication. It is expected that this affects greatly the communication ranges. The first test was made using Bluetooth technology where a wall was blocking the line of sight between devices, see figure 4.3. The second test was made using Wi-Fi Direct, and, in order to maintain the same environment as the previous experiment, to get reliable results, it was situated in the same place as the first, see figure 4.4. However, due to the environment configuration, it was impossible to recreate the experiment with only one wall, so two walls are now dividing the devices. Since the walls introduce a loss in the signal power, the more walls are between devices, the bigger the losses will be.

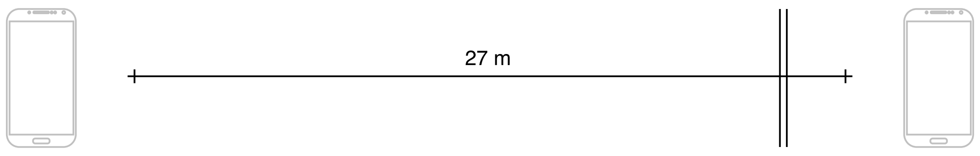


Figure 4.3: Max range of Bluetooth communication without line of sight between devices

As expected the obstacle, in this case the wall, created a significant decrease on the maximum range of communication. Bluetooth was able to communicate from a distance of 27m, closer to the theoretical 10 meters.

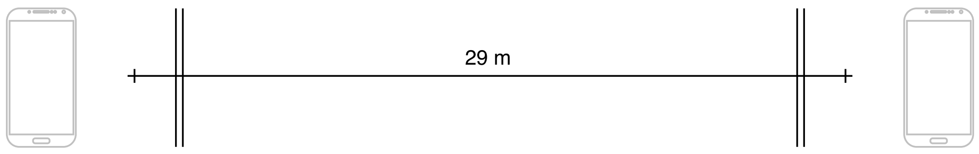


Figure 4.4: Max range of Wi-Fi Direct communication without line of sight between devices

Wi-Fi Direct was also able to communicate from a smaller maximum distance, measuring 29 meters, with the signal passing through both walls. From a smaller distance it was verified that this technology could communicate with only wall in the way, meaning it also surpasses Bluetooth when an obstacle is in the way of communication.

After these experiments it is possible to conclude that Wi-Fi Direct is more desirable, since it provides better coverage than Bluetooth to similar areas. Also, there is no evidence that Wi-Fi Direct suffers more losses from obstacles, maintaining its desirability. This was already to expect, both from the theoretical values and from the transmission powers[1], since Bluetooth is mostly known for its lower transmit powers, if compared to technologies such as Wi-Fi.

4.1.2 Battery consumptions

4.1.3 Discovery times

check

4.1.4 File transfer data rates

4.2 Tests on the developed application

4.2.1 Ranges of communication

4.2.2 Battery consumption during use

4.2.3 Discovery times on different topologies

4.2.4 Routes taken on different topologies

4.2.5 File transfer data rates on different topologies

[1]Transmission powers impact directly the range of transmission, since they affect the signal strength, a crucial characteristic for receivers to better capture the transmissions. A bigger transmit power, usually, creates a bigger signal strength leading to the signal being capture over bigger distances, as referred in [30], for instance.

Chapter 5

Conclusion

There are also some features that lack implementation, such as the request of multiple pages, video streaming and file downloads.

If this idea is pursued and further thought on its development is given it is my conviction that it can be a valuable asset to Android devices - such as is hot spot nowadays. Ho, to achieve a customer ready solution many problems must be solved, as well as new features added.

Bibliography

[1] W. S. Conner, J. Kruys, K. J. Kim, and J. C. Zuniga, “Overview of the amendment for wireless local area mesh networking,” 2006.

[2] C. Casetti, C. F. Chiasserini, L. C. Pelle, C. D. Valle, Y. Duan, and P. Giaccone, “Content-centric routing in wi-fi direct multi-group networks,” 2014.

[3] C. Funai, C. Tapparello, and W. Heinzelman, “Supporting multi-hop device-to-device networks through wifi direct multi-group networking,” 2015.

[4] A. A. Shahin and M. Younis, “Efficient multi-group formation and communication protocol for wi-fi direct,” 2015.

[5] K. Liu, W. Shen, B. Yin, X. Cao, L. X. Cai, and Y. Cheng, “Development of mobile ad-hoc networks over wi-fi direct with off-the-shelf android phones,” 2016.

[6] Statista, “Average number of connected devices used per person in selected countries in 2014.” https://www.statista.com/statistics/333861/connected-devices-per-person-in-selected-countries/, 2014.

[7] I. Poole, “Ieee 802.11n standard.” http://www.radio-electronics.com/info/wireless/wi-fi/ieee-802-11n.php.

[8] S. Choudhuri, “Understanding the difference between wireless encryption protocols.” http://blogs.cisco.com/smallbusiness/understanding-the-difference-between-wireless-encryption-protocols.

[9] J. J. Garcia-Luna-Aceves, “Content-centric networking.” https://www.ietf.org/proceedings/65/slides/DTNRG-12.pdf.

[10] C. Perkins, E. Belding-Royer, and S. Das, “Request for comments: 3561,” 2003.

[11] C. Perkins and P. Bhagwat, “Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers,” 1994.

[12] Oracle, “Class thread.” http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html.

[13] Oracle, “Providing constructors for your classes.” https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html.

[14] Oracle, “Class inputstream.” https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html.

[15] Oracle, “Class outputstream.” https://docs.oracle.com/javase/7/docs/api/java/io/OutputStream.html.

[16] G. Malkin, "Request for comments: 2453," 1998.

[17] Oracle, "Interface map." `https://docs.oracle.com/javase/7/docs/api/java/util/Map.html`.

[18] Oracle, "Arrays." `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html`.

[19] A. Developers, "Api guides: Bluetooth connectivity." `https://developer.android.com/guide/topics/connectivity/bluetooth.html#ConnectingDevices`.

[20] A. Developers, "Bluetoothclass.device." `https://developer.android.com/reference/android/bluetooth/BluetoothClass.Device.html#PHONE_SMART`.

[21] A. Developers, "Message." `https://developer.android.com/reference/android/os/Message.html`.

[22] A. Developers, "Edittext." `https://developer.android.com/reference/android/widget/EditText.html`.

[23] A. Developers, "Webview." `https://developer.android.com/reference/android/webkit/WebView.html`.

[24] Oracle, "Random." `https://docs.oracle.com/javase/8/docs/api/java/util/Random.html`.

[25] A. Developers, "Asynctask." `https://developer.android.com/reference/android/os/AsyncTask.html`.

[26] Oracle, "Bufferedinputstream." `https://docs.oracle.com/javase/7/docs/api/java/io/BufferedInputStream.html`.

[27] S. . HimalayanTahr, "Save webpage for offline use and invoke the same android." `https://stackoverflow.com/questions/17484115/save-webpage-for-offline-use-and-invoke-the-same-android`.

[28] B. S. I. Group, "How it works — bluetooth technology website." `https://www.bluetooth.com/what-is-bluetooth-technology/how-it-works`.

[29] W.-F. Alliance, "How far does a wi-fi direct connection travel?." `http://www.wi-fi.org/knowledge-center/faq/how-far-does-a-wi-fi-direct-connection-travel`.

[30] A. Boukerche, "Algorithms and protocols for wireless, mobile ad hoc networks," 2009.

Appendix A

Application code

A.1 InitActivity.java

```
package com.example.falcato.btrouting;

import ...

public class InitActivity extends Activity {

    private static final String TAG = "InitActivity";
    Button goButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_init);
    }

    @Override
    protected void onStart (){
        super.onStart();

        // Check if device has an Internet connection
        new NetworkCheck().execute();

        goButton = (Button) findViewById(R.id.button);
        goButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent ( InitActivity.this, BtActivity.class );
                startActivity(intent);
            }
        });
    }
    private class NetworkCheck extends AsyncTask<Void, Void, Boolean> {

        @Override
        protected Boolean doInBackground(Void... params) {
            Log.i(TAG, "hasActiveInternetConnection()");
            try {
                HttpURLConnection urlc = (HttpURLConnection) (
                        new URL("http://clients3.google.com/generate_204").openConnection());
                urlc.setRequestProperty("User-Agent", "Test");
                urlc.setRequestProperty("Connection", "close");
                urlc.setConnectTimeout(1500);
                urlc.connect();
                return (urlc.getResponseCode() == 204 && urlc.getContentLength() == 0);
            } catch (IOException e) {
                Log.e(TAG, "Error checking internet connection", e);
            }
            return false;
        }

        @Override
        protected void onPostExecute(Boolean result) {
            ((RoutingApp) getApplicationContext()).setHasNet(result);
        }
    }
}
```

A.2 BtActivity.java

```
package com.example.falcato.btrouting;

import ...

public class BtActivity extends Activity {

    // Debugging
    private static final String TAG = "BluetoothActivity";
    private static final boolean D = true;

    Button goButton;
    EditText mEdit;
    WebView mWebview;

    // Message types sent from the BluetoothChatService Handler
    public static final int MESSAGE_STATE_CHANGE = 1;
    public static final int MESSAGE_READ = 2;
    public static final int MESSAGE_WRITE = 3;
    public static final int MESSAGE_DEVICE_NAME = 4;
    public static final int MESSAGE_TOAST = 5;
    public static final int FILE_READ = 6;
    public static final int FILE_WRITE = 7;

    // Key names received from the BluetoothService Handler
    public static final String DEVICE_NAME = "device_name";
    public static final String TOAST = "toast";

    // Intent request codes
    private static final int REQUEST_ENABLE_BT = 3;

    // Name of the connected device
    private String mConnectedDeviceName = null;
    // List of peer devices
    private ArrayList<BluetoothDevice> peers = null;
    // String buffer for outgoing messages
    private StringBuffer mOutStringBuffer;
    // Local Bluetooth adapter
    private BluetoothAdapter mBluetoothAdapter = null;
    // Member object for the chat services
    private BluetoothService mService = null;
    // Check if peer discovery is finished
    private boolean discoveryFinished = false;
    // Save the message ID
    private int msgID = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if(D) Log.e(TAG, "+++ ON CREATE +++");

        // Set up the window layout
        setContentView(R.layout.activity_bt);

        // Get local Bluetooth adapter
        mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

        // Initialize peers array
        peers = new ArrayList<>();

        // If the adapter is null, then Bluetooth is not supported
        if (mBluetoothAdapter == null) {
            Toast.makeText(this, "Bluetooth is not available", Toast.LENGTH_LONG).show();
            finish();
```

```
            return;
        }

    }

    @Override
    public void onStart() {
        super.onStart();
        if(D) Log.e(TAG, "++ ON START ++");

        // If BT is not on, request that it be enabled.
        if (!mBluetoothAdapter.isEnabled()) {
            Log.e(TAG, "mBluetoothAdapter not enabled");
            Intent enableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableIntent, REQUEST_ENABLE_BT);

        } else {
            Log.e(TAG, "mBluetoothAdapter enabled");
        }

        // Register for broadcasts when a device is discovered
        IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
        this.registerReceiver(mReceiver, filter);

        // Register for broadcasts when discovery has finished
        filter = new IntentFilter(BluetoothAdapter.ACTION_DISCOVERY_FINISHED);
        this.registerReceiver(mReceiver, filter);

        mWebview = (WebView) findViewById(R.id.webView);
        mWebview.getSettings().setJavaScriptEnabled(true);
        mWebview.getSettings().setAllowFileAccess(true);
        mWebview.getSettings().setDomStorageEnabled(true);
        mWebview.getSettings().setAllowContentAccess(true);
        mWebview.getSettings().setAllowFileAccessFromFileURLs(true);
        mWebview.getSettings().setJavaScriptCanOpenWindowsAutomatically(true);

        // Make this device discoverable
        ensureDiscoverable();

        // update route table and start discovery
        // If device has net
        if(((RoutingApp)getApplicationContext()).getHasNet())
            ((RoutingApp)getApplicationContext()).updateRouteTable
                ("ADV;" + getOwnMAC() + ";0");
        // Otherwise infinite number of hops
        else
            ((RoutingApp)getApplicationContext()).updateRouteTable
```

```
                ("ADV;" + getOwnMAC() + ";16");

    // Remove comments
    doDiscovery();
    setupBluetoothService();

    goButton = (Button) findViewById(R.id.buttonGo);
    mEdit = (EditText)findViewById(R.id.editText);
    goButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // Test
            //getPage(mEdit.getText().toString());

            // Remove comments
            sendRequest(true, -1, mEdit.getText().toString());
        }
    });
}

@Override
public void onDestroy() {
    super.onDestroy();
    // Stop the Bluetooth chat services
    if (mService != null) mService.stop();
    if(D) Log.e(TAG, "--- ON DESTROY ---");
    unregisterReceiver(mReceiver);
}

private String getOwnMAC () {
    if (Build.VERSION.SDK_INT > 22)
        return android.provider.Settings.Secure.getString(this.getContentResolver(),
            "bluetooth_address");
    else
        return mBluetoothAdapter.getAddress();
}

private void ensureDiscoverable() {
    if(D) Log.i(TAG, "ensure discoverable");
    if (mBluetoothAdapter.getScanMode() !=
            BluetoothAdapter.SCAN_MODE_CONNECTABLE_DISCOVERABLE) {
        Intent discoverableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
        discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 0);
        startActivity(discoverableIntent);
    }
}
```

```
private void doDiscovery() {
    if (D) Log.i(TAG, "doDiscovery()");
    discoveryFinished = false;

    // If we're already discovering, stop it
    if (mBluetoothAdapter.isDiscovering()) {
        mBluetoothAdapter.cancelDiscovery();
    }

    mBluetoothAdapter.startDiscovery();
    Toast.makeText(getApplicationContext(), "Starting discovery. Please wait...",
            Toast.LENGTH_SHORT).show();
}

private void setupBluetoothService() {
    Log.i(TAG, "setupBluetoothService()");
    // Initialize the BluetoothChatService to perform bluetooth connections
    mService = new BluetoothService(this, mHandler);

    // Initialize the buffer for outgoing messages
    mOutStringBuffer = new StringBuffer("");
}

private void advertisePeers() {
    Log.i(TAG, "advertisePeers()");
    for (BluetoothDevice peer : peers){
        Log.i(TAG, "Peer class: " + peer.getBluetoothClass().getDeviceClass());
        // Check if peer is a cell phone
        if (peer.getBluetoothClass().getDeviceClass() == 524){
            //Debug
            //if(!peer.getName().contains(";HUAWEI P8 lite")) {
            // Check if peer is using app
            if (peer.getName().contains(";")) {
                Log.i(TAG, "Will advertise to: " + peer.getName());
                mService.connect(peer);

                // Wait until connection is done
                while (mService.getState() != BluetoothService.STATE_CONNECTED) {
                    if (mService.getState() == BluetoothService.STATE_LISTEN) {
                        Log.e(TAG, "Failed to connect, listening");
                        mService.start();
                        break;
                    }
                }

                // Device is connected, advertise
                int nrHops = ((RoutingApp) getApplicationContext()).getMinHop() + 1;
                sendMessage("ADV;" + getOwnMAC() + ";" + nrHops);

                // Wait until connection is finished
                long initTime = System.currentTimeMillis();
                while (mService.getState() != BluetoothService.STATE_LISTEN) {
                    if (System.currentTimeMillis() - initTime > 5000) {
                        break;
                    }
                }
            }
            //Debug
            //}
        }
    }
}

private void sendRequest(boolean owner, int msgID, String message) {
    Log.i(TAG, "sendRequest()");

    // Get the address of the next hop
    BluetoothDevice nextHop = mBluetoothAdapter.getRemoteDevice(
            ((RoutingApp)getApplicationContext()).getNextHop());
    // In case there is no next hop
    if (nextHop.getAddress() == null){
        Log.i(TAG, "There is no next hop");
        sendFail(msgID);
        return;
    }

    Log.i(TAG, "Will send request to: " + nextHop.getAddress());
    mService.connect(nextHop);

    // Wait until connection is done
    for (int aux = 0; mService.getState() != BluetoothService.STATE_CONNECTED; aux ++){
        if (mService.getState() == BluetoothService.STATE_LISTEN){
            Log.e(TAG, "Failed to connect, listening");
            mService.start();
            // Failed to connect send fail notification
            sendFail(msgID);
            return;
        }
    }

    // If device is the one sending the request
    if (owner) {
        // Device is connected, send request
        Random rn = new Random();
```

```
            int newMsgID = rn.nextInt();
            // Message -> RQT ; Message ID ; Own MAC ; Data
            sendMessage("RQT;" + newMsgID + ";" + getOwnMAC() + ";" + message);
            // Update the response table with own MAC to know when the response is received
            ((RoutingApp)getApplicationContext()).updateRspTable(newMsgID, getOwnMAC());
        // If device is forwarding the request
        }else{
            // Message -> RQT ; Message ID ; Own MAC ; Data
            sendMessage("RQT;" + msgID + ";" + getOwnMAC() + ";" + message);
        }
    }

    private void sendResponse(int msgID) {
        Log.i(TAG, "sendResponse(int msgID) " + msgID);
        // Retrieve the requester's MAC
        String nextHopMAC = ((RoutingApp)getApplicationContext()).getRspHop(msgID);
        // If message ID exists in the response table
        if (nextHopMAC != null){
            // Get the address of the next hop
            BluetoothDevice nextHopDevice = mBluetoothAdapter.getRemoteDevice(nextHopMAC);
            Log.i(TAG, "Will send response to: " + nextHopDevice.getAddress());
            mService.connect(nextHopDevice);

            // Wait until connection is done
            for (int aux = 0; mService.getState() != BluetoothService.STATE_CONNECTED; aux ++){
                if (mService.getState() == BluetoothService.STATE_LISTEN){
                    Log.e(TAG, "Failed to connect, listening");
                    mService.start();
                    return;
                }
            }
            // Send the response message
            sendMessage("RSP;" + msgID);
        }else{
            Log.i(TAG, "MAC for requested Message ID not found");
        }
    }

    private void sendFail(int msgID) {
        Log.i(TAG, "sendFail(int msgID) " + msgID);
        // Retrieve the requester's MAC
        String nextHopMAC = ((RoutingApp)getApplicationContext()).getRspHop(msgID);
        // If message ID exists in the response table
        if (nextHopMAC != null){
            // Get the address of the next hop
            BluetoothDevice nextHopDevice = mBluetoothAdapter.getRemoteDevice(nextHopMAC);
            Log.i(TAG, "Will send fail notification to: " + nextHopDevice.getAddress());
            mService.connect(nextHopDevice);

            // Wait until connection is done
            for (int aux = 0; mService.getState() != BluetoothService.STATE_CONNECTED; aux ++){
                if (mService.getState() == BluetoothService.STATE_LISTEN){
                    Log.e(TAG, "Failed to connect, listening");
                    mService.start();
                    return;
                }
            }
            // Send the response message
            sendMessage("FAIL;" + msgID);
        }else{
            Log.i(TAG, "MAC for requested Message ID not found");
        }
    }

    private void sendMessage(String message) {
        // Check that we're actually connected before trying anything
        if (mService.getState() != BluetoothService.STATE_CONNECTED) {
            Log.e(TAG, "Not connected, can't send message");
            return;
        }

        // Check that there's actually something to send
        if (message.length() > 0) {
            // Get the message bytes and tell the BluetoothChatService to write
            byte[] send = message.getBytes();
            mService.write(send);

            // Reset out string buffer to zero and clear the edit text field
            mOutStringBuffer.setLength(0);
        }
    }

    private void sendFile() {
        Log.i(TAG, "sendFile()");

        File file = new File(getFilesDir() + "file.mht");
        int size = (int) file.length();
        byte[] bytes = new byte[size];

        try {
            BufferedInputStream buf = new BufferedInputStream(new FileInputStream(file));
            buf.read(bytes, 0, bytes.length);
            buf.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        mService.writeFile(bytes);

        Log.i(TAG, "sent file with " + bytes.length + " bytes in: " + getFilesDir() + "file.mht");
    }

    private void analyzeMessage(String message) {

        // Advertising message
        if (message.contains("ADV")){
            // Check if new shortest path was found
            if (Integer.parseInt(message.split(";")[2]) <
                    ((RoutingApp)getApplicationContext()).getMinHop()){
                Log.i(TAG, "New best path will advertise");
                // If so, update table, initiate discovery and advertise new path
                ((RoutingApp)getApplicationContext()).updateRouteTable(message);
                discoveryFinished = false;
                doDiscovery();
            }else{
                Log.i(TAG, "New path is not the best will not advertise");
                // Otherwise update table and continue listening
                ((RoutingApp)getApplicationContext()).updateRouteTable(message);
            }

        // Request message
        }else if (message.contains("RQT")){
            // Save the message ID
            msgID = Integer.parseInt(message.split(";")[1]);
            // Update the response table
            ((RoutingApp)getApplicationContext()).updateRspTable(
                    Integer.parseInt(message.split(";")[1]), message.split(";")[2]);

            // If device is not connected to the Internet
            if (!((RoutingApp)getApplicationContext()).getHasNet()){
                // Forward the request
                sendRequest(false, Integer.parseInt(message.split(";")[1]), message.split(";")[3]);
            // If it is connected, fetch the web page and send the response
            }else{
                getPage(message.split(";")[3], Integer.parseInt(message.split(";")[1]));
            }

        // Response message
        }else if (message.contains("RSP")){

            // Save message ID to know the file name
            msgID = Integer.parseInt(message.split(";")[1]);

            // If device is the destination
            if (((RoutingApp)getApplicationContext()).getRspHop(Integer.parseInt(
                    message.split(";")[1])).equals(getOwnMAC())){
                // Request was successfully sent and response was received
                Log.i(TAG, "Received my response.");

            // Otherwise forward response to destination
            }else{
                // Forward the response
                Log.i(TAG, "Not the final destination will forward response.");
            }
        }else if (message.contains("FAIL")){
            // Save message ID to know the file name
            msgID = Integer.parseInt(message.split(";")[1]);

            // If device is the destination
            if (((RoutingApp)getApplicationContext()).getRspHop(Integer.parseInt(
                    message.split(";")[1])).equals(getOwnMAC())){
                // Request was successfully sent but response failed
                Toast.makeText(getApplicationContext(), "Unfortunately there was a problem along " +
                                "the path. Please try again later.",
                        Toast.LENGTH_SHORT).show();

                // Otherwise forward response to destination
            }else{
                // Forward the failed request
                Log.i(TAG, "Not the final destination will forward fail notification.");
                sendFail(msgID);
            }
        }
    }

    private void getPage(String url, final int msgID){
        Log.i(TAG, "getPage(String url, final int messageID)");
        //mWebview.setVisibility(View.VISIBLE);
        mWebview.setWebViewClient(new WebViewClient() {
              public void onPageFinished(WebView view, String url) {
                  // Fix to load all pages and not send 0 bytes

                  File file = new File(getFilesDir() + "file.mht");

                  try {
                      file.createNewFile();
                  } catch (IOException e) {
                      e.printStackTrace();
                }

                view.saveWebArchive(getFilesDir() + "file.mht");
                //mWebview.setVisibility(View.INVISIBLE);

                waitForWebPage dloader = new waitForWebPage();
                dloader.execute();
            }
        });
        mWebview.loadUrl("https://" + url);
    }

    @Override
    public void onBackPressed(){
        // If a page is being displayed, let the user enter a new URL
        if (mWebview.getVisibility() == View.VISIBLE) {
            mWebview.setVisibility(View.INVISIBLE);
        // Otherwise act normally and go back
        }else{
            super.onBackPressed();
        }
    }

    /* --- Logic to display the file --- */

    private void loadPage(){
        Log.i(TAG, "loading page...");

        mWebview.getSettings().setCacheMode( WebSettings.LOAD_CACHE_ELSE_NETWORK );
        mWebview.setVisibility(View.VISIBLE);
        // #1 try to send a re-request
        mWebview.setWebViewClient(new WebViewClient(){
            @Override
            public void onReceivedError(WebView view, WebResourceRequest request,
                                        WebResourceError error) {
                if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
                    Log.i(TAG, "No Internet connection, sending a re-request of: " +
                            request.getUrl().toString().split("://")[1]);
                    view.loadUrl("about:blank");
                    sendRequest(true, -1, request.getUrl().toString().split("://")[1]);
                }
            }
        });
        mWebview.setWebChromeClient(new WebChromeClient());

        if (Build.VERSION.SDK_INT < 22) {
            loadArchive();
        } else {
            mWebview.loadUrl("file:///" + getFilesDir() + "file.mht");
        }

        Log.i(TAG, "Loaded page in: file:///" + getFilesDir() + "file.mht");
    }

    private void loadArchive(){
        String rawData = null;
        try {
            rawData = getStringFromFile(getFilesDir() + "file.mht");
        } catch (Exception e) {
            e.printStackTrace();
        }
        mWebview.loadDataWithBaseURL(null, rawData, "application/x-webarchive-xml", "UTF-8", null);
    }

    public String getStringFromFile (String filePath) throws Exception {
        File fl = new File(filePath);
        FileInputStream fin = new FileInputStream(fl);
        String ret = convertStreamToString(fin);
        //Make sure you close all streams.
        fin.close();
        return ret;
    }

    public  String convertStreamToString(InputStream is) throws Exception {
        BufferedReader reader = new BufferedReader(new InputStreamReader(is));
        StringBuilder sb = new StringBuilder();
        String line = null;
        while ((line = reader.readLine()) != null) {
            sb.append(line).append("\n");
        }
        reader.close();
        return sb.toString();
    }

    /* --- End of logic --- */

    private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();

        // When discovery finds a device
        if (BluetoothDevice.ACTION_FOUND.equals(action)) {
            Log.i(TAG, "Found a device.");
```

```
            // Get the BluetoothDevice object from the Intent
            BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            // If new peer not counted already add it to peer list
            if (!peers.contains(device)) {
                peers.add(device);
            }
            // When discovery is finished, change the Activity title
        } else if (BluetoothAdapter.ACTION_DISCOVERY_FINISHED.equals(action)) {
            if(!discoveryFinished) {
                TextView peerText = (TextView) findViewById(R.id.textViewPeers);
                Log.i(TAG, "Discovery finished.");
                // Stop the discovery
                mBluetoothAdapter.cancelDiscovery();
                discoveryFinished = true;
                Toast.makeText(getApplicationContext(), "Discovery finished",
                        Toast.LENGTH_SHORT).show();
                Log.i(TAG, "Peers found: " + peers.toString());
                peerText.setText("Peers found: " + peers.toString());
                advertisePeers();
            }
        }
        }
    };

    // The Handler that gets information back from the BluetoothChatService
    private final Handler mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MESSAGE_STATE_CHANGE:
                    if(D) Log.i(TAG, "MESSAGE_STATE_CHANGE: " + msg.arg1);
                    switch (msg.arg1) {
                        case BluetoothService.STATE_CONNECTED:
                            Log.e(TAG, "Status: connected");
                            break;
                        case BluetoothService.STATE_CONNECTING:
                            Log.e(TAG, "Status: connecting");
                            break;
                        case BluetoothService.STATE_LISTEN:
                            Log.e(TAG, "Status: listen");
                            break;
                        case BluetoothService.STATE_NONE:
                            Log.e(TAG, "Status: none");
                            break;
                    }
                    break;
                case MESSAGE_WRITE:
```

```
    byte[] writeBuf = (byte[]) msg.obj;
    // construct a string from the buffer
    String writeMessage = new String(writeBuf);
    // handle sent message
    Log.i(TAG, "Sent a new message: " + writeMessage);

    // If a response was sent follow with the corresponding file
    if (writeMessage.contains("RSP;")){
        sendFile();
    }
    break;
case MESSAGE_READ:
    byte[] readBuf = (byte[]) msg.obj;
    // construct a string from the valid bytes in the buffer
    String readMessage = new String(readBuf, 0, msg.arg1);
    // handle received message
    TextView recvText = (TextView) findViewById(R.id.textViewReceived);
    Log.i(TAG, "Received a new message: " + readMessage);
    recvText.setText(recvText.getText() + "\n" + readMessage);

    // Restart the Bluetooth Service
    if (!readMessage.contains("RSP;")) {
        mService.start();
    }else {
        mService.fileReady = true;
    }
    analyzeMessage(readMessage);
    break;
case MESSAGE_DEVICE_NAME:
    // save the connected device's name
    mConnectedDeviceName = msg.getData().getString(DEVICE_NAME);
    Log.i(TAG, "Connected to " + mConnectedDeviceName);
    break;
case MESSAGE_TOAST:
    break;

case FILE_READ:
    Log.i(TAG, "Received a new file");

    byte[] readFileBuf = (byte[]) msg.obj;
    try {

        File file = new File(getFilesDir() + "file" + msgID + ".mht");
        if (!file.exists()) {
            file.createNewFile();
        }
        FileOutputStream stream = new FileOutputStream(
                getFilesDir() + "file.mht", false);
        stream.write(readFileBuf);
        stream.flush();
        stream.close();

        Log.i(TAG, "Saved the file in: " + getFilesDir() + "file.mht");
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }

    // Restart the Bluetooth Service
    mService.start();
    mService.fileReady = false;

    // If I am the destination
    if (((RoutingApp)getApplicationContext()).getRspHop(msgID).equals(getOwnMAC()))
        // Display the page
        loadPage();
    }else{
        sendResponse(msgID);
    }

    break;

case FILE_WRITE:
    byte[] writeFileBuf = (byte[]) msg.obj;
    try {

        File file = new File(getFilesDir() + "file.mht");
        if (!file.exists()) {
            file.createNewFile();
        }

        FileOutputStream stream = new FileOutputStream(
                getFilesDir() + "file.mht", false);
        stream.write(writeFileBuf);
        stream.flush();
        stream.close();

        Log.i(TAG, "Saved the file in: " + getFilesDir() + "file.mht");
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
```

```
                }
                // Display the page
                loadPage();
                break;
            }
        }
    };

    private class waitForWebPage extends AsyncTask<Void, Void, Void>{
        File file = new File(getFilesDir() + "file.mht");
        @Override
        protected Void doInBackground(Void... params) {
            Log.i(TAG, "Downloading webpage...");
            while(!(file.length() > 0)){
            }
            return null;
        }

        @Override
        protected void onPostExecute(Void result){
            Log.i(TAG, "saved web archive in: " + getFilesDir() + "file.mht with " +
                    file.length() + " bytes");
            sendResponse(msgID);
        }
    }
}
```

A.3 RoutingApp.java

```
package com.example.falcato.btrouting;

import ...

public class RoutingApp extends Application {

    private static final String TAG = "RoutingApp";

    private boolean hasNet;
    // Table with routing hops
    public Map<String, Integer> routeTable = new HashMap<>();
    // Table with MACs corresponding to message ID's
    public Map<Integer, String> rspTable = new HashMap<>();

    public boolean getHasNet () {
        Log.i(TAG, "getHasNet()");
    return hasNet;
}

public void setHasNet (boolean hasNet) {
    Log.i(TAG, "setHasNet() " + hasNet);
    this.hasNet = hasNet;
}

public void updateRouteTable (String msg) {
    Log.i(TAG, "updateRouteTable()");
    String dest = msg.split(";")[1];
    int hops = Integer.parseInt(msg.split(";")[2]);

    routeTable.put(dest, hops);
}

public int getMinHop () {
    Log.i(TAG, "getMinHop()");
    int minHop;
    try {
        minHop = Collections.min(routeTable.values());
    }catch (NoSuchElementException e){
        Log.e(TAG, e.toString());
        minHop = 16;
    }

    Log.i(TAG, "Minimal nr of hops is: " + minHop);
    return minHop;
}

public String getNextHop () {
    Log.i(TAG, "getNextHop()");
    int minHop = getMinHop();
    if (minHop == 16)
        return null;
    else
        return getKeyFromValue(routeTable, minHop);
}

private String getKeyFromValue (Map<String, Integer> hm, Integer value) {
    Log.i(TAG, "getKeyFromValue()");
    for (String key : hm.keySet()) {
        if (hm.get(key).equals(value)) {
            return key;
        }
    }
    return null;
    }

    public void updateRspTable (int msgID, String MAC) {
        Log.i(TAG, "updateRspTable()");
        rspTable.put(msgID, MAC);
        Log.i(TAG, "Updated message table: " + rspTable.toString());
    }

    public String getRspHop (int msgID) {
        Log.i(TAG, "getRspHop()");
        if (rspTable.containsKey(msgID))
            return rspTable.get(msgID);
        else
            return null;
    }
}
```

A.4 BluetoothService.java

```
package com.example.falcato.btrouting;

import ...

public class BluetoothService {
    // Debugging
    private static final String TAG = "BluetoothService";
    private static final boolean D = true;

    // Name for the SDP record when creating server socket
    private static final String NAME_INSECURE = "BluetoothRouteInsecure";

    // Unique UUID for this application
    private static final UUID MY_UUID_INSECURE =
            UUID.fromString("8ce255c0-200a-11e0-ac64-0800200c9a66");

    // Member fields
    private final BluetoothAdapter mAdapter;
    private final Handler mHandler;
    private AcceptThread mInsecureAcceptThread;
    private ConnectThread mConnectThread;
    private ConnectedThread mConnectedThread;
    private int mState;
    boolean fileReady = false;

    // Constants that indicate the current connection state
    public static final int STATE_NONE = 0;       // we're doing nothing
public static final int STATE_LISTEN = 1;     // now listening for incoming connections
public static final int STATE_CONNECTING = 2; // now initiating an outgoing connection
public static final int STATE_CONNECTED = 3;  // now connected to a remote device


/**
 * Constructor. Prepares a new BluetoothChat session.
 * @param context  The UI Activity Context
 * @param handler  A Handler to send messages back to the UI Activity
 */
public BluetoothService(Context context, Handler handler) {
    mAdapter = BluetoothAdapter.getDefaultAdapter();
    mState = STATE_NONE;
    mHandler = handler;
}

/**
 * Set the current state of the chat connection
 * @param state  An integer defining the current connection state
 */
private synchronized void setState(int state) {
    if (D) Log.i(TAG, "setState() " + mState + " -> " + state);
    mState = state;

    // Give the new state to the Handler so the UI Activity can update
    mHandler.obtainMessage(BtActivity.MESSAGE_STATE_CHANGE, state, -1).sendToTarget();
}

/**
 * Return the current connection state. */
public synchronized int getState() {
    return mState;
}

/**
 * Start the chat service. Specifically start AcceptThread to begin a
 * session in listening (server) mode. Called by the Activity onResume() */
public synchronized void start() {
    if (D) Log.i(TAG, "start");

    // Cancel any thread attempting to make a connection
    if (mConnectThread != null) {mConnectThread.cancel(); mConnectThread = null;}

    // Cancel any thread currently running a connection
    if (mConnectedThread != null) {mConnectedThread.cancel(); mConnectedThread = null;}

    setState(STATE_LISTEN);
```

```
        // Start the thread to listen on a BluetoothServerSocket
        if (mInsecureAcceptThread == null) {
            mInsecureAcceptThread = new AcceptThread();
            mInsecureAcceptThread.start();
        }
    }

    /**
     * Start the ConnectThread to initiate a connection to a remote device.
     * @param device  The BluetoothDevice to connect
     */
    public synchronized void connect(BluetoothDevice device) {
        if (D) Log.i(TAG, "connect to: " + device);

        // Cancel any thread attempting to make a connection
        if (mState == STATE_CONNECTING) {
            if (mConnectThread != null) {mConnectThread.cancel(); mConnectThread = null;}
        }

        // Cancel any thread currently running a connection
        if (mConnectedThread != null) {mConnectedThread.cancel(); mConnectedThread = null;}

        // Start the thread to connect with the given device
        mConnectThread = new ConnectThread(device);
        mConnectThread.start();
        setState(STATE_CONNECTING);
    }

    /**
     * Start the ConnectedThread to begin managing a Bluetooth connection
     * @param socket  The BluetoothSocket on which the connection was made
     * @param device  The BluetoothDevice that has been connected
     */
    public synchronized void connected(BluetoothSocket socket, BluetoothDevice
            device) {
        if (D) Log.i(TAG, "connected");

        // Cancel the thread that completed the connection
        if (mConnectThread != null) {mConnectThread.cancel(); mConnectThread = null;}

        // Cancel any thread currently running a connection
        if (mConnectedThread != null) {mConnectedThread.cancel(); mConnectedThread = null;}

        // Cancel the accept thread because we only want to connect to one device
        if (mInsecureAcceptThread != null) {
            mInsecureAcceptThread.cancel();
            mInsecureAcceptThread = null;
        }

        // Start the thread to manage the connection and perform transmissions
        mConnectedThread = new ConnectedThread(socket);
        mConnectedThread.start();

        // Send the name of the connected device back to the UI Activity
        Message msg = mHandler.obtainMessage(BtActivity.MESSAGE_DEVICE_NAME);
        Bundle bundle = new Bundle();
        bundle.putString(BtActivity.DEVICE_NAME, device.getName());
        msg.setData(bundle);
        mHandler.sendMessage(msg);

        setState(STATE_CONNECTED);
    }

    /**
     * Stop all threads
     */
    public synchronized void stop() {
        if (D) Log.i(TAG, "stop");

        if (mConnectThread != null) {
            mConnectThread.cancel();
            mConnectThread = null;
        }

        if (mConnectedThread != null) {
            mConnectedThread.cancel();
            mConnectedThread = null;
        }

        if (mInsecureAcceptThread != null) {
            mInsecureAcceptThread.cancel();
            mInsecureAcceptThread = null;
        }
        setState(STATE_NONE);
    }

    /**
     * Write to the ConnectedThread in an unsynchronized manner
     * @param out The bytes to write
     * @see ConnectedThread#write(byte[])
     */
    public void write(byte[] out) {
        // Create temporary object
        ConnectedThread r;
        // Synchronize a copy of the ConnectedThread
        synchronized (this) {
            if (mState != STATE_CONNECTED) return;
            r = mConnectedThread;
        }
        // Perform the write unsynchronized
        r.write(out);
    }

    public void writeFile(byte[] out) {
        Log.i(TAG, "writeFile(byte[] out)");
        // Create temporary object
        ConnectedThread r;
        // Synchronize a copy of the ConnectedThread
        synchronized (this) {
            if (mState != STATE_CONNECTED) return;
            r = mConnectedThread;
        }
        // Perform the write unsynchronized
        r.writeFile(out);
    }

    /**
     * Indicate that the connection attempt failed and notify the UI Activity.
     */
    private void connectionFailed() {
        Log.i(TAG, "connectionFailed()");
        // Send a failure message back to the Activity
        Message msg = mHandler.obtainMessage(BtActivity.MESSAGE_TOAST);
        Bundle bundle = new Bundle();
        bundle.putString(BtActivity.TOAST, "Unable to connect device");
        msg.setData(bundle);
        mHandler.sendMessage(msg);

        // Start the service over to restart listening mode
        BluetoothService.this.start();
    }

    /**
     * Indicate that the connection was lost and notify the UI Activity.
     */
    private void connectionLost() {
        Log.i(TAG, "connectionLost()");
        // Send a failure message back to the Activity
        Message msg = mHandler.obtainMessage(BtActivity.MESSAGE_TOAST);
        Bundle bundle = new Bundle();
        bundle.putString(BtActivity.TOAST, "Device connection was lost");
        msg.setData(bundle);
        mHandler.sendMessage(msg);

        // Start the service over to restart listening mode
        BluetoothService.this.start();
    }

    /**
     * This thread runs while listening for incoming connections. It behaves
     * like a server-side client. It runs until a connection is accepted
     * (or until cancelled).
     */
    private class AcceptThread extends Thread {
        // The local server socket
        private final BluetoothServerSocket mmServerSocket;

        public AcceptThread() {
            BluetoothServerSocket tmp = null;

            // Create a new listening server socket
            try {
                tmp = mAdapter.listenUsingInsecureRfcommWithServiceRecord(
                        NAME_INSECURE, MY_UUID_INSECURE);
            } catch (IOException e) {
                Log.e(TAG, "Listen() failed", e);
            }
            mmServerSocket = tmp;
        }

        public void run() {
            if (D) Log.i(TAG, "BEGIN mAcceptThread" + this);
            setName("AcceptThread");

            BluetoothSocket socket = null;

            // Listen to the server socket if we're not connected
            while (mState != STATE_CONNECTED) {
                try {
                    // This is a blocking call and will only return on a
                    // successful connection or an exception
                    socket = mmServerSocket.accept();
                } catch (IOException e) {
                    Log.e(TAG, "Accept() failed", e);
                    break;
                }
                // If a connection was accepted
                if (socket != null) {
                    synchronized (BluetoothService.this) {
                        switch (mState) {
                            case STATE_LISTEN:
                            case STATE_CONNECTING:
                                // Situation normal. Start the connected thread.
                                connected(socket, socket.getRemoteDevice());
                                break;
                            case STATE_NONE:
                            case STATE_CONNECTED:
                                // Either not ready or already connected. Terminate new socket.
                                try {
                                    socket.close();
                                } catch (IOException e) {
                                    Log.e(TAG, "Could not close unwanted socket", e);
                                }
                                break;
                        }
                    }
                }
            }
            if (D) Log.i(TAG, "END mAcceptThread");

        }

        public void cancel() {
            if (D) Log.i(TAG, "Cancel " + this);
            try {
                mmServerSocket.close();
            } catch (IOException e) {
                Log.e(TAG, "Close() of server failed", e);
            }
        }
    }

    /**
     * This thread runs while attempting to make an outgoing connection
     * with a device. It runs straight through; the connection either
     * succeeds or fails.
     */
    private class ConnectThread extends Thread {
        private final BluetoothSocket mmSocket;
        private final BluetoothDevice mmDevice;

        public ConnectThread(BluetoothDevice device) {
            mmDevice = device;
            BluetoothSocket tmp = null;

            // Get a BluetoothSocket for a connection with the
            // given BluetoothDevice
            try {

                tmp = device.createInsecureRfcommSocketToServiceRecord(
                        MY_UUID_INSECURE);
            } catch (IOException e) {
                Log.e(TAG, "Create() failed", e);
            }
            mmSocket = tmp;
        }

        public void run() {
            Log.i(TAG, "BEGIN mConnectThread");
            setName("ConnectThread");

            // Always cancel discovery because it will slow down a connection
            mAdapter.cancelDiscovery();

            // Make a connection to the BluetoothSocket
            try {
                // This is a blocking call and will only return on a
                // successful connection or an exception
                mmSocket.connect();
            } catch (IOException e) {
                // Close the socket
                try {
                    mmSocket.close();
                } catch (IOException e2) {
                    Log.e(TAG, "Unable to close() socket during connection failure", e2);
                }
                connectionFailed();
                return;
            }

            // Reset the ConnectThread because we're done
            synchronized (BluetoothService.this) {
                mConnectThread = null;
            }

            // Start the connected thread
            connected(mmSocket, mmDevice);
        }

        public void cancel() {
            try {
                mmSocket.close();
            } catch (IOException e) {
                Log.e(TAG, "Close() of connect socket failed", e);
            }
        }
}

/**
 * This thread runs during a connection with a remote device.
 * It handles all incoming and outgoing transmissions.
 */
private class ConnectedThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final InputStream mmInStream;
    private final OutputStream mmOutStream;

    public ConnectedThread(BluetoothSocket socket) {
        Log.i(TAG, "Create ConnectedThread");
        mmSocket = socket;
        InputStream tmpIn = null;
        OutputStream tmpOut = null;

        // Get the BluetoothSocket input and output streams
        try {
            tmpIn = socket.getInputStream();
            tmpOut = socket.getOutputStream();

        } catch (IOException e) {
            Log.e(TAG, "temp sockets not created", e);
        }

        mmInStream = tmpIn;
        mmOutStream = tmpOut;
    }

    public void run() {
        Log.i(TAG, "BEGIN mConnectedThread");
        byte[] buffer = new byte[8192];
        ByteArrayOutputStream output = new ByteArrayOutputStream();
        int bytes;

        // Keep listening to the InputStream while connected
        //while (no notification received)
        while (true) {
            try {
                // Read from the InputStream
                bytes = mmInStream.read(buffer);

                if (!fileReady) {
                    // Send the obtained bytes to the UI Activity
                    Log.e(TAG, "nr of bytes: " + bytes);
                    mHandler.obtainMessage(BtActivity.MESSAGE_READ, bytes, -1, buffer)
                            .sendToTarget();
                }else{
                    // Join the chunks of the file until we get the full file
                    Log.e(TAG, "Joining file chunks of " + bytes + "bytes");
                    output.write(buffer, 0, bytes);

                    // If we received the full file
                    if(bytes < 990) {
                        byte[] out = output.toByteArray();
                        output.flush();
                        output.close();
                        Log.e(TAG, "nr of bytes file: " + out.length);
                        mHandler.obtainMessage(BtActivity.FILE_READ, out.length, -1, out)
                                .sendToTarget();
                    }
                }

            } catch (IOException e) {
                Log.e(TAG, "disconnected", e);
                connectionLost();
                break;
            }
        }
    }

    /**
     * Write to the connected OutStream.
     * @param buffer  The bytes to write
     */
    public void write(byte[] buffer) {
        Log.i(TAG, "write(byte[] buffer)");
        try {
            mmOutStream.write(buffer);
            mmOutStream.flush();
            // Share the sent message back to the UI Activity
            mHandler.obtainMessage(BtActivity.MESSAGE_WRITE, -1, -1, buffer)
                    .sendToTarget();
        } catch (IOException e) {
            Log.e(TAG, "Exception during write", e);
        }
        }

        public void writeFile(byte[] buffer) {
            Log.i(TAG, "writeFile(byte[] buffer)");
            try {
                // Send the file in chunks of 990 bytes
                Double nrSends = Math.ceil((double) buffer.length / (double) 990);
                Log.i(TAG, "will send " + Math.round(nrSends) + " chunks");
                for (int currSend = 0; currSend < Math.round(nrSends); currSend ++){
                    if ((currSend + 1) * 990 > buffer.length) {
                        Log.i(TAG, "sending final chunk from " + (currSend * 990) + " to "
                        + (currSend * 990 + (buffer.length - (currSend * 990))));
                        mmOutStream.write(buffer, currSend * 990, (buffer.length -
                        (currSend * 990)));
                        mmOutStream.flush();
                        Log.i(TAG, "sent final chunk");
                    }else {
                        Log.i(TAG, "sending chunk from " + (currSend * 990) + " to " +
                        ((currSend * 990) + 990));
                        mmOutStream.write(buffer, currSend * 990, 990);
                        mmOutStream.flush();
                        Log.i(TAG, "sent chunk");
                    }
                }

                // Debug purposes
                /*mHandler.obtainMessage(BtActivity.FILE_WRITE, buffer.length, -1, buffer)
                        .sendToTarget();*/


            } catch (IOException e) {
                Log.e(TAG, "Exception during write file", e);
            }
        }

        public void cancel() {
            try {
                mmSocket.close();
            } catch (IOException e) {
                Log.e(TAG, "Close() of connect socket failed", e);
            }
        }
    }
}
```