A fully distributed Multilayer Framework for Opportunistic Networks as an Android Application

Master Thesis

by

Andre Ippisch

born in
Düsseldorf

submitted to

Technology of Social Networks Lab
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

March 2015

Supervisor:
Jun.-Prof. Dr.-Ing. Kalman Graffi

Abstract

Nowadays we have a widespread adoption of feature-rich smartphones in society. These devices feature powerful processors, high bandwidth communication possibilities and huge storage space. However, some use cases are not supported today, for example the exchange of large files between geographically close participants. The exchange of those files over services in the Internet or through Bluetooth is limited due to bandwidth, scale and the data plans of the participants. The distribution of such files among several users is time-consuming and complicated although all functions to make it easy and cost-efficient are already available in current smartphones.

We present a Multilayer Framework for the Android operating system that uses the principle of Opportunistic Networking for local data exchange. Without user interaction the Opportunistic Network is created and files and messages are transmitted and forwarded according to a hybrid routing scheme composed of different routing principles in current literature. We use the functionality of Wi-Fi tethering hotspots on Android smartphones to create WPA-encrypted Wi-Fi connections that offer security, reliability and high transmission speed.

We developed two applications. One to manage the connection to other smartphones and to exchange data as a transport layer and a second application for processing files to be encrypted and transmitted over the created network. We used both symmetric and asymmetric cryptosystems to offer an end-to-end encryption.

We evaluated several aspects of the applications' functionality with an Android smartphone test bed. We tested initial connectivity, routing, battery life, memory consumption, range, and transmission and encryption speed.

We came to the conclusion that the network works in all tested environments but we can not make clear statements about connection times and transmission speed because there are too many interference factors. We showed that Android works not deterministically in many ways and that its network functionality has many problems and bugs which makes it difficult to work with this features.

Acknowledgements

I would like to thank all the people that supported me during this thesis, and beyond, especially my girlfriend, but also my colleagues and my birds. Without you I probably would not have come this far.

A basket full of Thank Yous goes out to my parents for the support over all the years.

Thanks to all the people that provide the world with their excellent work and develop the free and open source software that I used to write this thesis.

And last but not least, surely not for the last time, a special thanks to my supervisor Jun.-Prof. Dr.-Ing. Kalman Graffi for the opportunity, help and guidance.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

Nowadays we have a widespread adoption of feature-rich smartphones in society. These devices feature powerful processors, high bandwidth communication possibilities and huge storage space. However, some use cases are not supported nowadays, for example the exchange of large files between geographically close participants. The exchange of those files over services in the Internet or through Bluetooth is limited due to bandwidth, scale and the data plans of the participants. The distribution of such files among several users is time-consuming and complicated although all functions to make it easy and cost-efficient are already available in current smartphones.

The simple case of plain comfort is not the only motivation for a framework that allows offline exchange of files. Several circumstances can exist in which Internet connectivity is not given. These are, for example, areas without a well-developed mobile network, buildings without reception or an intentionally caused lack of Internet connectivity as happened at the Hong Kong protests in 2014.

While there are several attempts to solve the described problem, as seen in 1.2, we declare our own demands which we present in 3.1. We thereby give a motivation in the form of requirements that are nowadays not available in this form and this combination.

1.2 Related Work

The experimental Request For Comments 5050 [SB07] specifies a so called Bundle Protocol that was developed for exchanging messages in Delay Tolerant Networks. The protocol does not specify an underlying transport layer and is therefore defined in broad terms which results in design decisions

that conclude in an overhead of at least one eighth of the size of the transmitting files.

Morgenroth, Schildt and Wolf present an implementation of the bundle protocol on Android devices [MSW12]. Some aspects of the work are similar as the authors used routing protocols that form the basis of our routing scheme in Section 3.2.4. However we concentrated more on our own routing scheme as a hybrid version of different protocols and used a simple protocol that keeps the transmitted data low.

Trifunovic, Kurant, Hummel and Legendre presented an article about opportunistic networking on smartphones [TKHL15] while this thesis was written. They use similar concepts for connections but use given routing schemes from current literature instead of combining them.

Conti, Delmastro, Minutiello and Paris looked into transmitting files in opportunistic networks through Wi-Fi Direct [CDMP13]. In Section 3.2.1 we will discuss why Wi-Fi Direct is not useful for our demands.

1.3 Outline

In this chapter we gave the motivation for implementing an Android application for Opportunistic Networks and looked at attempts of other authors. The rest of the thesis is organised as follows:

In Chapter 2 we look at the basics regarding the Android operating system and the fundamentals of Opportunistic Networks. We give an introduction in Android in Section 2.1 in which we list all components for the development of an Android application. In Section 2.2 we look at Opportunistic Networks, how they are defined, what their purpose is and in which way we use them for our thesis.

In Chapter 3 we explain the demands for the Android application in Section 3.1 and discuss how to accomplish the demands with a design proposal which is given in Section 3.2.

In Chapter 4 the implementation components are given and explained. Section 4.1 gives an overview of the environment in which we implemented the applications. Section 4.2 lists the components of our framework which are explained in detail in the Sections 4.3, 4.4 and 4.5. In Section 4.6 we clarify the installation of the applications.

In Chapter 5 we present the results of several tests conducted with our opptain network application and with our FileShipping application. Different aspects are tested and evaluated. At first the fundamental ability to open and connect to hotspots is tested. In addition we test the battery life and memory usage for an ongoing usage of the opptain network application. Transmission tests inside and outside are

evaluated and we ensure that the opptain application does not interfere with other functions of smartphones. Speed tests are conducted and analysed regarding the transmission and the encryption as well the decryption of files. Finally we test the routing mechanisms of our opptain network application. That includes the creation and exchange of *MeetingSummaries*, the routing of *Packets* by means of the *MeetingSummaries* and the routing with the help of *Attributes*.

In Chapter 6 we deliver a conclusion of this thesis and the developed results with regard to the demands we expressed at the beginning. We describe how the opptain network application fulfilled our goals and how the FileShipping application provides the proof of work. In addition we present possibilities to extend our work in the future. Aspects which can be looked at in this context are, among others, a longer battery life, higher safety and a more targeted data transmission.

Chapter 2

Fundamentals

In this chapter the fundamentals for the further developments of this thesis are explained. We will look at the basics of Android and the Software Development Kit provided by Google Inc in Section 2.1 and explain Opportunistic Networking in Section 2.2 in respect to the network structure and the routing possibilities.

2.1 The Android Operating System

In this section we introduce the Android operating system, look at its structure and features and present the components of an Android application. We extend and update Krauthoff's [Kra12] summary of Android's architecture and fundamentals and the lifecycle of an application, look at the components, compatibility and permissions of Android and give information about Android's and our licensing. At last we take a look at the rooting of an Android device.

2.1.1 Basics

Android is an operating system developed by Google for mobile devices. It is open source software and is built on top of the Linux kernel [Mel15] for the purpose that many hardware devices can run Android because it can be adapted easily by the manufacturers. The Linux kernel, a 3.x kernel from the Jelly Bean release upwards, includes important features that are essential for a comfortable, fast and secure functionality of an Android smartphone. The most important ones are:

- Driver Model: Easy to use environment for developing drivers to run applications on the Android operating system.

- Security: Linux kernel manages the security of the system and of the applications which run as Linux processes.
- File system management: Linux kernel manages the data storage and file system.
- Memory and process management: Linux kernel allocates and deallocates memory and resources for applications, processes and the file system.
- Network: Linux kernel manages all network communication.

Android does not use the Vanilla Linux Kernel but a modified one since it runs on mobile devices and needs additional drivers, e.g. for 3G or SIM card. Additional Android kernel features include binders for inter-process communication, logger functionality and wake locks to keep the system awake.

2.1.2 Application Components

The following components are fundamental for the development of an Android application [Goo15c].

- Activities: An Activity is a user interface that typically comes with a single screen. An Activity serves as an entry point when the user starts the application, called a launcher Activity, and can display any kind of information for the user.
- Fragments: A Fragment is a user interface that can be integrated in any Activity. Several Fragments can be contained in one Activity and the Fragments can be exchanged at runtime and be reused by different Activities.
- Services: A Service represents a background process without a user interface for operations that exceed the users interaction with the application. With the help of an AIDL interface it is possible to exchange data between applications directly through a Service.
- Broadcast Receivers: A Broadcast Receiver can react to messages that are transmitted through the system.
- Intents: An Intent is an abstract operation that can be performed. It is used to start Activities, Services and can be received by Broadcast Receivers. To receive certain Intents an Intent Filter can be declared by an application.
- Content Providers: A Content Provider is a persistent data storage that can be shared among the system.

2.1.3 Device Compatibility

Android runs on several types of devices such as phones and tablets but also watches and televisions [Goo15e]. Android developers can provide feature requirements and Android platform version restriction to limit the devices that can install the application based on implemented features of the application and available functionality of the operating system.

2.1.4 Permissions and Features

Android applications are provided with a unique system identity and run in virtual machines called Dalvik VM. Each application runs as a Linux process in a sandbox environment to isolate it from other applications. A developer has to declare permissions [Goo15h] that allow the application to share resources and data with other applications and the system. Another declaration a developer can announce for the application is the usage of features. A declared feature describes which technology a device needs to have to be able to install the application. Those features can be hardware-based like a Wi-Fi module or software-based like SIP or VOIP services.

Our applications use the System permissions seen in 2.1. Some of the permissions imply the use of features, in our applications those are the WIFI_STATE permission that implies the presence of a Wi-Fi module, the ACCESS_FINE_LOCATION permission that implies a GPS module and the CAMERA permission that requires a camera module. Instead of just relying on implied feature usage we declared the features explicitly.

The word “Internet” might be misleading in this context. The INTERNET permission grants usage of sockets in common and does not necessary imply that a developer uses access to the Internet.

If a developer wants to allow other applications to use its Services or other resources like Content Providers the developer can declare self created permissions that other applications can use.

2.1.5 Application Lifecycle

The lifecycle of an Android application 2.1 depends in most cases on the lifecycle of the applications Activities [Goo15a] which is described in [Kra12]. However this is the case for most applications that focus on user interaction and short life processes. Such an application is started by the user and is stopped when the user closes it. The closing, i.e. shutting down the application can be executed by the user explicitly or by the system if memory resources are needed.

Permission (Implied Feature)	Necessary for	Application
ACCESS_WIFI_STATE (Wi-Fi)	accessing Wi-Fi connectivity	opptain
CHANGE_WIFI_STATE (Wi-Fi)	changing Wi-Fi connectivity	opptain
ACCESS_NETWORK_STATE	accessing networks in common	opptain
CHANGE_NETWORK_STATE	changing networks in common	opptain
INTERNET	creating sockets	opptain
ACCESS_COURSE_LOCATION (Location)	location updates over networks	opptain
ACCESS_FINE_LOCATION (Location)	location updates over all sources	opptain
WAKE_LOCK	keeping the CPU alive	opptain
WRITE_EXTERNAL_STORAGE	writing to external storage	opptain, FileShipping
READ_CONTACTS	reading contact information	FileShipping
WRITE_CONTACTS	changing contact information	FileShipping
CAMERA (Camera)	accessing camera	FileShipping

Table 2.1: Permissions used in the applications

Also there are applications that are bound to a background Service [Goo15g] to allow further use of the device. One example is a music player application that does not stop playing music while the user opens another application or turns the screen off. The background Service does not need to be long running, a Service can also be used if the user does not need to see a visible reaction to a performed task. The same way a part of one application can be executed by another application with the help of a Service. It should be mentioned that the system prevents shutting down applications that run a background Service as long as the option for a simultaneous notification is set.

2.1.6 Licensing

Android software is mostly licensed with the Apache 2.0 license, some parts are licensed differently, the Linux kernel patches, for example, are licensed with the GPLv2 license [Goo15f].

Our framework is licensed with the Apache License, Version 2.0. As required, all our code starts with a comment including the boilerplate notice seen in Listing 2.1. The License can be found at [The15].

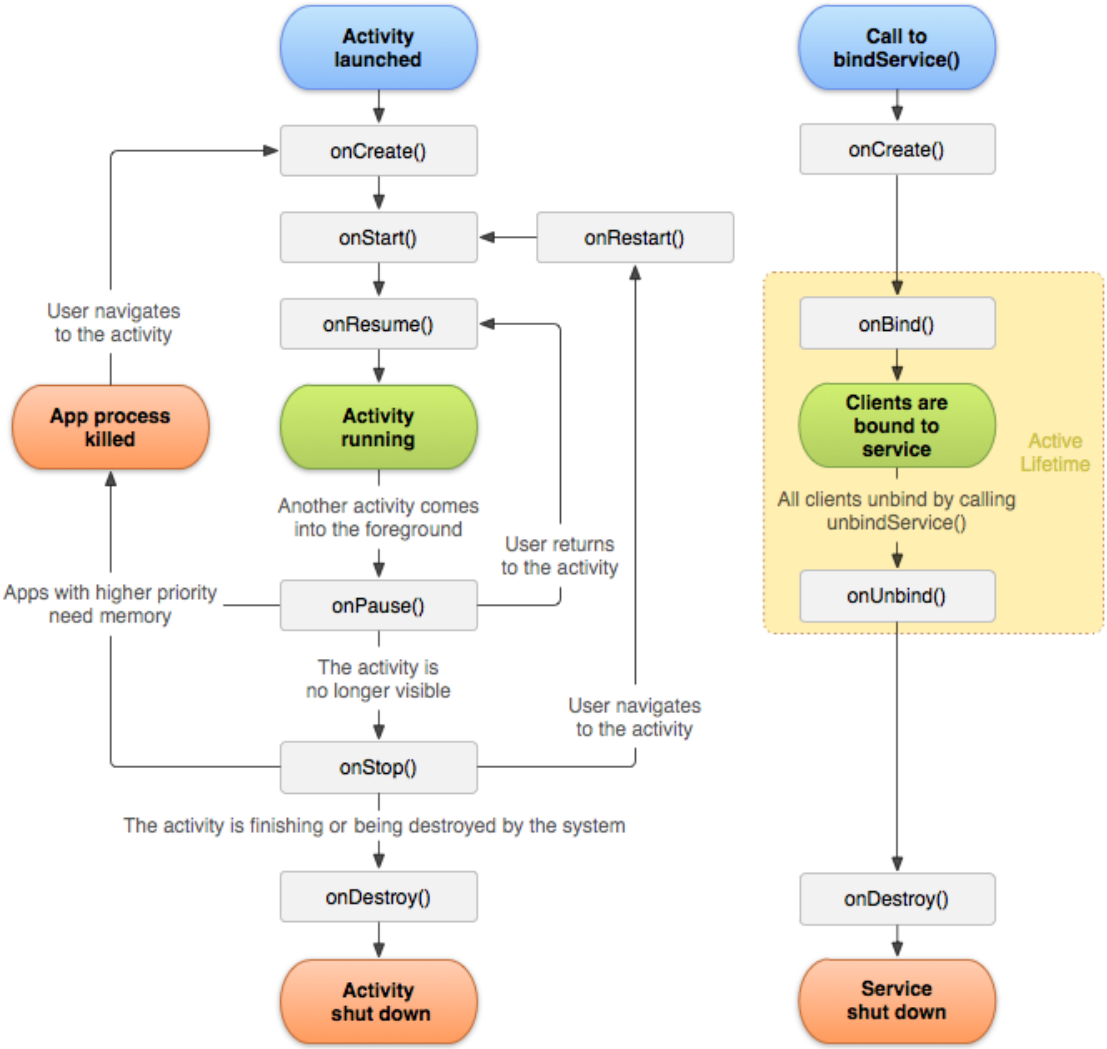


Figure 2.1: The lifecycle of an Android application.
Based on the figures of [Goo15a] and [Goo15g] by Google Inc. and adapted with permission of the given CC BY 2.5 license [Cre15]

Listing 2.1: Apache License, Version 2.0, boilerplate notice

```
Copyright 2015 Andre Ippisch

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

Version	Codename	API	Distribution
2.2.x	Froyo	8	0.4%
2.3.x	Gingerbread	9-10	6.9%
3.x	Honeycomb	11-13	$< 0.1\%$
4.0.x	Ice Cream Sandwich	14-15	5.9%
4.1.x - 4.3.x	Jelly Bean	16-18	42.6%
4.4.x	KitKat	19	40.9%
4.4W	Wear	20	$< 0.1\%$
5.0.x	Lollipop	21	3.3%

Table 2.2: Android version distribution in March 2015

2.1.7 SDK Software Development Kit

The Android Software Development Kit (SDK) is an archive that contains basic tools for developing and testing applications for Android. The SDK is released without an Android platform and third-party libraries. An Android platform is a system image for a particular processor architecture that is used to compile applications against specific Android versions. At least one Android platform has to be installed to develop applications.

Platform Versions

Google Inc. provides updates for Android frequently and has released many platform versions for the operating system. Since the manufacturers use their own build of Android platforms they have to provide updates themselves which does not necessarily happen fast or at all. All devices that were used for developing were released by Google Inc. with a non-modified Android version. At the beginning of this thesis KitKat was the latest version and was used as a platform version for developing and compiling. All platform versions are backwards compatible except for new features that were introduced in later versions and are forward compatible in the form that all applications compiled with an older platform version will run on phones with a newer version. Exceptions are illustrated in 2.1.7.

Application Programming Interface

The Android platforms that are distributed by the SDK contain only a subset of functionality that is actually available on Android devices and used by the system itself. Many parts of Android that concern network connectivity have a larger set of functions available and those functions are not released through the official Application Programming Interface (API).

Since our framework needs to change major connectivity states, like opening a tethering hotspot or connecting to a network that was previously unknown, we decided to build our own platform version based on the KitKat release from the official framework in the Nexus smartphones 5.1. All functionality we use was available from Jelly Bean on and the latest platform Lollipop still provides all used functions. There is no guarantee that those functions will be available in later versions of Android.

Our framework should run on all devices from Jelly Bean on that support Hotspot Tethering and whose manufacturers did not disable these functions on purpose. In March 2015 nearly 87% of all Android devices run at least with Jelly Bean as seen as in Table 2.2 which is taken from the Android dashboards [Goo15d].

2.1.8 Rooting

Rooting is a process that enables Android to run with raised permissions by installing a Linux superuser application. It allows the user to read and alter any file on Android's file system. Legality and conservation of warranty of rooting an Android device are different in any country. Rooting is in principle possible for all devices that are supported by drivers that are normally distributed by a rooting community. Because rooted devices can read all files and therefore all private application data it is not possible to secure data inside the application.

2.2 Opportunistic Networks

Opportunistic Networks (OppNets) are mobile ad hoc networks (MANETs) in which "unpredictable and unstable topologies, prolonged disconnections, and partitions can occur frequently" [WDAV13]. They are a subset of Delay (or Disruption) Tolerant Networks (DTNs) [Fal03] in which "communication opportunities (contacts) are intermittent, so an end-to-end path between the source and the destination may never exist" [HLT08].

In the following sections different aspects of OppNets will be illustrated. One aspect is the addressing

and identification of nodes which will be looked at in Section 2.2.1. Furthermore an important point is the routing in OppNets which will be described in Section 2.2.2. We will show the differences to the routing in MANETs and, based on this, name and explain different routing protocols which can be used in the given context.

2.2.1 Addressing Nodes in Opportunistic Networks

In this section we want to discuss how we can address peers in Opportunistic Networks.

We assume that for the assignment of a unique identification there is no server we can access, neither local or through the Internet. On Android devices there are a few possibilities for unique identification that are provided by the device itself. Each device has an International Mobile Equipment Identity (IMEI) and MAC addresses for several network adapters. Since the IMEI can be used to block a mobile device for network communications by mobile providers after, for example, theft of the device, it is a sensitive number that should not be used for the purpose of identification in a self-initialised network. A MAC address is used for addressing a device on the data link layer. MAC addresses are unique when delivered with the belonging network adapter because MAC address blocks are distributed by the IEEE Registration Authority and manufacturers are allowed to give out unique addresses.

We take the MAC address of the device's Wi-Fi module as the main identification of the device. The MAC address is unique because there is no possibility to change the MAC address on unrooted Android devices either on hardware or software side. The MAC address in which all 48 bits are set to "1" (ff:ff:ff:ff:ff:ff) can be used to broadcast messages in the network with restriction to the used routing scheme explained in the next section.

2.2.2 Routing in Opportunistic Networks

In Opportunistic Networks (OppNets) reliable routing is not possible because the contained nodes are possibly never connected to each other at the same time. While with Mobile Ad-hoc Networks (MANETs) a disconnect is considered rare, it is highly usual in OppNets. Also in MANETs data is sent from one node to the next with the intention that the node is the intended receiver of the message or will forward the message immediately. There are several routing protocols for MANETs that can be used to perform reliable transmission of data from one point of the network to another one. In contrast OppNets do not necessarily have to be completely connected networks. Routing protocols that work in MANETs do not work in OppNets since the route might be non-existing at the time the sender has found a route.

In current literature there can be found many various protocols which can be used for OppNets. These can be divided into different categories. The most notable categories are the context free, the mobility based, and the social context based. While the context free category deals with the distribution of messages in general, in the mobility based category a context which allows a more targeted distribution of messages is added. The aspect of aimed routing is improved in the social context based routing.

With regard to the question how different routing protocols, of which the before mentioned categories consist, could be used in the given context, the most interesting ones are mentioned, explained and categorised in the following. The further approach for the usage of these protocols is explained in Section 3.2.4.

Context Free Routing

Context free routing is a principle in which peers have no information about other peers and their context to deliver a message.

In the area of context free routing one scheme considered the most common is the flooding of the network. Two derived principles are blind and controlled flooding. With blind flooding a peer sends out a routing message to all connected peers and those do the same until everyone in the network including the receiver got the message.

A well known routing principle that derives from flooding is Dynamic Source Routing which is used in MANETs. All peers that receive a copy of the message add their identification to the message before forwarding it which results in the message's addressee knowing the path that the message took on its way through the network. The receiver can now send an answer to the sender on the path found with the query message. The sender now can use the path to send the data over the path to the destination. This principle does not work in OppNets since the route that is found when the routing message has reached the destination may be not existing anymore at the time the answer message is arriving.

Instead of sending a route message the sender can just send the data packet instead. It would reach the destination as well as all other peers in the network.

To limit the flooding of the message a hop counter or a Time-To-Live can be declared to stop the network of overflooding. Vahdat and Becker proposed Epidemic Routing [VB$^+$00] for mobile devices that uses Time-To-Live for expiration of messages. When two devices connect to each other they compare messages they have in their waiting list and exchange the ones they do not have in common. The message gets delivered when one peer with a copy of the message connects with the destination.

Another controlled flooding algorithm is Spray and Wait [SPR05] by Spyropoulos, Psounis and Raghavendra. Only the sender can create copies of the original message and distributes them to peers that are connected, this is called the Spray phase. The receivers of the copied messages, however, are not allowed to copy the message but only forward the single copy they got to the destination, called the Wait phase, so that only a constant number of copies exists in the network. Time-To-Live is used in this protocol, too, for the expiration of the message. Spray and Focus [SPR08] is an extension of the Spray and Wait protocol in which the peers that received the copy of the message are allowed to forward, but not copy, the message to a peer after not having been able to deliver the message to the receiver, the Wait phase is replaced by the Focus phase. The owner of one copy of the message will only forward the message to a peer that has a high utility to deliver the message to the destination. With Seek and Focus [SPR08] there is another aspect of the Spray and Focus routing that allows the owner of the message to forward the message randomly to other peers until one is found that has a high utility to deliver the message.

Mobility Based Routing

Mobility based routing is a principle in which the mobility and connectivity patterns are shared among peers to enable a targeted delivery of messages.

In PRoPHET (Probabilistic Routing Protocol using History of Encounters and Transitivity) [LDS04] by Lindgren, Doria and Schelén, a peer is sending the messages according to a delivery probability that is calculated by using the *History of Encounters and Transitivity*, a list of meetings with other peers that everyone collects and shares with others. A peer forwards a message only to another peer if that one has a higher delivery probability value.

A similar example of mobility based routing is the Meeting and Visits, short MV, protocol [BBL05] by Burns, Brock and Levine. Not only the meetings with other peers are collected and shared among connected devices but also visits to geographical locations. Like in PRoPHET this information is used for routing messages to peers that have a higher delivery probability.

Social Context Based Routing

Social context based protocols use not only the mobility information of the peer to route through the network but also the social aspects of the peers. Routing protocols can be used with additional context information to predict the delivery probability not only by mobility information but also by other values that are given by the context of the peer. One used principle is that social activities and

Name	Author	context	Description
Epidemic Routing [VB$^+$00]	Vahdat et al.	context-free	blind flooding
Spray and Wait [SPR05]	Spyropoulos et al.	context-free	controlled flooding
Spray and Focus [SPR08]	Spyropoulos et al.	context-free	controlled targeting
Seek and Focus [SPR08]	Spyropoulos et al.	context-free	controlled targeting
PRoPHET [LDS04]	Lindgren et al.	context-aware	probabilistic targeting
HiBOp [BCJP07]	Boldrini et al.	context-aware	probabilistic targeting
MV [BBL05]	Burns et al.	context-aware	probabilistic targeting

Table 2.3: List of considered routing schemes that can be used for Opportunistic Networks on Android devices

also geographical structures motivate the mobility of people, which are considered a target area in the field of social context based routing.

HiBOp (History Based Routing Protocol) [BCJP07] by Boldrini, Conti, Iacopini and Passarella uses a node profile to describe the peer. The node profile, called Identity Table, consists of personal information like name and email, residence, workplace, profession and hobbies. The peers share their Identity Table among other peers and use it not only to describe a current context but also for calculating habits of the peer. Current context and predicted habits are used for calculating delivery probabilities.

Chapter Conclusion

In this chapter the fundamentals for this thesis have been explained. In Section 2.1 we looked at the Android operating system which is an open source software developed by Google for mobile devices.

In this context the basics in 2.1.1 as well as the Application Components in 2.1.2 were looked at. In Section 2.1.3 the broad spectrum of Device Compatibility was discussed which is one important factor for the overall decision to use Android as the operating system for the applications developed in this thesis. This will be explained in greater detail in Chapter 3.1.

In Section 2.1.4 we looked at the topic of Permissions and Features. In this context it was explained that a developer needs to permit other applications and the system to share resources and data. In addition it was explained how a developer can define which technology a device is obliged to have to be able to install the application and it was described which conditions were imposed for the usage of our application.

Furthermore the Application Lifecycle in 2.1.5 which deals with the possibilities to shut down an

application or keep it running as well as the Licensing in 2.1.6 were discussed.

In Section 2.1.7 it was explained that the Software Development Kit is an archive that contains basic tools for developing and testing applications for Android. As it is released without an Android platform we discussed the topic of platform versions. In this thesis KitKat is used as the platform version as it was the latest version at the beginning of this thesis. We explained further on, that the official Application Programming Interface does not include all functionality that is actually available on Android devices. Therefore the decision was made to build our own platform version based on the KitKat release from the official framework in the Nexus smartphones.

All functionality we use was available from Jelly Bean on. As explained nearly 87% of all Android devices run at least Jelly Bean so that a great coverage is reached. In Section 2.1.8 the principle of Rooting was shortly explained.

In Section 2.2 we looked at the topic of Opportunistic Networks. After a brief definition of the term we look at addressing and routing in OppNets.

In Section 2.2.1 we dealt with the question how the devices in an OppNet can be addressed unmistakably. For this purpose the MAC address was chosen as it is unique and in contrast to the also unique IMEI not to be considered sensitive information.

Section 2.2.2 focused on routing in OppNets. The main difference to routing in MANETs was pointed out which is the fact that in an OppNet a disconnect is highly usual and a route is likely to be nonexisting at the time the sender has found that route. Considering this it was made clear that routing protocols for MANETs do not work for OppNets and that routing protocols which could be used for OppNets should be looked at in detail. The most important ones in view of the developments in this thesis were mentioned and described.

In this context known protocols were divided into different categories which are the context free category, the mobility based category and the social context based category. Whereas in the context free routing the devices do not have any information about other peers in context free and mobility based routing such information is collected and exchanged and can be used to reach a more targeted distribution of data.

In the next chapter we will declare demands and use the fundamentals of this chapter to design the concept of our framework.

Chapter 3

Demands and Design

In this chapter we first want to point out the objective of this thesis in Section 3.1. In this context the various fundamental requirements are mentioned and explained in detail. Thereby the basis is built for the decisions concerning the design which are then explained in Section 3.2. In this section different aspects are looked at in particular. These are Networking, Partition of Tasks and Responsibilities, Runtime and Routing in our Network.

3.1 Demands

The idea behind this master thesis is to create a use case for Opportunistic Networks (OppNets) that can be established today with available resources and that does not entirely depend on future development. The main requirement for a working OppNet is a certain, not widespread, distribution of peers, whereby a widespread distribution helps the network to work faster and more stable. To achieve this distribution, as a node we chose smartphones as they are widespread and bound to a person, staying and moving always with them and therefore are one big OppNet already. Android is a perfect environment for the development of such a system as it is the by far most widespread operating system for smartphones. In the year 2014 Android reached a market share of 81.5% [IDC15] and shipped, with over one billion units, more smartphones than the next five great sellers combined.

The task was to build an Android application that uses OppNet mechanisms to transfer any kind of data between Android devices wirelessly. The application should work without an Internet connection and transfer the data fast and securely.

To ensure the before mentioned widespread distribution of the application there should be a high cost-benefit-ratio for the user. There should be a wide possible spectrum of use cases for the network. The users should be able to use the network for their own tasks like sending messages or transferring any

kind of files.

Furthermore, the user of the application should be minimally bothered in her routine using the device. Both by running the application in the background and by giving her the possibility to temporarily disable the application's network connection to get access to other Wi-Fi networks. Also there should be no necessity for user interaction when it comes to connecting devices and exchanging data.

For delivering data to devices, especially those that are not directly connected, a routing scheme must be implemented that suits the given prerequisites and makes full use of all aspects of modern smartphones. For that we have to evaluate routing protocols in current literature and choose either the most promising one or combine advantages of several routing protocol to develop a new routing scheme.

To accomplish a marketable solution not only we have to design the network application that represents the OppNet but also a use case to provide a proof of work. We realised this in the form of a filesharing application that can be used with the network. A filesharing application covers several of the mentioned aspects. We can present a functional routing algorithm, the exchange of files with fast and secure transfer and that user interaction is not necessary to fulfil most of the tasks.

To provide access to the network for other applications, both self implemented and third party owned, an Application Programming Interface (API) is necessary to connect the applications.

3.2 Design

In this section we give design proposals for the demands declared in the previous section. Different aspects are looked at in particular. These are Networking, Partition of Tasks and Responsibilities, Runtime and Routing in our Network.

3.2.1 Networking

The main task of the application is to connect with other devices that run the same application. To maintain this connection we have to choose a wireless technology. There are many possibilities to transfer data from one Android device to another but if we take a look at the implemented ad hoc standards that Android offers we see that they do not fulfil our demands 3.1.

- Wi-Fi Direct is a standard that uses IEEE 802.11 (Wi-Fi) technology which was developed to connect devices directly without a need for an access point. Androids Wi-Fi Peer-to-Peer

ISO/OSI	Our model	Device A	Device B	Device C
Application	Application	API app		API app
Application	Transport	network app	network app	network app
Transport	Transport	TCP	TCP	TCP
Transport	Transport	IP	IP	IP
Transport	Transport	802.11 MAC	802.11 MAC	802.11 MAC
Transport	Transport	802.11 PHY	802.11 PHY	802.11 PHY

Table 3.1: Protocol stack including the network application as part of our model's transport layer and an arbitrary API application for network usage

which is derived from Wi-Fi Direct has no API to accept connection requests programmatically, therefore the user would have to accept each link manually and it would not be possible to automate the process.

- Bluetooth is a standard to connect devices over a short range. The Bluetooth implementation can only provide a connection if the devices have been previously paired by the device owners and the user has to accept the pairing manually.

- Near Field Communication is a standard for small data transmission that requires an even closer distance between the devices. Additionally, only a low bandwidth can be reached.

Another approach that is not in the field of Peer-To-Peer communication is the possibility to enable a Wi-Fi tethering hotspot on Android devices. It serves as an access point to other devices which can connect to the access point. The functions to programmatically enable a hotspot are not available in the official API but are implemented in Android's source code from Jelly Bean on 2.1.7. Advantages are long range transmission and the use of the TCP protocol for reliability.

We will use Wi-Fi as the connection technology by creating a tethering hotspot on one device and connecting to it from another device. For security reason the connection will be WPA-encrypted. One major problem is that the passphrase has to be pre-shared and saved on the device. This approach however makes the passphrase visible to all users of rooted devices 2.1.8 and the network vulnerable. Therefore we can use the encryption to prevent spontaneous attacks only. In addition non-participants that automatically connect to non-encrypted Wi-Fi networks are prevented from connecting to our network. We will use the term "tethering hotspot" for the access point that an Android device can open while the general term "access point" not necessarily refers to an Android hotspot.

3.2.2 Partition of Tasks and Responsibilities

One of the goals from Section 3.1 was to use the connection for a variety of tasks like transferring files or using it for a chat program. Therefore the application should concentrate on transferring plain data between devices and let other applications built on the network application to perform higher tasks. We designed the network application to act as the network layer and offer an API to let other applications serve as the layer that builds upon the network layer. The network application only manages the connection and transmits the data. Like an IP packet or a TCP segment we have to offer a similar datagram structure for our network application. Additionally the communication between the applications for the handover of this datagram is offered by an interface that third party applications can implement against.

Table 3.1 shows the protocol stack in which network and API application are divided into two separate layers with regard to our model. As seen for Device B the API application is unnecessary for forwarding the data to the next device.

The principle of the partition of tasks and responsibilities is one of Android's policies, too. That means that there should not be any functionality in an application that can be achieved by another application that already implements this feature. For example a browser application that downloads a PDF document is not meant to open that file but should be able to delegate the task of opening the file to an application that displays documents.

Following that principle we will not, for example, implement a file explorer functionality inside of our filesharing application but instead rely on file explorer applications that give the user the possibility to simply choose a file so that our filesharing application only has to manage the additional information necessary to transfer this file.

3.2.3 Runtime

For the requirement that the network application should run in the background Android offers the *Service* class that represents a long running process that runs without a graphical user interface (GUI) in the background of an application. A *Service* can be started by an *Activity* with the intention to run a task and stop itself after finishing. On the other hand a *Service* can run a long running task that does not stop by itself. One or more Activities that want to use and communicate with the *Service* can bind it which starts the *Service* if it was not running before. The *Service* stops when the last *Activity* unbinds. We can use this *Service* class to run the Wi-Fi functionality that connects with other devices and therefore sets up the Opportunistic Network. The *Activity* that binds the *Service* offers the GUI to start and stop the *Service*. We show a Notification in the Notification Bar to make the user aware of

the running *Service* and to have a quick entry point back to the *Activity* to stop the *Service* if wished for.

3.2.4 Routing in our Network

Smartphones are predestined for working as nodes in OppNets since they are powerful devices that come with all necessary hardware and software installed to use current literature routing protocols. They feature powerful processors, high bandwidth communication possibilities and huge storage space, they are equipped for tracking time and location, like GPS and cell tower localisation. Since most smartphone users tend to be in social communities, usually software is installed that can be used for gathering information for social context.

The before mentioned possibilities were the basis for the selection of routing protocols in chapter 2.2.2. The routing protocols which benefit from the possibilities were chosen. They are used in an hybrid form to enable the routing in our mobile network.

The protocols of Section 2.2.2 have been chosen because they can all be used and combined to a protocol for our application. For the purpose of spreading messages in general we use a hybrid of spray and wait, spray and focus and seek and focus. For the purpose of calculating delivery probabilities we use the frequency of *Meetings*, locations of those *Meetings* and additional social context *Attributes*. All with a Time-To-Live that is required to remove messages when they are expired.

The basis for the usage of this routing protocol is to keep a record of the Meetings. This does not represent a problem for the devices as they are able to save forwarded messages and data packets and to store information about other devices. When two devices connect they exchange an information set. The information set contains all information that is necessary for the routing algorithms.

The *History of Encounters and Transitivity* of PRoPHET and the *Meetings* principle of HiBOp are combined to our form of *Meetings* and *Meeting Summaries*.

Meetings A *Meeting* consists of the MAC address of the two connected devices and the time when, and optionally the place where, it happened.

Meeting Summaries When at least two *Meetings* between two devices exist, a *Meeting* Summary is created that does not contain all *Meetings* but a summary of those *Meetings* in the form of the amount and frequency of the *Meetings* as well as an optional area where those *Meetings* happened.

The Dijsktra Algorithm is used to calculate if there is a potential path between the connected device and each *Packet* receiver. The potential path is based on the *Meeting* Summaries and the frequency of *Meetings*. The *Meeting* Summary can be seen as an edge in the Dijkstra graph with the two connected devices as nodes and the frequency as edge weight.

In addition to the information when and how often *Meetings* take place information about the place of the *Meeting* and the social context of the devices respectively their owners can be saved and used to allow targeted distribution of *Packets* and to calculate probabilities. This aspect of the routing process was realised in a simplified manner which is presented in Chapter 4. Many further possibilities to use and evaluate these information are imaginable. This is illustrated in the Future Work in Chapter 6.2.

Additional Information

Additional information that is used for routing, according to the MV, PRoPHET and HiBOp routing schemes are

- GPS and/or area information where the broadcast or receiver should go to, reduced to one GPS position.
- HiBOp's Identity Table, reduced to one field, called *Attribute* that the user can set herself.

For a real life scenario the calculation of the weight is calculated by the amount and frequency of *Meetings*, in our small world example the weights are set to the same value which means that the existence of a path in the Dijkstra algorithm is sufficient enough for choosing the connected device as a next hop for the *Packet*.

For every combination of addressing methods (Receiver or Broadcast) and additional information (GPS and/or *Attribute*) there is a suitable routing scheme combination. Table 3.2 shows the desired action for each combination.

Saving Meetings, Visits and Attributes

When two devices connect, each device saves the details of that *Meeting* in its list of *Meetings*. The details of the *Meeting* are the time, and the place as GPS coordinates. Each device then calculates a *MeetingSummary* which describes the frequency of the *Meetings* amongst other things and saves it in a list of *Meeting* Summaries. The *MeetingSummary* is only created if at least two *Meetings* between theses devices occurred thus preventing that random *Meetings* have relevance for the routing. The list

Addressing	GPS	Attribute	Routing Scheme(s)
Receiver	no	no	MV, PRoPHET
Receiver	yes	no	MV, PRoPHET
Receiver	no	yes	MV, PRoPHET and HiBOp
Receiver	yes	yes	MV, PRoPHET and HiBOp
Broadcast	no	no	Epidemic Routing with Hop-Counter
Broadcast	yes	no	Spray and Focus
Broadcast	no	yes	Spray and Focus
Broadcast	yes	yes	Spray and Focus

Table 3.2: Routing schemes considered for routing based on the additional information

of *MeetingSummaries* is shared with the connected device and can be used to calculate the relevance of the connected device for routing the *Packets*. The same way a list of *Attributes* is collected and shared between the devices.

Only *MeetingSummaries* that have been created by devices connected directly are stored on the device, by that only a two hop neighbourhood is saved on the device, but a three hop neighbourhood can be used for calculating the delivery probability.

Listing 3.1: Connection between devices

```
1  onConnect(listOfMeetingSummariesFromOtherDevice, attributeFromOtherDevice)
2      calculate listOfOwnMeetingSummaries with listOfMeetings
3      send listOfOwnMeetingSummaries and own attribute to other device, invoke onHandshake method on that device
4      invoke onHandshake(listOfMeetingSummariesFromOtherDevice, attributeFromOtherDevice) on this device
5  end function
6
7  onHandshake(listOfMeetingSummariesFromOtherDevice, attributeFromOtherDevice)
8      save meeting in listOfMeetings
9      merge listOfAllMeetingSummaries with listOfMeetingSummariesFromOtherDevice
10     add attributeFromOtherDevice to own listOfAttributes
11     remove expired entries from listOfAllMeetingSummaries
12     for each packet in listOfPackets
13         calculate with listOfAllMeetingSummaries the delivery probability for connected device
14         if probability is high enough
15             send packet to connected device
16         end if
17     end for
18 end function
```

Based on Figure 3.1 and Table 3.3 the process of exchanging *MeetingSummaries* is explained step-wise. The setting consists of five devices in a line with three devices staying at one place and two

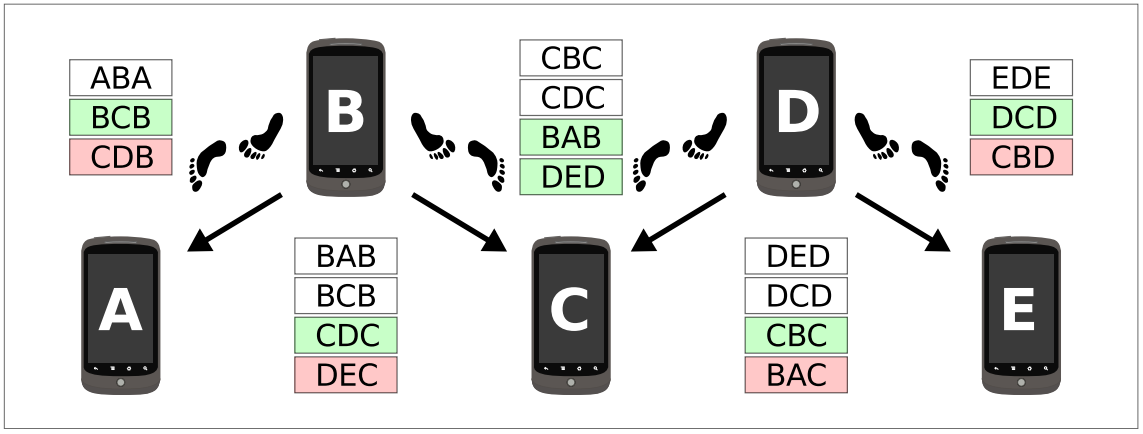


Figure 3.1: Scenario for MeetingSummary exchange

Round					
1	AB		C	D-	E
2	A	B-	C		DE
3	A		BC	-D	E
4	A	-B	CD		E
5	AB		C	D-	E
6	A	B-	C		DE
7	A		BC	-D	E
8	A	-B	CD		E
9	AB		C	D-	E
10	A	B-	C		DE

Table 3.3: Prerouting Connections for each round. XY represents a connection between device X and Y, -X stands for X moving towards the left side, X- stands for X moving towards the right side, X alone stands for a device not being connected in this round

devices moving between them. To accomplish a state, in which each device has all *MeetingSummaries* it can get, ten rounds are necessary for our example. Table 3.4 shows all creations and exchanges of *MeetingSummaries* in and between the rounds and Tables 3.5 and 3.6 show all *Meetings* and *MeetingSummaries* after each round.

Meetings are given in the form XY where X is the device that created the *Meeting* and Y is the connected device. *MeetingSummaries* are given in the form XYZ where X is the device that created the *MeetingSummary*, Y is the device to which there was a frequent connection and Z is the device the *MeetingSummary* was obtained from. If the *MeetingSummary* was not obtained from another device Z is equal to X.

Prior to the first round no device has neither a *Meeting* nor a *MeetingSummary* yet. In the first round devices A and B connect to each other thus creating a *Meeting* itself and not exchanging any *Meet-*

Round	Action
1	-
2	-
3	-
4	-
5	-
-	A creates ABA
-	B creates BAB
6	-
-	D creates DED
-	E creates EDE
7	C gets BAB
-	B creates BCB
-	C creates CBC
8	C gets DED
8	D gets CBC
8	D gets BAC
-	C creates CDC
-	D creates DCD
-	D deletes BAC
9	A gets BCB
-	-
10	E gets DCD
10	E gets CBD
-	E deletes CBD

Table 3.4: Prerouting Actions for and after each round. A row labelled with a number represents the actions during a round, a hyphen stands for an action after a round.

Round	A (M)	A (MS)	B (M)	B (MS)	C (M)	C (MS)
1	AB	-	BA	-	-	-
2	AB	-	BA	-	-	-
3	AB	-	BA,BC	-	CB	-
4	AB	-	BA,BC	-	CB,CD	-
5	AB,AB	ABA	BA,BC,BA	BAB	CB,CD	-
6	AB,AB	ABA	BA,BC,BA	BAB	CB,CD	-
7	AB,AB	ABA	BA,BC,BA,BC	BAB,BCB	CB,CD,CB	CBC,BAB
8	AB,AB	ABA	BA,BC,BA,BC	BAB,BCB	CB,CD,CB,CD	CBC,BAB,CDC,DED
9	AB,AB,AB	ABA,BCB	BA,BC,BA,BC,BA	BAB,BCB	CB,CD,CB,CD	CBC,BAB,CDC,DED
10	AB,AB,AB	ABA,BCB	BA,BC,BA,BC,BA	BAB,BCB	CB,CD,CB,CD	CBC,BAB,CDC,DED

Table 3.5: Prerouting Meetings (M) and MeetingSummaries (MS) after each round for devices A-C

ingSummaries because they do not have created some before. In the second, third and fourth round devices D and E, B and C, and C and D respectively, connect to each other likewise. In round five

Round	D (M)	D (MS)	E (M)	E (MS)
1	-	-	-	-
2	DE	-	ED	-
3	DE	-	ED	-
4	DE,DC	-	ED	-
5	DE,DC	-	ED	-
6	DE,DC,DE	DED	ED,ED	EDE
7	DE,DC,DE	DED	ED,ED	EDE
8	DE,DC,DE,DC	DED,DCD,CBC	ED,ED	EDE
9	DE,DC,DE,DC	DED,DCD,CBC	ED,ED	EDE
10	DE,DC,DE,DC,DE	DED,DCD,CBC	ED,ED,ED	EDE,DCD

Table 3.6: Prerouting Meetings (M) and MeetingSummaries (MS) after each round for devices D-E

devices A and B meet for the second time. While there are still no *MeetingSummaries* to exchange, after having connected both devices create *MeetingSummaries* ABA and BAB. Devices D and E do the same in round six. In round seven devices B and C meet and C gets the *MeetingSummary* BAB that documents the frequent *Meetings* between B and A. After the connection devices B and C as well have connected twice and create the *MeetingSummaries* BCB and CBC. In round eight devices C and D connect and exchange *MeetingSummaries*. C gets *MeetingSummary* DED from device D and D gets *MeetingSummaries* CBC and BAB from device C. Of these two *MeetingSummaries* CBC is kept because it involves the previously connected device C but BAB gets deleted because it exceeds the two hop neighbourhood. In the following rounds the creation of *Meetings* and *MeetingSummaries* as well as the exchange of *MeetingSummaries* happens according to the same principle. In the tenth round there are no more *MeetingSummaries* that are created anew or exchanged for the first time but only updated versions of the previous ones.

Chapter Conclusion

In this chapter the demands and idea behind this thesis as well as the design decisions have been explained.

In Section 3.1 we focused on the fundamental demands for the developed applications. The main task was to built a application as a use case for OppNets with today available resources. The more widespread distributed a OppNet is the better and faster it works. To achieve this the following demands were expressed. The application should be available for many users and a high cost-benefit-ratio for the user should be reached. Therefore the application is to be developed for Android devices and it should be possible to use it to securely and quickly send any kind of data without limiting other functions of the devices. Another important task to realise a working OppNet was to develop a

suitable routing scheme that makes full use of the possibilities modern smartphones offer. To deliver a proof of work a filesharing application is to be developed. To allow this filesharing application and other applications to access the network an API is necessary.

In Section 3.2 important decisions concerning the design of the application were explained. In Section 3.2.1 we discussed different available wireless technologies regarding the networking of our application and justified the decision to use Wi-Fi as the connection technology to build our OppNet upon as Android devices offer the possibility to enable Wi-Fi tethering hotspots as access points to which other devices then can connect. Also the matter of security was illustrated.

The basic concept of the partition of tasks and responsibilities was explained in Section 3.2.2. It means that an application should not offer any functionality another applications already provides. This is a fundamental policy of Android. Therefore it was described how this concept was adopted in our application.

The solution to ensure that the network application runs as long as needed is described in Section 3.2.3. A *Service* class offered by Android is used that represents a long running *Service* which can be bind by *Activities* which want to use it or communicate with it.

The important topic of routing in our network was dealt with in Section 3.2.4. It is illustrated why smartphones are predestined for working as nodes in OppNets and how the protocols of Section 2.2.2 are combined to our routing scheme. In addition we showed how the principles of PRoPHET and HiBOp are used to develop our form of *Meetings* and *MeetingSummaries* which represent the basic for our routing scheme. The routing scheme is explained stepwise and in detail with the help of an example.

In the next chapter we will implement these design proposals in different applications.

Chapter 4

Implementation

In Chapter 3 we discussed the demands and the design of our framework. In this chapter we want to present the implementation of the framework. Section 4.1 gives an overview of the environment in which we implemented the applications. Section 4.2 lists the components of our framework which are explained in detail in the Sections 4.3, 4.4 and 4.5. In Section 4.6 we clarify the installation of the applications.

4.1 Development

For the development of the framework we used Eclipse, an open source Integrated Development Environment (IDE) combined with the Android Development Tools (ADT) plugin provided by Google Inc. At the start of the thesis there was the option to use Android Studio, the official IDE developed by Google Inc., but it was only available in a beta version and the SDK was integrated which made it problematic to compile against our own Android platform version. During the development of the framework a stable version of Android Studio was released but we stayed with the familiar Eclipse IDE. Eclipse and the ADT were always used in the most current version, Eclipse at last in the Luna version 4.4.1 and the ADT at last in version 23.0.4. The SDK was used in its latest version as well, the last version of the Android SDK tools was 24.0.2, the latest SDK Platform-tools version was 21 and the latest SDK Build-tools version was 21.1.2. We compiled, as mentioned in 2.1.7, against the self-built KitKat platform version.

We tested the framework with test devices that are described in 5.1 and used the JUnit framework for testing java classes that had no Graphical User Interface (GUI). We could not test the application with the device emulator that comes with the development tools because there is no possibility to emulate a tethering hotspot in it.

Android is using the Dalvik Virtual Machine which is based on the Java technology developed by Sun Microsystems and Java is the programming language used for developing the applications. Android includes only a subset of all Java classes and leaves processor-intensive ones out.

Android comes with a built-in logging system called *LogCat* that allows developers of applications to save information about the application. *LogCat* allows the developer to filter logs by many criteria and divides logs by application and Java classes. However *LogCat* does not come with the functions to save logs permanently on the device which is required for testing the application properly over a longer time. We decided to use *logback* and *slf4j* as alternative logging library whereby *logback* is based on the log4j logging library and adapted to be used on Android and *slf4j* is a logging facade that can be used with *logback*, Android's *LogCat* and other logging frameworks. All logs are both saved on the device and piped to *LogCat* by which no functionality is lost. *logback* is dual-licensed under the EPL v1.0 [Ecl15] and the LGPL 2.1 [Fre15], sjf4j under the MIT [Ope15] license.

To integrate communication within the application we use the Otto event bus that enables event driven correspondence between classes without referencing each other.

For providing QR-Code functionality we use the ZXing library that allows to be integrated in every application and performs all actions necessary to scan or show QR- and barcodes.

We use the Apache Commons Lang Library for Serializable functions that help us to send data from one device to another.

Otto, ZXing and the Apache Commons library are open source and licensed under the Apache License, Version 2.0 [The15].

4.2 Structure

We provide the framework in the form of a Toolkit that contains all necessities to provide the network, manage the connections between devices and make usage of the network by providing additional information and functions that every third party application needs for basic functionality or for security purposes.

The Toolkit consists of the following three components:

- Library: The library contains all API information and optional classes for encryption and decryption.

(a) Icon of the opptain network application

(b) Icon of the FileShipping application

Figure 4.1: Icons for the opptain network and FileShipping application

- opptain application: The opptain network application for providing and managing the Opportunistic Network.

- FileShipping application: The toolkit provides a sample filesharing application as a use case for the opptain network application.

Figure 4.1 shows the icons of the two applications. The library has no own icon since it is not installed as an application on the device. In the following Sections we will take a closer look at the library and the applications.

4.3 Toolkit Library

The Toolkit Library is a collection of necessary API information for interaction with the opptain network application and optional helping classes for encryption, decryption, hashing and information gathering. The API defines the *Packet* class which is used as a header for transmission data exchanged on the network layer and for interaction between the applications. The library also contains several classes that have been shared across the opptain network and the FileShipping application, and were therefore put into the library and can also be used by third party applications.

One important element of our network applications is the security aspect. We will look at security in common and at the classes that offer encryption and decryption to secure all data from users that are not supposed to read it and from potential attackers. While we decided to do transmission on the network layer without end to end encryption we recommend encryption for sensible data on the next layer, the application layer, hence for all applications that use the opptain network. To encourage encryption we developed all necessary classes that all third party developers can easily use to encrypt and decrypt any data with symmetric and asymmetric encryption and put them into the library. The FileShipping application described in Section 4.5 uses these classes to encrypt transmitted files.

Field Name	Datatype	Description
_ID	long	used for database unique identification
packetId	String	hash to identify the packet
sender	String	MAC address of the sender
receiver	String	MAC address of the receiver, null if broadcast
receiverAttribute	String	attribute to represent the HiBOp Identiy Table
targetAreaLat	String	latitude for Meeting and Visits routing
targetAreaLng	String	longitude for Meeting and Visits routing
expiration	long	time to live in milliseconds since 1970
obtained	String	MAC address of device from which the packet was received
obtainedList	list of Strings	MAC address on the path from sender to current device
appId	String	identification of API application
message	String	field for ascii data
data	array of bytes	field for byte data
pathToFile	String	path to the file that contains main byte data
nameOfFile	String	original name of file
startOfFile	long	first byte of file to be transmitted
sizeOfFile	long	size of bytes to be transmitted
hashOfFile	String	SHA1 hash of file
created	long	point of time when Packet was created

Table 4.1: Definition of the Packet

4.3.1 Packet definition

The *Packet* class consists of the fields seen in Table 4.1. The fields *_ID*, *packetID* and *created* are used for identification of the *Packet* in different scenarios, the fields *sender*, *receiver*, *obtained*, *obtainedList* and *appId* are used for addressing the *Packet*, the fields *receiverAttribute*, *targetAreaLat*, *targetAreaLng*, *expiration*, *obtained* and *obtainedList* are used for routing in the network and the fields *appId*, *message*, *data*, *pathToFile*, *nameOfFile*, *startOfFile*, *sizeOfFile* and *hashOfFile* contain the user data and additional data.

All outgoing *Packets* are sent to the network application for transport and all *Packets* the network application is receiving are forwarded to the application that created it, identified by the field *appId*. If the application is installed on the device it will receive information about the incoming *Packet*.

4.3.2 Security

There are two different forms of protection we have to discuss. First there is the protection of the Wi-Fi network that we open by creating a mobile hotspot. Second, we need to protect the data from

people that are allowed to receive but not allowed to read the data.

Protection of the Wi-Fi Network

If we want to protect a Wi-Fi network we nowadays use one of the two Wi-Fi Protected Access (WPA) protocols. In our environment, in which for example two devices want to connect to each other while never having met before, a pre-shared key has to be used to protect the Wi-Fi network with WPA since a global authentication server is not available in the scenario of Opportunistic Networks. This key has to be the same for all devices and therefore it has to be saved on each device when installing the application. Since a rooted Android device 2.1.8 can read all data this password can not be protected from potential attackers. Nevertheless WPA protects the application and the device from being connected to by non-participants that automatically connect to non-secured Wi-Fi access points.

Protection of Transmitted Data

The network application itself does not offer any kind of encryption to fulfil the concept of Partition of Tasks And Responsibility, explained in Section 3.2.2. Third party applications can decide if they want to implement an encryption method themselves or use the toolkit library that offers two kinds of cryptography, symmetric and asymmetric. Additionally all third party applications must react to the random number generator problem on Android devices.

Symmetric Cryptography Within the toolkit library we offer symmetric block cipher in the form of the Advanced Encryption Standard [oST01]. We use cipher-block chaining (CBC) mode, described in [EMST78] and [Dwo01], because it has the optimal ratio between speed and safety, which is secured by random and therefore unpredictable initialisation vectors, and PKCS#5 padding [Kal00]. The secret key, with the highest possible key length of 256 bit, that is used for encrypting and decrypting the data is hashed with the PBKDF2 function and the HMAC-SHA-1 message authentication scheme described in [Kal00] and [KBC97].

The process of encrypting a file with the AES cryptosystem of the opptain library is as follows. First we have to generate a salt and an initialisation vector (IV) and use the passphrase that the user entered to created a salted hash. With the salted hash and the IV the file is encrypted with AES. The file is sent along with salt and IV to the destination. At the destination salt and the newly entered passphrase are used again to create the same salted hash as on the sender's side to decrypt the file with the salted hash and the IV.

Asymmetric Cryptography Further we offer the RSA cryptosystem [JK03] for asymmetric cryptography. We use the electronic codebook (ECB) mode, since it is the only one that Android is offering, and PKCS#1 padding [JK03]. While the ECB mode is insecure when encrypting blocks with repeating byte sequences we can safely use it since our RSA cryptosystem only encrypts and decrypts randomly generated keys.

The process of encrypting a file with RSA is more complicated but serves another purpose. While with symmetric cryptography we do not have to exchange a secret before the process of file transmission, with asymmetric encryption we have to create pre-shared keys to be able to encrypt the data on the senders side and to decrypt the data at the destination. The public and private keys should be stored securely on the device. Instead of using a passphrase to encrypt the data, a random secret session key is generated and AES is used to encrypt the data. Then the public key of the receiver is used to encrypt the session key and the encrypted session key and the IV that was used for AES are sent to the destination. There the encrypted session key can be decrypted with the private key of the receiver and with session key and IV the data can be decrypted, too.

Random Number Generator After the compromise of a bitcoin transaction [Goo15b] the Android developer blog printed a quick fix for the random number generator that every application that generates random numbers should run at start. The quick fix is built into the toolkit applications and is available in the toolkit library for other applications. It is not known if Android fixed the problem in the system itself whereby making the quick fix obsolete.

Protection of Local Data

Local data is protected by Android. All data that belongs to an application can only be read and written by the application itself as long as no explicit permissions for other applications are set. As mentioned before the rooting of an Android device can give other applications nevertheless the possibility to read and write data that was secured without rooting.

4.4 opptain Network Application

The opptain network application is the core of the framework. It manages the network connections and represents the layer directly on top of the transport layer. All management functions are executed by a *Service* that can be started and stopped by the user in the main *Activity*. The *Activity* binds to the *Service* by sending an *Intent* to the system that starts the *Service* and connects it with the *Activity*.

A *Notification* appears in the notification bar that is required to remind the user of a *Service* running in the background. In the notification bar a click on the *Notification* opens the main *Activity* of the application for the user to stop the *Service* if wished.

The *Service* is a complex system of components that are used to initialise and manage connections, both for inter-device and inter-application connections, for filehandling and several other aspects that are listed in the following sections.

4.4.1 Wi-Fi Connection Manager

The *WifiConnectionManager* is a wrapper class that controls all connectivity over Wi-Fi. It can open tethering hotspots if the function is available on the device, scan for and connect to Wi-Fi access points and request Wi-Fi state information. It holds a *WifiManager* object that holds all functions for connectivity and a *WifiStateHelper* object that holds all Wi-Fi state functionality. The WifiStateHelper for example has methods for reading and evaluating the arp table because the *WifiManager* has limited informative content regarding Wi-Fi connection information.

Creating a Tethering Hotspot With our application two devices can connect if one device opens a Wi-Fi tethering hotspot and another device connects to this hotspot. Creating a tethering hotspot on an Android device is only intended for the operating system and the functions to start and stop the hotspot are hidden from the official API. As mentioned in 2.1.7 we use our own platform version to compile the application and on the device all functions are now available. Listing 4.1 shows how a hotspot is started. The tethering hotspot is created by specifying a *WifiConfiguration* and to start the hotspot with the now available function *setWifiApEnabled* with the configuration object and the boolean value *enabled* set to true. The function returns a boolean value that concludes the operations success. A failing value is an indication of lack of tethering functionality on the device. The *WifiConfiguration* needs to be filled with the SSID of the hotspot to create, and with a passphrase to secure the WPA-encrypted connection. We use Open System authentication that is required for WPA. The operating system requires the Wi-Fi functions to be disabled before a hotspot can be started. The SSID is a concatenation of the SSID-prefix “P2PHOTSPOT-” and the MAC address of the device for the identification of the hotspot. The passphrase is used just for obscurity reasons as explained in 3.2.

The function *setWifiApEnabled* can close the hotspot too by setting the enabled parameter to false.

Listing 4.1: Methods for opening and closing a tethering hotspot

```
private WifiManager mWifiManager;
private static final String PASSWORD_OBSCURITY = "jfk4qfjc4m94t8c4t8";

public boolean openHotspot(String hotspotName) {
    mWifiManager.setWifiEnabled(false); // Wi-Fi has to be disabled for opening hotspot
    WifiConfiguration wifiConfiguration = new WifiConfiguration();
    wifiConfiguration.SSID = hotspotName;
    wifiConfiguration.preSharedKey = PASSWORD_OBSCURITY;
    wifiConfiguration.allowedAuthAlgorithms.set(WifiConfiguration.AuthAlgorithm.OPEN);
    wifiConfiguration.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.WPA_PSK);
    boolean enabled = true;
    return mWifiManager.setWifiApEnabled(wifiConfiguration, enabled);
}

public boolean closeHotspot() {
    boolean enabled = false;
    WifiConfiguration wifiConfiguration = null;
    return mWifiManager.setWifiApEnabled(wifiConfiguration, enabled);
}
```

Connection to Access Points The *WifiConnectionManager* can scan for Wi-Fi networks in range and return results about all networks with all necessary information for connecting to it. After the application scanned for access points it sorts out all tethering hotspots and chooses up to one hotspot for connecting according to 4.4.5. Listing 4.2 shows how the connection to the hotspot is established. The SSID and BSSID are according to the scanned information, all protection settings and the passphrase are identical to the ones on the hotspot creating side of the connection. In contrast to the function that enables a tethering hotspot the *connect* function returns directly and provides a result in form of an action listener callback.

Listing 4.2: Methods for connecting to and disconnecting from an access point

```
private WifiManager mWifiManager;
private static final String PASSWORD_OBSCURITY = "jfk4qfjc4m94t8c4t8";

public void connectToAccessPoint(final String accessPointName, final String bssid) {
    WifiConfiguration wifiConfiguration = new WifiConfiguration();
    wifiConfiguration.SSID = "\"" + accessPointName + "\""; // quotation marks are
        necessary
    wifiConfiguration.BSSID = bssid;
    wifiConfiguration.preSharedKey = "\"" + PASSWORD_OBSCURITY + "\"";
    wifiConfiguration.allowedAuthAlgorithms.set(WifiConfiguration.AuthAlgorithm.OPEN);
    wifiConfiguration.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.WPA_PSK);
    ActionListener actionListener = ...; // action listener for results
    mWifiManager.connect(wifiConfiguration, actionListener);
}

public void disconnectFromAccessPoint() {
    mWifiManager.disconnect();
}
```

4.4.2 API

An API is provided for other applications to use the network provided by the network application. Like an IP packet or a TCP segment we offer a datagram structure for our network application that allows third party applications to have an interface for the data they want to send and receive. The structure of this *Packet* is given by the Toolkit Library described in Section 4.3 and the *Packet* can be exchanged between the applications through inter-process communication by using *Intents*.

A *Packet* can hold a reference to a file on the file system for transmission to another device. Therefore all byte data that has to be exchanged can be stored in a file to forward the data from the API application to the network application. A *FileManager* is used for opening streams to the files, for renaming and for creating files when receiving them over other devices.

4.4.3 Exchange of Data

When two devices connect to each other there is one device that acts as Server and one that acts as Client. The Server device opens a *Socket* and waits for incoming connections. The Client connects to this *Socket*. The only difference between Server and Client after they are connected is that the Client starts transmitting a *Handshake* packet to which the Server replies with its own *Handshake* packet. After the receiving of the *Handshake* packet both devices start transmitting all packets that

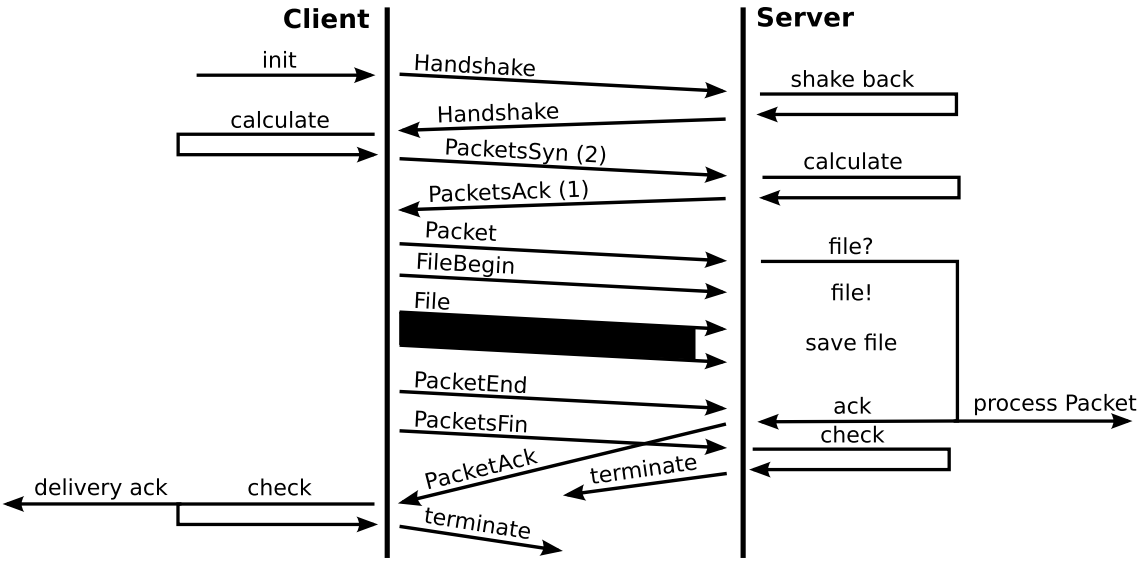


Figure 4.2: Exchange of data between smartphones according to the given protocol

the *NetworkRoutingManager* calculated for transmission.

The *Dispatcher* thread queues all objects that are ready for transmission and all objects that have been received by the opposite. While the queue is processed the Dispatcher thread sends appropriate objects like *ACK* or *FIN* packets to the other device. If both devices have received the *FIN* packet they close the *Sockets* and the device that is Client disconnects.

Figure 4.2 illustrates the sequence of exchanged packets between two devices. The sequence is defined by a protocol. For clearness it only shows the traffic initiated by the *PacketsSyn* of the Client. The Client has two *Packets* that are suitable for the Server. The Client sends a *PacketSyn* with the IDs of the two *Packets*. The Server receives it, looks up in its *Database* which of these *Packets* are needed and sends back a *PacketsAck* that now only contains the ID of the one *Packet* that is needed. The Client receives the *PacketsAck* and sends the *Packet* to the Server. Because the *Packet* contains a file a *FileBegin* and the bytes of the file are sent back to back to the Server. Thereafter a *PacketEnd* is sent to signal the end of the *Packet*. The Server processes the received *Packet* in an appropriate way and sends a *PacketAck* back to the Client. After the Client sent its last *Packet* a *PacketsFin* is sent to signal that the Client has sent all data. The Server receives the *PacketsFin* and checks if all its data is sent, too. If all data is processed the connection is terminated. In the given example the *PacketSyn* contains two files that are considered for transmission however the Server only has need for one of those and the *PacketAck* gives the Client the information which *Packet* to send.

Only if a device has received all *PacketAcks*, which concludes that the opposite has received all *Packets*, and the *PacketsFin*, and itself has sent back all *PacketAcks* to the opposite, the connection can be terminated.

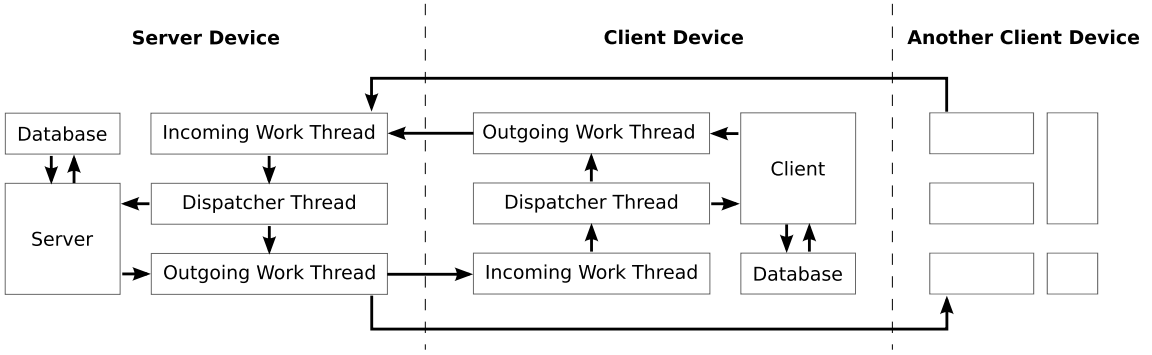


Figure 4.3: Exchange of data between applications

Incoming and outgoing data is handled by several work threads that are created when devices connect to each other. When a device is in the Server stage or when a device in Client stage connects to a hotspot a *DispatcherThread* is started along with an *IncomingWorkThread* and an *OutgoingWorkThread* for incoming and outgoing data respectively.

Figure 4.3 shows how the flow of data is processed in the network application. Both *Server* and *Client* get the received data inside the *IncomingWorkThread* and process them through the *DispatcherThread*. According to the received Protocol part either a answer is sent to the connected device or the main thread that is labelled as “Server” or “Client” is notified of the incoming protocol packet. The appropriate answer is calculated with the help of the *RoutingManager* and the *Database* and sent to the *OutogingWorkThread* for transmission to the other device. At the same time several Client devices can be connected to a Server device.

4.4.4 Stage System

The Android operating system does not offer functionality for peer to peer connections. Therefore the network application uses a sequence of stages to offer a possibility to let user devices connect to each other. The devices are either in a hotspot (or Server) mode or a peer (or Client) mode. The application resides always in one of the following three stages.

Init stage In Init stage the network application scans in its immediate vicinity for available hotspots. Then the Client-Server-Decision-Mechanism is used to determine if the Client stage or the Server stage is the next one.

Client stage In Client stage the network application connects to the hotspot that was found in Init stage. After connecting there is exchange of data with the other device according to Section 4.4.3.

After the connection was successful or an error occurred while transmitting the connection is closed and the application goes back to the Init stage.

Server stage In Server stage a hotspot is created by the network application to which other devices can connect. After there is an incoming connection there is exchange of data with connected devices. The hotspot is open for a span of time defined in the settings by the user. If there are other devices still connected the Server will stay open and the countdown to the next decision is renewed, otherwise the Server stage ends and the Init stage is started again.

4.4.5 Client-Server-Decision-Mechanism

To prevent a device from connecting to a device that was already connected before and to decide if a device should act as Client or Server we developed the following mechanism. When two devices connect they save the MAC address of the opposite along with the time of connection and mark the connection as not yet successful. When the exchange of data was successful the devices mark the connection like that, otherwise the connection will stay unsuccessful.

When a device is scanning its surroundings there are two possible scenarios that affect this mechanism. First there is the possibility that no device in Server mode is in reach. The scanning device will change into Server mode and act as a hotspot for other devices. Second, there could be one or more Server mode devices in reach that the scanning device recognises. In this case the *packets* to be delivered and the last connection with every peer is taken into consideration. If at least one of the peers is the receiver for one or more *packets* the one with the most waiting *packets* is chosen as next Server to connect to. If there are no peers that will get delivered directly by address but at least one of the last connections was unsuccessful the one that was connected most recently is chosen as next Server to connect to. If there are no unsuccessful last connections but at least one peer without any connection the one that has the best signal strength is chosen as next Server to connect to. If all peers have been connected before and successfully at last, the one that was connected least recently is considered as next Server but only if the last connection is elapsed more than a certain threshold that the user can choose in the settings.

Listing 4.3 shows pseudocode of the Client-Server-Decision-Mechanism that summarises the mentioned process.

Listing 4.3: Client-Server-Decision-Mechanism

```
1  function decide_client_server_mechanism()
2      get potential_server_list by scanning
3      create empty waiting_packets_list, unsuccessful_server_list, never_connected_server_list, server_list
4
5      if potential_server_list is empty then
6          start_server()
7      else
8          for peer in potential_server_list do
9              if peer is direct receiver of one or more packets then
10                 add peer to waiting_packets_list
11             else if peer was unsuccessful last time then
12                 add peer to unsuccessful_server_list
13             else if peer was never connected before then
14                 add peer to never_connected_server_list
15             else
16                 add peer to server_list
17             end if
18         end for
19
20         if waiting_packets_list is not empty then
21             sort waiting_packets_list by increasing amount of packets
22             start_client() with last of waiting_packets_list
23         else if unsuccessful_server_list is not empty then
24             sort unsuccessful_server_list by increasing time
25             start_client() with last of unsuccessful_server_list
26         else if never_connected_server_list is not empty then
27             sort never_connected_server_list by increasing signal strength
28             start_client() with last of never_connected_server_list
29         else
30             sort server_list by increasing time
31             get potential_server by choosing first of server_list
32
33             if potential_server was not connected recently then
34                 start_client() with potential_server
35             else
36                 start_server()
37             end if
38         end if
39     end if
40 end function
```

4.4.6 Database

All Android devices have access to a built-in SQLite *Database* that is used by the operating system and can be used by any application. We use the *Database* to save all gathered information about connections and all *packets* that are to be sent over the network. The following *Database* tables are used.

Meeting Times In the *MeetingTimes* table we save which devices, identified by MAC address, we exchanged data with. The data is at least one week old.

Meeting Places In the *MeetingPlaces* database we save GPS information for the meetings that are saved in the *MeetingTimes* table if we could fetch a GPS location.

Attributes In the *Attributes* table we save the *Attributes* of the devices we connected with.

Meeting Summaries In the *MeetingSummaries* table we save the *MeetingSummaries* that are created of the times, places and the *Attributes* of all saved connections.

Packet In the *Packet* table we save the *Packets* that arrived from other applications for transmitting over the opptain network.

Packet Delivery In the *PacketDelivery* table we save the devices that received certain *Packets*.

Connections In the *Connections* table we save the SSIDs of the devices we connected to and if the connection was successful. The table is used by the Client-Server-Decision-Mechanism described in Section 4.4.5 for priority calculations.

4.4.7 Routing Manager

The *NetworkRoutingManager* consists of a connection to the *Database*, a *Dijkstra* module for calculating paths between sender and receiver of a *Packet* and the rules for routing in the network. Each time the device connects to another one, all information about the connection are processed through

the *NetworkRoutingManager*. All new *MeetingSummaries*, connection times, success information and *Packet* delivery information are inserted into the *Database*. When there is a new connection to a device the *NetworkRoutingManager* retrieves all *Packets* from the *Database* and calculates those that are relevant for the connected device. The calculation is done with the help of the rules that define the opptain routing scheme and the Dijkstra module that is used for managing the local view of the network.

The following rules for routing in the opptain network are defined, a *Packet* is either dropped, which means, that it will not be considered for transmitting, or it will be considered. The list is divided into negative and positive rules that are processed in the order shown below. Each *Packet* has to fulfil each negative rule to be processed by the positive ones and if they are fulfilled by one positive rule, they are considered for transmission. The final list of *Packets* will be transmitted, as a list of PacketIDs, to the connected device and the connected device will answer with the PacketID list of *Packets* that it has not already received before and hence should be transmitted.

The negative rules:

- If the *Packet* was already delivered to the connected device, the *Packet* is dropped.
- If the *Packet* was already delivered often enough due to the Spray and Wait routing scheme, the *Packet* is dropped.
- If the *Packet* was already delivered often enough due to the Spray and Focus routing scheme, the *Packet* is dropped.
- The *Packet* may have been routed through the connected device already, if so, the *Packet* is dropped.
- Each *Packet* has an expiration time, a Time-To-Live. If the *Packet* is expired, the *Packet* is dropped.

The positive rules:

- If the *Packet* is to be broadcast, it is considered.
- If the connected device is the receiver of the *Packet*, it is considered.
- If the connected device is part of a path to the receiver, according to the Dijkstra algorithm, it is considered.

4.4.8 Timer

A *Timer* is used to give a workaround due to the fact that many callbacks are needed to determine in which state, with regard to Wi-Fi connectivity, the Android operating system currently is. Because the callbacks do not arrive at the moment the state is changed or are not given at all the *Timer* has to be set at the beginning of several actions like the start of Init-, Server- or Client stage or connection starts to periodically check for changes.

4.4.9 Notification System

To give the user the possibility to disable the networks application temporarily a sticky notification stays in the notification bar. The user can navigate back to the application by pressing the notification to stop the application.

4.4.10 Settings

A *SettingsActivity* is used to give the user the possibility to change some variables that are used for connections and routing. In the development phase of the application it is not clear which timer lengths and connection timeouts are best for routing. Therefore the possibility to change these values was included into the settings. The user can set the *Attribute* that is used for the HiBOp routing scheme in the Settings. Also there is a value to be set how many logging information should be stored on the device.

4.4.11 Bus-based Property Change Listener

For refreshing classes, in which information has changed, through the application the Otto library is used to provide a system that goes without connecting all classes with each other just for maintaining information exchange. Although the Model-View-Controller pattern is applied, the Bus system is used for listening and reacting to specific changes in the application.

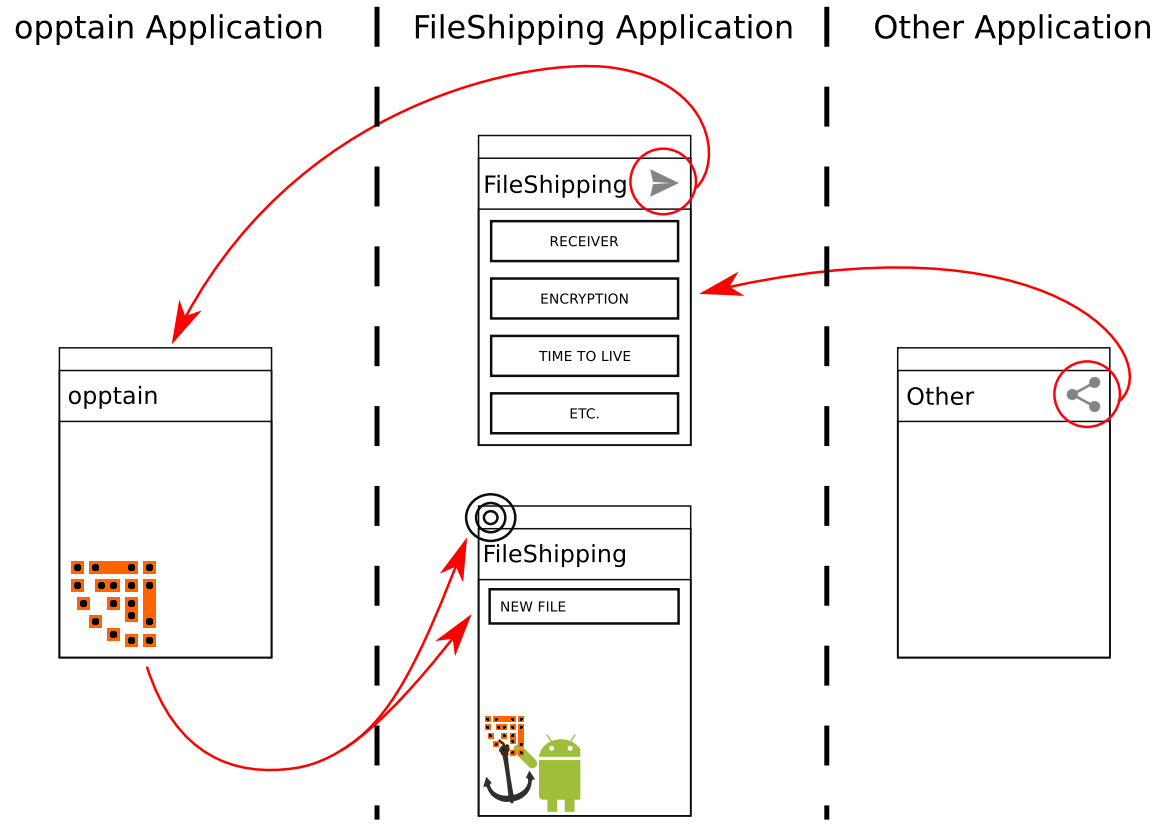


Figure 4.4: The interaction of the FileShipping application with other applications

4.5 FileShipping Application

We present a usecase in the form of a filesharing application called FileShipping that is able to send files to other devices and offers a distribution function with which other devices can request files. The course of the FileShipping application in interaction with other applications on the device is illustrated in Figure 4.4.

The FileShipping application contains five views in the form of *Activities*. The first *Activity* is the *PacketsActivity* that is also the launcher *Activity* which means that the user can start this *Activity* by pressing the FileShipping application icon, the second is the *OutgoingPacketActivity* that is created when the application receives a “SEND” *Intent*, the third one is the *SettingsActivity* that can be started by pressing the *Settings* item in the options menu of the *PacketsActivity*. There are also two *Activities* provided by the ZXing Library to show and scan QR codes. Additionaly there is a *IncomingPacketService* that is used as a entry point for *Intents* for incoming *Packets*, and for creating *Notifications*.

4.5.1 PacketActivity

The *PacketActivity* contains an *ActionBar* and a *ContentView*, the *ActionBar* shows the name of application and *Activity* and shows the *OptionsMenu*, the *ContentView* shows the history of received Packets. The *OptionsMenu* contains menu items for the following tasks:

- opening the *Settings* menu
- scanning QR codes and therefore saving MAC addresses and public keys
- showing an *Activity* to show the QR code with the own public key for others to scan
- clearing the history

If a new Packet is received by the application it will be shown in the history and a click on the item fires a "SEND" *Intent*. The user can choose an application to open the received file. A long click on the item opens a context menu that enables the user to remove the single item from the history. The *PacketActivity* is the main *Activity* that is launched when the user clicks on the application icon or a *Notification* that is created by the application.

4.5.2 SettingsActivity

The *SettingsActivity* is a wrapped *PreferencesFragment* which is a *ListView* that contains all preferences. The preferences of the application can be changed and are saved in the user directory of the application so that only the application has access to them. In the development phase of the application there are no active settings to change. As future work settings for file path choosing and preferences for chunking of files are imaginable.

4.5.3 QRCodeShowActivity

The ZXing Library contains functionality to create, show and scan QR- and barcodes. We create a QR code that represents MAC address and public key and show it to the user with a modified *Activity* of the library.

(a) Share icon

(b) Send icon

Figure 4.5: Share and Send icon

4.5.4 QRCodeScanActivity

We use the scan functionality of the ZXing Library to scan QR codes of other devices and save their MAC addresses and public keys on the device. We modified the *Activity* of the ZXing Library to work in portrait mode and to scan only QR codes.

4.5.5 OutgoingPacketActivity

Any Android application that implements the "SHARE" functionality represented by the icon shown in Figure 4.5a can be used to send a "SEND" *Intent* to other activities. The user chooses a file, that may be anything like picture or text document and presses the "SHARE" button. Now the user can choose between applications that can receive the "SEND" *Intent*. The FileShipping application is one those. The *OutgoingPacketActivity* is the *Activity* that receives the *Intent* and its view can be used to edit the information that is necessary to send the file to other devices. The user can choose via checkbox if the file should be sent as a broadcast or to a specific receiver. The receiver is identified via MAC address. The user can choose the receiver MAC through a contact where the MAC address is saved or type it in. The next option to choose is the encryption method if wanted. Via checkbox the user can choose if he wants to have encryption at all. After enabling encryption the choices of symmetric and asymmetric encryption is given. Symmetric encryption can always be chosen and has to be concluded by entering a password. Asymmetric encryption is only available if a specific receiver is chosen and a public key for this user is saved. Also additional information can be used to send the file in the right direction by typing in target area coordinates or an attribute that represents the HiBOp Identity Table. After the editing the user presses the "SEND" button shown in Figure 4.5b to send the information as a *PacketIntent* to the system. The opptain application should be the only application to receive this special kind of *Intent* whereas there should be no dialogue to chose the destination of the *Intent* this time. The *Activity* disappears with the sending of the *Intent*.

The process of Alice sending a photo to her friend Bob over opptain could be like this for Alice and Bob respectively:

Alice opens a gallery application and chooses a picture she would like to send to Bob. She presses the “SHARE” button and chooses our FileShipping application. The *OutgoingPacketActivity* of the FileShipping application opens and Alice chooses Bob as the receiver. Because she has saved Bob’s public key beforehand she now can use public key encryption. Alice knows that Bob works at the University of Duesseldorf, so she puts “HHU”, an acronym, in the attribute field. Now she clicks on the “SEND” button and the editing *Activity* disappears. As long as opptain is running in the background Alice has to do nothing more. The opptain application receives the *Intent* and starts a background service to add the *Packet* to the list of *Packets*. opptain does not open any window for this, Alice should see the *Activity* that she used for SHARING the file in front of her.

Some time later on Bobs device, the opptain application will receive the *Packet* that is supposed to be delivered to Bob. After receiving the *Packet*, opptain will create an *Intent* to notify Bob’s FileShipping application that a new *Packet* has arrived. Bob’s FileShipping application receives the *Intent* that contains all necessary information to decrypt the file and starts a background service to decrypt. A *Notification* appears in the notification bar. Bob clicks on the *Notification* or simply starts the application which opens the *PacketActivity* that shows the received file on top of the history. Bob can now click on the entry to choose a application to present the photo.

For Alice and Bob not visible, also Charlotte, David and Eve get in contact with the *Packet*. The devices of Alice and Charlotte connect and while there are no meetings with Bob in Charlotte’s *MeetingSummaries* there are entries about David and Eve that have the same attribute, “HHU”. The *Packet* is transmitted from Alice’s device to Charlotte’s. Charlotte’s device is later connecting with David’s and Eve’s. Since the last time Charlotte has connected with them there were several contacts with Bob, so the *MeetingSummaries* of David and Eve forecast high delivery probability. The *Packet* is sent to David’s and Eve’s device. While David will later connect with Bob and transmit the *Packet*, Eve wants to have a look at the photo. However, because the photo is encrypted, she can not.

4.6 Installation

The API for managing tethering hotspots is available on Android devices from Jelly Bean on as mentioned in Section 3.2.1. We therefore developed our applications to work only on devices that Android version 4.1 or higher by setting the requirement attribute to this version.

For installation the applications are published and delivered as a compressed installation file called *opptain.apk* and *fileshipping.apk*. Before publishing the applications have to be signed either in debug mode while developing the application or in release mode when the application is to be distributed. Since the application is not to be distributed in an application store or otherwise we stay in the devel-

opment phase and let the application be signed by the IDE we use for developing and debugging. The application can be installed by opening the Eclipse IDE, importing the application folders and run the application on an emulator or on a connected Android device. Running the application on an emulator works but there will be no network functionality available. The Android device has to be configured for debugging applications, the configuration process is different for each device and can be found in the respective manual of the device.

As mentioned before Android is used by many manufacturers. We do not know if there are any manufacturers that changed the hidden API of the Android platform and are thus not able to use the hotspot functionality as expected by us.

Chapter Conclusion

In this chapter we documented the implementation of our applications. We started with an overview of the development environment, in which we worked, in Section 4.1. We divided the tasks and responsibilities of our framework into a library and two applications as seen as in Section 4.2.

The library is explained in Section 4.3 and offers the *Packet* definition that is part of the API that connects the applications and is used for the transmission between devices, too. Also there are classes for encryption and decryption in the library for third party applications to use.

The Opportunistic Network is provided by the opptain network application. The implementation details were listed in Section 4.4. The network application manages the network and handles all connections between devices and administrates the routing.

The FileShipping application was developed as a proof of work for the network application. The details of this application were given in Section 4.5. The application enables filesharing for the user over the network provided by the network application. Features of the application are transmission of files, encrypted or not, and the distribution of files.

The Section 4.6 explains how the applications can be installed on test devices.

In the next chapter we will evaluate these applications with regard to several aspects.

Chapter 5

Evaluation

In the following chapter we show the results of our tests and evaluations of the developed applications. We tested different aspects of the opptain network application and the FileShipping application. We tested the ability to open and connect to hotspots as well as the behaviour while the applications are running with regard to battery life and memory usage. Another interesting issue to evaluate was the transmission range in which devices are able to connect and transmit data. Tests inside and outside were conducted. In Section 5.4 we tested if there is a traceable correlation between the behaviour of the opptain network application and different preconditions and settings. Further aspects tested are the simultaneous use of the opptain network application with other functions of a smartphone, the transmission speed and the encryption and decryption speed. In the last section the results of testing our routing scheme are given. The creation of *Meetings* and *MeetingSummaries* as well as the exchange of *MeetingSummaries* were tested. Also the routing of *Packets* with and without *Attributes* which can be used to help the routing was evaluated.

We used the Android devices listed in Table 5.1 for testing the opptain network application and the FileShipping application.

5.1 Connectivity

First we tested the opptain network application on all test devices to determine if the main functionality i.e. the connectivity between devices can be established at all. The test devices include smartphones and tablets with different Android versions in range from 4.3 to the newest 5.0.1 version, we tested tablets with and without a GSM/UMTS/LTE module. Table 5.2 shows the test devices with additional information regarding kernel, baseband version and build number.

The graphical user interface of the Android operating system on tablets without a GSM/UMTS or

Alias	Name	Model	Manufacturer	Type	Version
$\mathbf{N}_{7B}^{08}$, $\mathbf{N}_{AE}^{1F}$	Galaxy Nexus	GT-I9250	Samsung	Smartphone	4.3
$\mathbf{O}_{3D}^{66}$, $\mathbf{O}_{F5}^{05}$	One	A0001	OnePlus	Smartphone	4.4.4
$\mathbf{P}_{4E}^{1A}$	Google Nexus 4	LG E960	LG	Smartphone	5.0.1
$\mathbf{Q}_{6E}^{11}$	Google Nexus 7	ASUS-1A019A	Asus	Tablet	4.4.4
$\mathbf{R}_{CC}^{08}$	Google Nexus 9	OP82100	HTC	Tablet	5.0.1
$\mathbf{S}_{12}^{17}$, $\mathbf{S}_{75}^{60}$	Google Nexus 9	OP82200	HTC	Tablet	5.0.1

Table 5.1: All Android devices for testing

LTE module does not allow to open a tethering hotspot because in the philosophy of Android there is no need for a connection of devices over Wi-Fi if there is no possibility to forward mobile internet connection. However the API of Android allows to open the hotspot programmatically. The connection functionality should be given on all devices regardless of the presence of GSM/UTMS or LTE modules and therefore on all our test devices.

The test was successful for all smartphones and the Nexus 7 tablet that runs Android 4.4.4, both for opening a tethering hotspot and connecting to one. However the analysis of the debugging logs showed significant differences in the internal process of opening tethering hotspots and scanning for access points on different devices. This takes its origin from different Android versions and manufacturers, and results in different delays when opening a hotspot or connecting to an access point. On the Nexus 9 tablets there were mixed results. All tablets could successfully create a hotspot but tests over several weeks showed inconsistent results concerning the connection to a hotspot. There was no correlation between different operation system settings and the ability to connect to an access point and receiving a callback regarding the connection.

Table 5.3 shows the results of the test. We give an example of a typical time span for opening a hotspot or connecting to one and refer to Section 5.4 in which we show that even same devices have different values for connection times.

5.2 Battery Life and Memory Usage

The battery life and memory usage of the device are important for the application's ability to run in the form we desire. We tested the battery life and memory usage on three different devices to determine how long the device could run when the network service is enabled and if the memory usage stays

Alias	Kernel	Baseband-Version	Build-Number
$\mathbf{N}_{7B}^{08}$, $\mathbf{N}_{AE}^{1F}$	3.0.72-gfb3c9ac	I9250XXLJ1	JWR66Y
$\mathbf{O}_{3D}^{66}$, $\mathbf{O}_{F5}^{05}$	3.4.0-cyanogenmod-gd8c0761	MPSS.DI.2.0.1.c7	KTU84Q
$\mathbf{P}_{4E}^{1A}$	3.4.0-perf-g16e203d	M9615A-CEFWMAZM-2.0.1701.05	LRX22C
$\mathbf{Q}_{6E}^{11}$	3.4.0-g03485a6	(no baseband available)	KTU84P
$\mathbf{R}_{CC}^{08}$	3.10.40-ga3846f1	(no baseband available)	LRX22C
$\mathbf{S}_{12}^{17}$, $\mathbf{S}_{75}^{60}$	3.10.40-ga3846f1	0.07.30.1015_2_0047	LRX22C

Table 5.2: Testing devices with kernel, baseband and build information

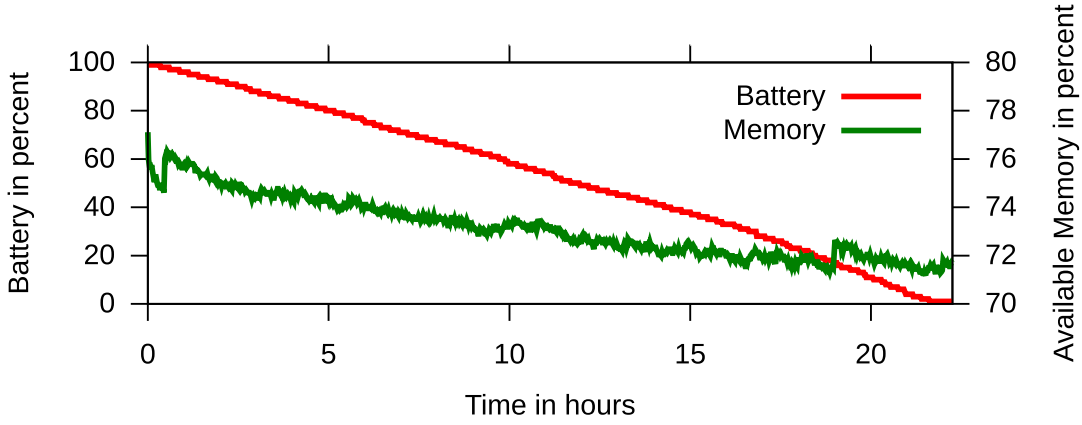
Device	GSM/UMTS	Server	Server time	Client	Client time
$\mathbf{N}_{7B}^{08}$	yes	yes	2.399	yes	2.542
$\mathbf{N}_{AE}^{1F}$	yes	yes	2.238	yes	2.228
$\mathbf{O}_{3D}^{66}$	yes	yes	12.113	yes	8.219
$\mathbf{O}_{F5}^{05}$	yes	yes	2.526	yes	2.304
$\mathbf{P}_{4E}^{1A}$	yes	yes	12.390	yes	0.821
$\mathbf{Q}_{6E}^{11}$	no	yes	4.964	yes	5.097
$\mathbf{R}_{CC}^{08}$	no	yes	0.853	partial	
$\mathbf{S}_{12}^{17}$	yes	yes	0.848	partial	
$\mathbf{S}_{75}^{60}$	yes	yes	0.896	partial	

Table 5.3: Results for testing connectivity functionality

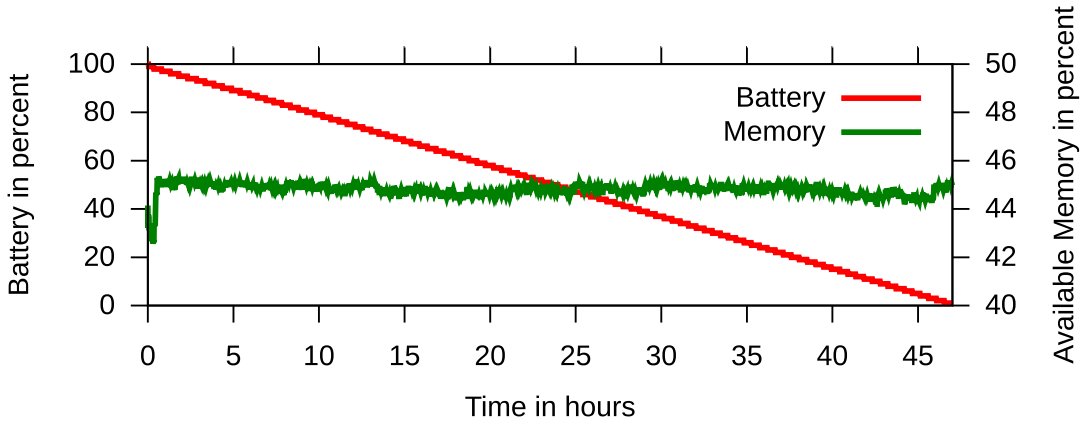
constant over the time.

5.2.1 Battery Life

Since the application is mostly using Wi-Fi either when hosting a hotspot or searching for access points nearby it is consuming much energy. The most energy consumption happens in the phase in which the tethering hotspot is enabled. This is because hosting a hotspot means to send signals out which takes more energy than receiving signals on the device itself. We tested the life span of the device when the application is running without devices nearby hence always being in server mode and having the hotspot enabled. There were no other applications running except for system applications



(a) Battery and available Application Memory of device $\mathbf{P}_{4E}^{1A}$



(b) Battery and available Application Memory of device $\mathbf{S}_{12}^{17}$

Figure 5.1: Battery and available Memory

that run in the background all the time. Also there was no mobile data connection enabled and the devices' screen was turned off for almost the entire test. Only to check if the device was still running without any problems the screen was turned on shortly.

Figure 5.1 shows the battery lives of the devices $\mathbf{P}_{4E}^{1A}$ and $\mathbf{S}_{12}^{17}$. As expected the energy consumption is a linear process. We can see that the older device $\mathbf{P}_{4E}^{1A}$ that has been used for over two years daily and has a weaker battery pack can live up to 22 hours under the before mentioned conditions. The newer device $\mathbf{S}_{12}^{17}$ that has never been used before and has a stronger battery pack can live up to 47 hours. We ran the battery test several times on the $\mathbf{N}_{7B}^{08}$ device, too, but the device always crashed

after several hours due to a fatal exception in a system process as seen as in Listing 5.1. The logs show that the class that handles tethering state changes raises the exception and is a error of the operating system and not related to our application. The problem was probably fixed by the Android developers since the newer devices, as seen in the Figure, run the application till the battery dies.

Listing 5.1: Exception

```
1 E/AndroidRuntime(11629): *** FATAL EXCEPTION IN SYSTEM PROCESS: NetworkStats
2 E/AndroidRuntime(11629): java.lang.RuntimeException: Error receiving broadcast Intent
      [...]
3 E/AndroidRuntime(11629):  at android.app.LoadedApk\$ReceiverDispatcher\$Args.run(
      LoadedApk.java:773)
4 E/AndroidRuntime(11629):  at android.os.Handler.handleCallback(Handler.java:730)
5 E/AndroidRuntime(11629):  at android.os.Handler.dispatchMessage(Handler.java:92)
6 E/AndroidRuntime(11629):  at android.os.Looper.loop(Looper.java:137)
7 E/AndroidRuntime(11629):  at android.os.HandlerThread.run(HandlerThread.java:61)
```

Battery Life Consideration

If the device has unsuccessfully searched for hotspots and the user is not physically moving at this time, the probability is high that other hotspots will come to the user rather than the user is going to other devices. Therefore a moving device should concentrate more on hosting a hotspot and a resting device should concentrate more on looking for hotspots. With consideration of those facts, motion provided by GPS positions or accelerator data can be used to save battery.

5.2.2 Memory Usage

The network application is a powerful tool that has to keep much information in memory to work properly. We ran the application several times through a memory usage analysing tool to look for potential memory leaks. Although we did not find any leaks, we used the battery tests to simultaneously monitor the memory as well.

Figure 5.1 also shows the memory usage of the devices $\mathbf{P}_{4E}^{1A}$ and $\mathbf{S}_{12}^{17}$. On the device $\mathbf{P}_{4E}^{1A}$ after the setup of the application the available memory for the application decreases from 76% to 71.7% over the 22 hours. The test with device $\mathbf{S}_{12}^{17}$ however shows no decrease of memory usage over the 47 hours, the available memory remains at 45% after the initial setup.

5.3 Transmission Range

We tested the maximal range between devices to still be able to maintain a connection and exchange the *Handshake*. We tested two different scenarios. In a building with obstacles and several other Wi-Fi access points active nearby, and outside without obstacles between the two devices and with only a few access points nearby.

5.3.1 Outside tests

The scenario outside without obstacles and with only a few access points can be considered as almost ideal. In the test for this scenario we documented if a connection was successfully created. Also the transfer time for the handshake was measured. Conducting the test great fluctuations of the maximal range occurred depending on the models involved in the connection.

Figure 5.2 shows the connection time span for the different combinations of test devices. In the case the connection was unsuccessful the measurement was repeated to ensure that it was not an one-time error but that the range was the actual reason for the unsuccessful connection.

The tested Nexus devices showed a significantly higher range than the tested OnePlus devices. Testing the connection between the two Nexus devices a maximal range of 40 meters was reached. On the contrary the two OnePlus devices did not create a connection successfully above 10 meters. In addition both the range of a Nexus device as a server and the range of a Nexus device as a client were higher. This was shown in the tests between different models. In the test where a Nexus device operated as a server a range of 15 meters was reached whereas the test with a Nexus device as a client only succeeded up to a range of 5 meters.

The maximal range difference between different devices may take their origin in the choice of manufacturers that keep the transmission range low to save energy and therefore battery because the typical use case for the tethering hotspot is still the forwarding of Internet connectivity which usually takes only a range of a few meters.

With regard to the given test results the following statements can be made. First, a connection between a device with a high range and one with a low range shows in total a low maximal transmission range which results in a restricted mobility of the connected devices. Second, only when two devices with a high range connect a certain mobility is given. Also it can be assumed that the probability of two high range devices connecting is low. Therefore in most cases a low transmission range and mobility is given.

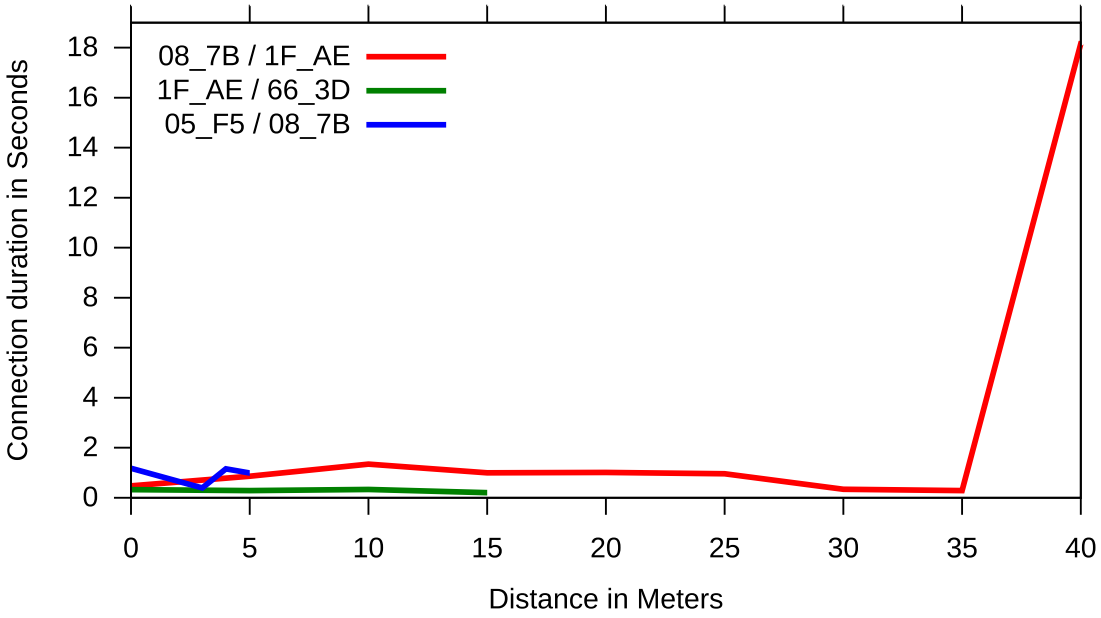


Figure 5.2: Connection duration outside without obstacles with regard to distance

5.3.2 Inside tests

The scenario inside with obstacles and with several other access points active nearby is the non-ideal one. In this test we documented if the connection was successfully created and how long it took to transfer a five megabyte sized file between two devices. By that we can draw conclusions about transmission range and also transmission speed which we will discuss in Section 5.6. We chose to take the two Nexus devices for the test since those showed the best test results regarding range and connectivity.

Figure 5.3 shows the transfer time depending on range and amount of walls between the two test devices. The connection was successful for the shown ranges and amounts of walls. The transmission tests within a range of 20 meters and two walls were unsuccessful. We tested each of the cases at least three times to see if the results are constant. In each case the transmission of the five megabyte sized file completed when the connection was successful except for one 20 meter test in which the connection was established once but did not conclude successfully with the transmission of the file.

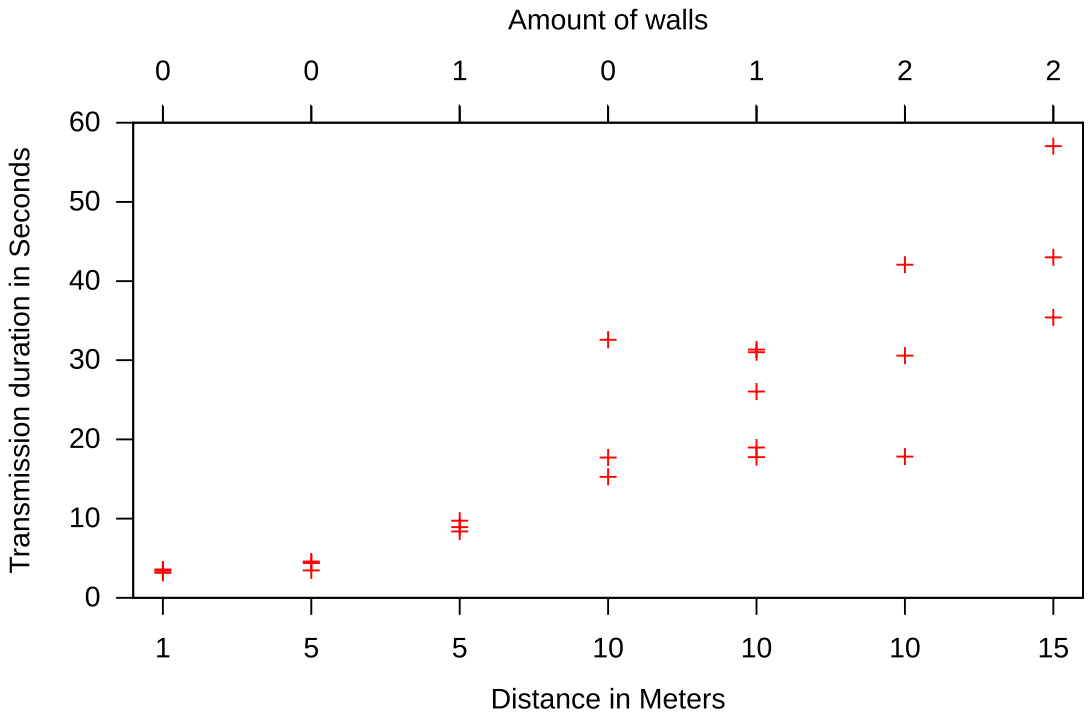


Figure 5.3: Transmission duration inside with obstacles with regard to distance and amount of walls

In this scenario, too, we see that a higher range between two devices results in higher connection and transmission times. The additional obstacle aspect shows that walls between the two devices interfere with the transmission, too. Compared to the test of the two Nexus devices outside, the test that was conducted in a building showed that the maximal range to connect to each other decreased from 40 meters to 15 meters.

This shows that the transmission of files is successful for scenarios in which the two devices are separated either by small distance but walls in between which represents adjacent rooms or even adjacent houses, or by long distances and less obstacles in between.

5.4 Determinism

In the progress of implementing and testing the opptain network application there have been several cases in which the two Samsung Nexus devices behaved differently though being the same model with

same Android version and settings and having the same application version installed.

We examined this occurrence under test conditions with several devices. Even after resetting these devices with the Android own factory reset method and starting the application under the same conditions there are major behavioural differences between the background processes of the device. One of those results in different starting times for tethering hotspots. Between the moment the Init stage starts and the launching of the tethering hotspot there are up to three Wi-Fi related actions that the device has to execute:

1. enabling Wi-Fi because Wi-Fi has to be on to scan for access points

2. scanning for access points

3. disabling Wi-Fi because Wi-Fi has to be off to launch the tethering hotspot

We tested the span of time between the Init stage start and the tethering hotspot launch both for the case that Wi-Fi was disabled and enabled before. In the first case all three steps mentioned before had to be executed, in the second case only the last two. The time spans for all cases are shown in Figure 5.4.

As we can see the time spans vary significantly both on the same device and on two devices, that are the same model, each under the same preconditions. From the procedure explained above we can assume that the case in which Wi-Fi was already enabled before the process of the launching would be faster, however in the tests conducted no clear coherence of this kind occurred.

5.5 Simultaneous use of opptain and other functions

As we pointed out in Section 3.1 it is important that the user of our applications is minimally bothered in her routine. For verification that we fulfilled this demand a short test of the application while using other functions of the smartphone is necessary.

First we tested the behaviour of our application during phone calls. There was no interference as a phone call could be made and received without any problems while the application connected to hotspots in the surroundings and opened a hotspot itself. We tested different cases. In one the opptain application was running on both the devices participating in the phone call. In the other case one device running the opptain application made a call to another device not running the opptain application. However other devices with an active opptain application were in the surroundings. In both tests no interference was found.

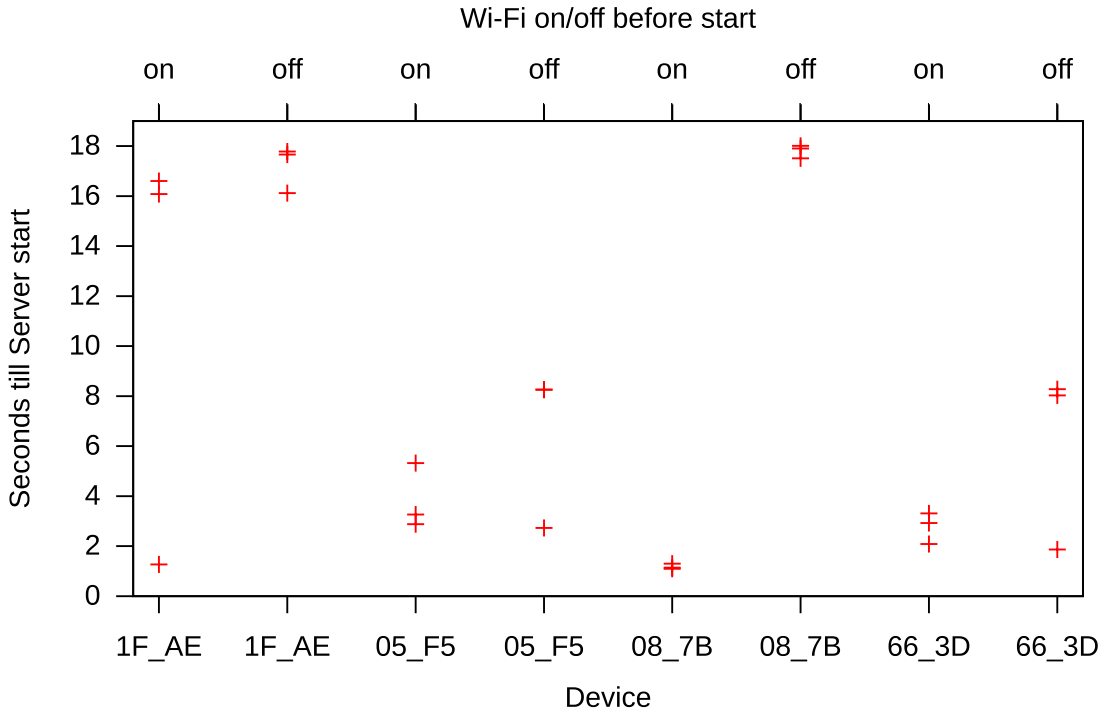


Figure 5.4: Determinism test.
Each mark represents the time of span between the Init stage start and the tethering hotspot launch on several devices both with disabled and enabled Wi-Fi

In a second test other applications were used while running the opptain application. In these test no interference was determined, either. The opptain application was able to host hotspots and to connect to other access points without any problems.

5.6 Transmission Speed

The transmission speed between two devices is a factor that has to be evaluated to give a hint about how fast devices can exchange data.

Since the devices connect over Wi-Fi we do not use a loss-free connection. TCP sockets are used for maintaining the connection which guarantees that data which is written on the socket on one device is read on the other device as long as the connection does not break.

We transmitted a fixed sized file between two devices in several different ranges to determine the transmission speed according to distance and obstacles. We chose the file size of five megabyte which represents a high quality photo on a modern smartphone for the test and used the scenario from before

where the devices are in a building in which other access points are nearby.

Figure 5.3 shows the time that the five megabyte file needed to be transferred from one device to another according to the given combinations of distance and amount of walls. We see that the transmission takes longer when higher distance and more walls are between the two devices. This is because the TCP layer has to repeat the transmission of TCP segments on the lossful channel. The transmission speed ranges from 11.75 Mbit per second at a distance of one meter to 0.7 Mbit per second at a distance of 15 meters and two walls between the devices.

The overall transmission speed ranges from 0.7 Mbit to 11.75 Mbit per second. Only when devices are directly next to each other the transmission reaches a speed that is suitable for the transmission of large data in our scenarios. While Wi-Fi is considered the fastest approach to transmit data on Android devices, there is also the fact to be considered that many Wi-Fi access points next to the devices can interfere with the connection and therefore slow down the transmission speed.

5.7 Encryption and Decryption Speed

With the opptain library we offer symmetric and asymmetric encryption for all third party applications that use the opptain network and use the encryption system in our FileShipping application. We tested how fast different devices can encrypt and decrypt different sized files. We used one device of each model we had for testing and used a 5MB and a 100MB file. Table 5.4 shows the test devices with additional information regarding CPU and memory.

Figure 5.5 shows the time that is needed to encrypt and decrypt the files with the AES and RSA cryptosystems. For the test all necessary functions for encrypting and decrypting are included in the test. The exact functions that are called in the process of encrypting and decrypting are listed in Section 4.3.2. We can see that older devices like the Galaxy Nexus need more time to encrypt and decrypt than the newer devices like the Google Nexus 9. The Google Nexus 9 device has a better processor and is therefore faster when encrypting files.

We also see that RSA is faster than AES though RSA uses AES to encrypt the file and then additionally encrypts the randomly generated key. This is due to the retrieving of the secret key for encrypting and decrypting that is only randomly generated in RSA which does not cost time and the salted hash function that has to create a key from a passphrase for the symmetric encryption. The receiver gets the salt and will enter the passphrase again, so the secret key has to be generated again on the receiving side. Therefore the symmetric decryption takes longer than the asymmetric decryption as well.

The symmetric encryption of an 5MB file takes between 231.9 and 1136 milliseconds on average de-

pending on the device with a standard deviation between 52.8 and 461 milliseconds. The asymmetric encryption takes between 96.1 and 964.4 milliseconds on average with a deviation between 8.1 and 164 milliseconds.

The symmetric decryption of the encrypted 5MB file takes between 200.5 and 1074.9 on average depending on the device, the standard deviation is between 69.3 and 477.2 milliseconds. The asymmetric decryption takes between 96 and 976.5 milliseconds with a deviation between 13.4 and 310.5 milliseconds.

We repeated the procedure with a 100MB file to see the relation between a smaller and a larger file. The time needed is not linear in the whole because the time for preparation of the keys, salts and IVs takes much time. After subtracting the preparation time from the whole time, the factor 20 is given and the encrypting and decrypting is linear to the file size.

The time for encrypting files is not as relevant for the FileShipping application as the value for decrypting because files get encrypted if the user chooses encryption when packing a file for transmission. When the user clicks the send button the encryption runs in the background and the user can go on using her phone. When the encryption has finished the Packet is sent to the opptain network application without the user noticing and is ready for transmission. The decryption of asymmetric files is done when the packet is received, once the file is decrypted the user gets the notification that a new file is delivered. Only in the case where the file was encrypted symmetrically the user has to enter the passphrase. This is done when the user clicks on the file the first time in the list of received packets because he is not forced to enter the passphrase the moment the packet arrives. So this would be the only one of the four cases where the user has actually to wait for the decryption of the packet to use the decrypted file.

These values actually would allow us to encrypt and decrypt byte data on the fly shortly before transmitting them instead of encrypting and decrypting them separately before and after the transmission process. Especially on the sending device the additional hard drive space that is used to save both the original and the encrypted version of a file can be reduced.

5.8 Routing

5.8.1 Exchange of Meeting Summaries

As explained in Section 3.2.4 the exchange of *MeetingSummaries* between devices is the condition for target-oriented routing in our network. In the next Section we tested in the setting proposed in

Alias	CPU	Memory
$\mathbf{N}_{7B}^{08}$, $\mathbf{N}_{AE}^{1F}$	1.2 GHz dual-core ARM	1 GB
$\mathbf{O}_{3D}^{66}$, $\mathbf{O}_{F5}^{05}$	2.5 GHz quad-core Krait	3 GB
$\mathbf{P}_{4E}^{1A}$	1.5 GHz quad-core Krait	2 GB
$\mathbf{Q}_{6E}^{11}$	1.51 GHz quad-core Krait	2 GB
$\mathbf{R}_{CC}^{08}$, $\mathbf{S}_{12}^{17}$, $\mathbf{S}_{75}^{60}$	2.3 GHz dual-core Denver	2 GB

Table 5.4: Testing devices with CPU information

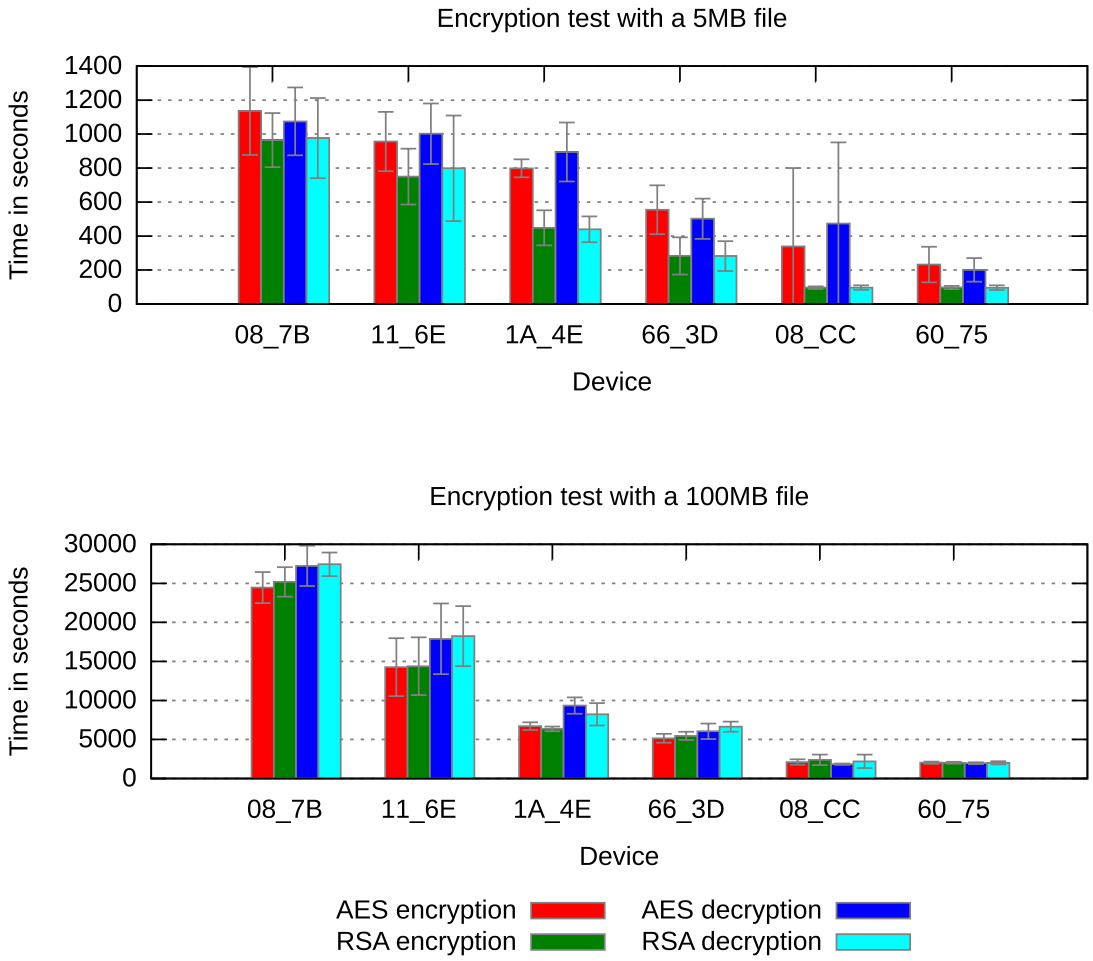


Figure 5.5: Encryption and Decryption time test

Figure 3.1 if the process operates as in theory.

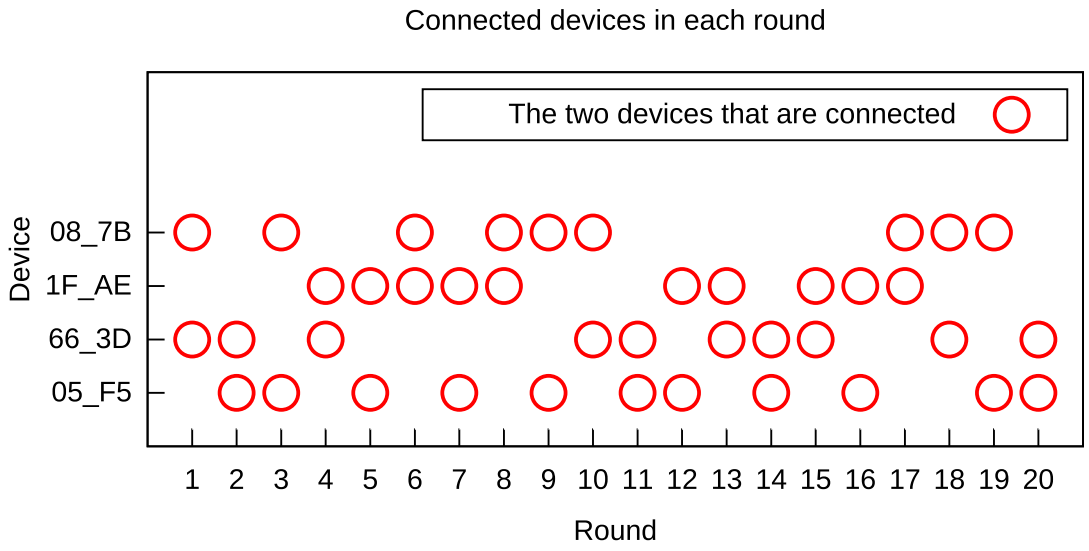


Figure 5.6: Prerouting Connections

We also used the devices $\mathbf{N}_{7B}^{08}$, $\mathbf{N}_{AE}^{1F}$, $\mathbf{O}_{3D}^{66}$ and $\mathbf{O}_{F5}^{05}$ as and started the test with a reset version of the opptain network application. All applications were started simultaneously and were stopped when all *MeetingSummaries* were exchanged. Figure 5.6 shows in which round the devices connected and exchanged *MeetingSummary* lists. In round 20 all *MeetingSummaries* were exchanged.

5.8.2 Routing of Packets with MeetingSummaries

We used the setting proposed in Figure 3.1 to see if the routing protocol works. We used the devices $\mathbf{N}_{7B}^{08}$, $\mathbf{N}_{AE}^{1F}$, $\mathbf{O}_{3D}^{66}$, $\mathbf{P}_{4E}^{1A}$ and $\mathbf{Q}_{6E}^{11}$ as devices A to E. In the following scenario we only use the *MeetingSummaries* and no additional information for routing, also we look at the case that only devices with a non-zero delivery probability are used for forwarding in the Spray phase. We tested if a device knows when connected to another device if a packet should be transmitted or not. There are several cases to cover.

We start with the cases in which A has a *Packet* for another device.

- When A has a *Packet* for B and B is connected then B gets the *Packet* because B is the destination.
- When A has a *Packet* for C and B is connected then B gets the *Packet* because the stored *MeetingSummary* BCB covers a path to C.
- When A has a *Packet* for D and B is connected then B gets the *Packet* because the temporary *MeetingSummary* CDB covers a path to D.
- When A has a *Packet* for E and B is connected then B does not get the *Packet* because there is no *MeetingSummary* that indicates a path to E.

Now B has a *Packet* for another device. We do not show some cases that are already covered by analogous A cases.

- When B has a *Packet* for A and A is connected then A gets the *Packet* because A is the destination.
- When B has a *Packet* for C, D or E and A is connected then A does not get the *Packet* because there is no *MeetingSummary* that indicates a path to the destination.
- When B has a *Packet* for C, D or E and C is connected then C gets the *Packet* because C is the destination or can forward the *Packet*.
- All *Packets* from the A cases that have been transmitted to B for the purpose of forwarding are forwarded to C.

Packets that C wants to transmit to the other four devices will be routed to the destination because the four *MeetingSummaries* that C holds have enough information for finding a path to the receiver.

With this list all cases that can be accomplished by the routing based on exchange of *MeetingSummaries* are covered.

Expanding the Neighbourhood

The maximal amount of *MeetingSummaries* on a device when all devices are arranged in a line and each device has at most two neighbours is 4, the minimal amount is 1, both when the devices have

connected at least twice. When n devices are not connected in a line but all with each other, the amount of *MeetingSummaries* on each device will be

$$(n-1)+(n-1)(n-2) = n^2 - 2n + 1$$

after they have connected often enough.

Taking this into consideration we can determine a minimal and maximal number of *MeetingSummaries* stored on the device. A *MeetingSummary* currently has a size of about 150 bytes and in this scenario with 5 devices that are connected in a line we have at most a size of 600 bytes on device C that has collected four *MeetingSummaries*. In a scenario in which all devices connect to each other we can get a size of 1350 bytes on each device since there are nine *MeetingSummaries* saved on each device. Since today's devices have a large storage space we can increase the multi-hop-neighbourhood by saving and exchanging *MeetingSummaries* that cover more than a two-hop-neighbourhood. The storing scheme in the form of a database could be changed to compress the *MeetingSummaries* more and therefore save storage space.

Also the *MeetingSummaries* can be stored and exchanged based on their significance. That means that for example *MeetingSummaries* that indicate the frequent and periodic meeting of two devices can be exchanged and stored over more hops than a *MeetingSummary* that in fact indicates regularity but not frequency.

5.8.3 Routing with Attributes

We tested how *Attributes* can help with the routing. In the setting from before it was not possible for A to send a *Packet* to E because the *MeetingSummaries* do not indicate a path between A and E. When B, C, D, and E share the same Attribute and A has marked the *Packet* with this Attribute then A routes the *Packet* to B. In this example it would be sufficient if only B and E shared the same Attribute. When the Attribute is part of the *MeetingSummary* and is exchanged over the same amount of hops, only D and E had to share the same Attribute to give A the possibility to route a *Packet* to E.

Chapter Conclusion

In this chapter we presented the results of the different tests we conducted for the developed opptain network application as well as for the FileShipping application.

At first we tested the general ability of our test devices to open and connect to a hotspot. For all tested

smartphones and the Nexus 7 tablets the tests were successful. Also the tested Nexus 9 tablets were able to open hotspots without problems. However they showed inconsistent results with the regard to the ability to connect to access points in the surroundings.

In a further test the battery life and memory usage during a lasting activity of the opptain network application were tested. With regard to the battery life a expected linear descent was observed. The tests of memory usage showed the expected results, too, as there was only a slight or no decrease in the different tests.

In Section 5.3 the results of our transmission range tests are given. In inside and outside tests we tested the maximal range within a connection between devices could be built and data could be transmitted. The results showed great fluctuations for the different manufacturers of our test devices. The Nexus devices showed significantly higher ranges.

While implementing our opptain network application great differences in the behaviour regarding the time needed to launch a tethering hotspot occurred. Therefore we decided to examine this aspect in tests which as well showed great fluctuations also for devices which are the same model and were tested under the same conditions.

We successfully tested if the opptain network application can be used simultaneously with other functions of a smartphones like making or receiving a phone call or using other applications.

Also tests of the speed with which data can be exchanged between devices were conducted depending on the distances between the devices.

The important aspect of encryption and decryption speed was tested as well. We tested the speed for different devices and different sized files.

In Section 5.8 the results of extensive tests of the routing of the opptain network application are given. We successfully tested the creation of *Meetings* and *MeetingSummaries* and the exchange of *MeetingSummaries* within a three-hop-neighbourhood. In addition we tested the routing of *Packets* which functioned according to our routing protocol. The routing with attributes was tested as well as they represent a possibility for a more targeted routing.

Chapter 6

Conclusion and Future Work

In this chapter we will discuss to which extend the goals of this thesis have been fulfilled and which possibilities exist to extend this work. We summarise our achievements in Section 6.1 and look at possible future work in Section 6.2.

6.1 Conclusion

In this thesis we developed and implemented an Android application that uses Opportunistic Networks for transmitting data from one device to another regardless of being connected directly or over a multi-hop time-independent neighbourhood. We designed a new routing scheme based on several other routing protocols from current literature that have been tested mostly in a simulated environment rather than on real conditions. The new routing scheme takes all aspects of modern smartphones into consideration and is therefore a good choice for an application that uses opportunistic routing.

We designed the network application based on the demands for this work and fulfilled all of them. The network application runs in the background and does not disturb the user while running. The user still has the ability to run other applications, can make and receive phone calls, and can use the mobile data network, hence is connected to the Internet almost all the time. The only limitation, the absence of Wi-Fi connectivity while running the network application, can be lifted by easily pausing the network application.

The network application can be used by other applications for Opportunistic Network connectivity. We gave a proof of work by implementing another Android application that can be used to send and receive files. This application is equipped with functions that fulfil several further demands on the framework. We implemented both a symmetric and asymmetric encryption for files whereby the symmetric encryption makes use of a passphrase and is therefore useful for spontaneous file transmissions.

The asymmetric encryption uses public and private keys to encrypt files that are scheduled for a long range transmission and will be transmitted to different devices on the path to the destination. Also a principle of filesharing was developed that uses a distribution list and a file request system to retrieve files without the owner of the file interacting with the requesting user.

We tested the network and FileShipping applications together in several environments to ensure that they operate as desired. We came to the conclusion that the network works in all tested environments but we can not make clear statements about connection times and transmission speed because there are too many interference factors. We showed that Android works not deterministically in many ways and that its network functionality has many problems and bugs which makes it difficult to work with this features. Over the course of this thesis new problems occurred regularly regarding the behaviour of Android.

We see that Android is a user oriented operating system that can be used to fulfil all aspects of application development that is common nowadays but struggles with new functionality that is still uncommon like peer to peer connectivity. We had to use a major workaround by using a Server-Client structure to maintain a connection between two Android devices because Android does not offer functionality to fully enable and maintain an ad-hoc network. The internal possibilities to query network state information are limited and if they are given they often give false responses. We have shown that these workarounds work in general but conclude in a solution that has to struggle with aspects like connectivity issues and transmission speed loss.

6.2 Future Work

In this section we will show the possibilities which exist to enhance the functionalities of the applications implemented in this thesis. These possibilities comprise, among others, a longer battery life, higher safety and a more targeted data transmission.

While working on the thesis we encountered several possibilities to improve the applications. Those easy to realise were fulfilled while working on the implementation. Others were withheld for future implementation and will be presented in the following.

It would be a great improvement if a connection was held open until all applications were able to react to incoming packets. Up to now when to devices connect they exchange packets, disconnect and then deliver the received packets to the respective API applications. In the case that, for example, a device holds a certain file and then receives the related want-packet it is yet not able to directly send this packet back.

Another aspect of the connection between devices that could be looked at in future is that devices should be given the ability to scan for access points several times in a row and to thereby determine if a hotspot is moving or not. In this case the probability that the connection will last long and can be maintained as long as needed is higher. The device should then prefer to connect to this not moving hotspot.

In the small environment in which the applications were tested there was no need to implement an even more thoughtful and elaborate probability calculation for meeting summaries as a packet normally reaches its destination after only a few transmissions. Considering that, compromises regarding the meeting summaries were made. The meeting summaries do not cover the full identity table the routing protocol HiBOp proposes. Therefore routing probabilities cannot be judged and processed to the full extent. The complete implementation of this would be a great improvement of the routing as it would allow to calculate significant probabilities. A far more targeted routing is then possible.

In the same context it is to be mentioned that it was not possible to fully evaluate the location functionality because Android uses mobile network connections as well as GPS to do location updates. A further limitation in using the location functionality is that in buildings GPS information is not available. Despite these restrictions the location functionality offers an important possibility to improve the routing. For example by collecting information about the locations an area in which connections between devices take place frequently can be determined. In a use case this could be, for example, a company building or a university.

Rooting the device could enable ad-hoc-networking but takes away possibility of big network since no device could use the application then. But it is worth to take a look at.

Another aspect that could be worked on is the protection against spam which up to now is not realised. Since devices always accept all Packets they are offered, they could fill up their device up to a point where no storage space is available.

As always when an user oriented application is developed there are countless possibilities to alter and extend the GUI of the application.

Bibliography

[BBL05] B. Burns, O. Brock, and B.N. Levine. Mv routing and capacity building in disruption tolerant networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 1, pages 398–408 vol. 1, March 2005.

[BCJP07] Chiara Boldrini, M. Conti, Jacopo Jacopini, and A. Passarella. Hibop: a history based routing protocol for opportunistic networks. In *World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a*, pages 1–12, June 2007.

[CDMP13] Marco Conti, Franca Delmastro, Giovanni Minutiello, and Roberta Paris. Experimenting opportunistic networks with wifi direct. In *Wireless Days*, pages 1–6. IEEE, 2013.

[Cre15] Creative Commons. Creative commons — attribution 2.5 generic — cc by 2.5. `http://creativecommons.org/licenses/by/2.5/`, March 2015. Last checked: March 15th 2015.

[Dwo01] Morris Dworkin. Sp 800-38a. recommendation for block cipher modes of operation: Methods and techniques. Technical report, Gaithersburg, MD, United States, 2001. Last checked: March 15th 2015.

[Ecl15] Eclipse Foundation. Eclipse public license v1.0. `http://www.eclipse.org/legal/epl-v10.html`, March 2015. Last checked: March 15th 2015.

[EMST78] W.F. Ehrsam, C.H.W. Meyer, J.L. Smith, and W.L. Tuchman. Message verification and transmission error detection by block chaining, February 1978. Last checked: March 15th 2015.

[Fal03] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for*

Computer Communications, SIGCOMM '03, pages 27–34, New York, NY, USA, 2003. ACM.

[Fre15] Free Software Foundation, Inc. Gnu lesser general public license (lgpl), version 2.1. http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html, March 2015. Last checked: March 15th 2015.

[Goo15a] Google Inc. Activities | android developers. http://developer.android.com/guide/components/activities.html, March 2015. Last checked: March 15th 2015.

[Goo15b] Google Inc. Android security vulnerability. https://bitcoin.org/en/alert/2013-08-11-android, March 2015. Last checked: March 15th 2015.

[Goo15c] Google Inc. Application fundamentals | android developers. http://developer.android.com/guide/components/fundamentals.html, March 2015. Last checked: March 15th 2015.

[Goo15d] Google Inc. Dashboards | android developers. https://developer.android.com/about/dashboards/, March 2015. Last checked: March 15th 2015.

[Goo15e] Google Inc. Device compatibility | android developers. http://developer.android.com/guide/practices/compatibility.html, March 2015. Last checked: March 15th 2015.

[Goo15f] Google Inc. Licenses | android developers. https://source.android.com/source/licenses.html, March 2015. Last checked: March 15th 2015.

[Goo15g] Google Inc. Services | android developers. http://developer.android.com/guide/components/services.html, March 2015. Last checked: March 15th 2015.

[Goo15h] Google Inc. System permissions | android developers. http://developer.android.com/guide/topics/security/permissions.html, March 2015. Last checked: March 15th 2015.

[HLT08] Chung-Ming Huang, Kun-chan Lan, and Chang-Zhou Tsai. A survey of opportunistic networks. In *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on*, pages 1672–1677, March 2008.

[IDC15] IDC. Android and ios squeeze the competition. `http://www.idc.com/getdoc.jsp?containerId=prUS25450615`, March 2015. Last checked: March 15th 2015.

[JK03] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003. Last checked: March 15th 2015.

[Kal00] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000. Last checked: March 15th 2015.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Last checked: March 15th 2015.

[Kra12] Tobias Krauthoff. Eine android app für alltägliche studieninformationen am beispiel der heinrich-heine-universität düsseldorf. Master's thesis, Heinrich-Heine-Universität Düsseldorf, August 2012.

[LDS04] Anders Lindgren, Avri Doria, and Olov Schelén. Probabilistic routing in intermittently connected networks. In Petre Dini, Pascal Lorenz, and JoséNeuman de Souza, editors, *Service Assurance with Partial and Intermittent Resources*, volume 3126 of *Lecture Notes in Computer Science*, pages 239–254. Springer Berlin Heidelberg, 2004.

[Mel15] Mel Khamlichi. Why is android built on linux kernel? `http://www.unixmen.com/why-is-android-built-on-linux-kernel/`, March 2015. Last checked: March 15th 2015.

[MSW12] Johannes Morgenroth, Sebastian Schildt, and Lars Wolf. A bundle protocol implementation for android devices. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 443–446, New York, NY, USA, 2012. ACM.

[Ope15] Open Source Initiative. The mit license (mit). `http://opensource.org/licenses/MIT`, March 2015. Last checked: March 15th 2015.

[oST01] US National Institute of Standards and Technology. Federal information processing standards publication (FIPS 197). Advanced Encryption Standard (AES). `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, November 2001. Last checked: March 15th 2015.

[SB07] K. Scott and S. Burleigh. Bundle Protocol Specification. RFC 5050 (Experimental), November 2007.

[SPR05] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. Spray and wait: An efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Delay-tolerant Networking*, WDTN '05, pages 252–259, New York, NY, USA, 2005. ACM.

[SPR08] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. Efficient routing in intermittently connected mobile networks: The multiple-copy case. *IEEE/ACM Trans. Netw.*, 16(1):77–90, February 2008.

[The15] The Apache Software Foundation. Apache license, version 2.0. `http://www.apache.org/licenses/LICENSE-2.0`, March 2015. Last checked: March 15th 2015.

[TKHL15] Sacha Trifunovic, Maciej Kurant, Karin Anna Hummel, and Franck Legendre. Wlan-opp: Ad-hoc-less opportunistic networking on smartphones. *Ad Hoc Networks*, 25, Part B(0):346 – 358, 2015. New Research Challenges in Mobile, Opportunistic and Delay-Tolerant Networks Energy-Aware Data Centers: Architecture, Infrastructure, and Communication.

[VB+00] Amin Vahdat, David Becker, et al. Epidemic routing for partially connected ad hoc networks. Technical report, Technical Report CS-200006, Duke University, April 2000.

[WDAV13] I. Woungang, S.K. Dhurandher, A. Anpalagan, and A.V. Vasilakos. *Routing in Opportunistic Networks*. SpringerLink : Bücher. Springer, 2013.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 16. März 2015 Andre Ippisch

Diese CD enthält:

- eine *pdf*-Version der vorliegenden Masterarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Masterarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Masterarbeit erstellten Android-Software
- die zur Auswertung verwendeten Logging-Dateien
- die Websites der verwendeten Internetquellen