Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Daniel Lee

Robust wireless data communication for Android Smartphones using Fountain Codes

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Lee

Robust wireless data communication for Android Smartphones using Fountain Codes

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. habil. Dirk Westhoff
Zweitgutachter : Prof. Dr. -Ing. Martin Hübner

Abgegeben am 2. November 2011

Daniel Lee

Thema der Bachelorarbeit

Robust wireless data communication for Android Smartphones using Fountain Codes

Stichworte

Android, Smartphone, Fountain Codes, rauschbehaftetes Medium, kabellose Kommunikation, mehrfache Empfänger

Kurzzusammenfassung

Android Smartphones werden stetig beliebter. Die Kommunikation zwischen diesen Smartphones läuft ausschließlich kabellos. Und kabellose Verbindungen können leicht durch externe Störquellen behindert werden. Es wäre also wünschenswert, eine Möglichkeit zu haben, effizient kommunizieren zu können, auch wenn mal das eine oder andere Paket verloren geht. Das Ziel dieser Bachelorarbeit ist es, eine Applikation für Android Smartphones zu schreiben, die mit Hilfe der Fountain Codes Daten effizient an mehrere Empfänger gleichzeitig verschicken kann. Fountain Codes bieten eine spezielle Kodierungsart, die es ermöglicht, Packetverluste zu kompensieren und sind daher ideal für kabellose Verbindungen.

Daniel Lee

Title of the paper

Robust wireless data communication for Android Smartphones using Fountain Codes

Keywords

Android, Smartphone, Fountain Codes, unreliable medium, wireless communication, multiple receivers

Abstract

Android Smartphones keep getting more popular. The communication between these smartphones is exclusively done wirelessly. Wireless connections can easily be disturbed by external sources. That is why it would be beneficial to have a possibility to efficiently communicate with other smartphones even though some packets might get lost. The goal of this bachelor thesis is to implement an application for Android Smartphones, which distributes files to multiple recipients using the Fountain Codes. Fountain Codes offer a special coding scheme which makes it possible to compensate packet losses. Therefore, they are ideal for wireless communication.

Contents

List of Tables 7

List of Figures 8

1 Introduction 10
1.1 Motivation . . . 10
1.2 Thesis Overview . . . 11
1.3 Organizing of Report . . . 11

2 Fountain Codes 13
2.1 What are Fountain Codes . . . 13
2.1.1 Principle of operation . . . 14
2.2 LT Codes . . . 15
2.2.1 LT encoding process . . . 15
2.2.2 LT decoding process . . . 16
2.3 Use of Fountain Codes . . . 17
2.3.1 Wireless sensor networks (WSN) . . . 18
2.3.2 Automobile Industrie . . . 20

3 Analysis 24
3.1 Scenarios . . . 24
3.1.1 User wants to distribute data . . . 24
3.1.2 User wants to receive data . . . 27
3.1.3 Scenarios for security measures . . . 28
3.2 Requirements . . . 31
3.2.1 Requirements for sender . . . 31
3.2.2 Requirements for receiver . . . 32
3.2.3 Requirements for security . . . 33
3.3 Summary . . . 33

4 Android 34
4.1 What is Android? . . . 34
4.2 Android System Architecture . . . 35

4.2.1 Android Security . . . 36
4.3 Android Programming Fundamentals . . . 37
4.3.1 UI programming . . . 38

5 Application Design **39**
5.1 Wireless Local Area Network . . . 39
5.2 Architecture Overview . . . 40
5.3 Security Design . . . 42
5.3.1 Data integrity . . . 43
5.3.2 Establishing a session key . . . 47
5.3.3 Data encryption . . . 49
5.4 Sender Design . . . 49
5.4.1 Sender . . . 50
5.4.2 Fountain Encoder . . . 50
5.5 Receiver Design . . . 54
5.5.1 Receiver . . . 55
5.5.2 Information Packet Listener . . . 55
5.5.3 Fountain Decoder . . . 56
5.6 Summary . . . 58

6 Application Implementation **59**
6.1 Sender . . . 59
6.1.1 SenderTab . . . 59
6.1.2 FountainSender . . . 60
6.1.3 Other classes . . . 61
6.2 Receiver . . . 62
6.2.1 ReceiverTab . . . 63
6.2.2 DataInformationListener . . . 64
6.2.3 FountainReceiver . . . 64

7 Test and Evaluation **66**
7.1 Test . . . 66
7.1.1 Sending files . . . 66
7.1.2 Receiving files . . . 71
7.2 Evaluation . . . 74
7.2.1 Fountain Codes . . . 74
7.2.2 Application performance . . . 80

8 Summary and Future Work **81**
8.1 Summary . . . 81
8.2 Future Work . . . 82

Bibliography **84**

List of Tables

7.1 Initially implemented degree distribution. 75
7.2 Time listing of receiver. 77
7.3 Listing of encoded packets needed to encode the file. File is split into 4257 data blocks. 77
7.4 Implemented degree distribution 77
7.5 Listing with new degree distribution from table 7.4. 78
7.6 Listing with the new degree distribution from table 7.4 and the new selection scheme from figure 7.12. 79

List of Figures

2.1 Sender generates and broadcasts code word. Receiver gets code word and decodes. 14
2.2 LT Codes decoding process [1]. 17
2.3 Possible realization of a wireless sensor network [2]. 18
2.4 One sensor node receiving a code image from two senders. 20
2.5 Audi A8 navigation system with integrated Google Earth [3]. 21
2.6 Different firmwares deployed to cars using Fountain Codes. 22
2.7 Files are being sent from one smartphone to multiple devices simultaneously. 23

3.1 Sender broadcasting file to everybody in the ad-hoc network. 25
3.2 Sender multicasting file to members of his project team. 27
3.3 Adversary creating own data packet and sending at higher rate. 29

4.1 Android system architecture [4]. 35

5.1 First possible application architecture. 40
5.2 Sequence diagram of architecture to implement. 42
5.3 5 steps to generate the key pair in the Digital Signature Algorithm [5]. 44
5.4 Sender signs message and receiver verifies the signature [5]. 45
5.5 User interface for application. 51
5.6 Flowchart showing procedure of the Fountain Encoder. 53
5.7 Table showing found degree distributions and their results [6]. 54
5.8 User interfaces and dialogs for receiver. 55
5.9 Flowchart of Information Packet Listener. 56
5.10 Flowchart of Fountain Decoder as a whole. 56
5.11 Detailed flowchart of mode *Process new packet*. 57
5.12 Detailed flowchart of mode *New plaintext added*. 58

6.1 UML diagram of the sender. 60
6.2 UML diagram of the receiver. 62

7.1 Sender tab configuration UI. 67
7.2 Screenshot of the browse and progress dialog. 68
7.3 Screenshot of abort dialog and receive dialog from MacBook receiver. 69

7.4 Toast messages informing the user that information is missing. 70
7.5 MacBook receives encrypted file. 71
7.6 Initial ReceiverTab user interface and UI when receiver is listening for data information. 72
7.7 Data information dialogs for supported file types. 73
7.8 Receive progress dialog, abort dialog, and changed UI after reception of files. 74
7.9 Individual dialogs shown depending on file type. 75
7.10 Dialog promting user for password. 76
7.11 Initial scheme to select random data blocks. 78
7.12 Improved scheme to select random data blocks. 79

1 Introduction

1.1 Motivation

Nowadays almost one out of four people in Germany own a smartphone and the amount of owners keep increasing rapidly [7]. A lot of those smartphones run the Android Operating System. Data communication on smartphones is mainly achieved wirelessly. Communications include files being shared via Bluetooth, Wi-Fi, or UMTS. Furthermore, software updates that are downloaded to increase the usability of the smartphone, make it run more robust or to add new features. Especially the communication over Wi-Fi is very popular as it is very fast. Another important reason is that Wi-Fi is mostly free of charge. Be it at home or one of the many free hot spots offered in the cites.

Often it is desired to send a file to multiple receivers simultaneously. Currently, inspired by services like Facebook, a lot of people want to share their private data like pictures, music tracks, or videos with the public. Even with people they do not know. Another motivation to send data to multiple recipients could be a software update. Users that have already downloaded the update could send the code image to smartphone owners who did not yet have the opportunity do download it. In the course of distributing the data, it is important that almost every recipient does get the complete data correctly. However, establishing a reliable connection-oriented connection to let us say hundreds of recipients is a heavy administrative workload for the sending side. Apart from the overhead which would occur since the recipients send feedback about successfully received and lost packets. That is why it is more efficient to send the data by using the UDP protocol and thus broadcast the data. But there is one problem when using UDP. The datagrams may or may not reach the target. And particularly using an unreliable medium such as Wi-Fi could lead to a recurring data loss. Of course, Wi-Fi itself is not unreliable. Wi-Fi uses protocols of the network stack which offer mechanisms that assure reception of sent packets with high probability. But due to interfering sources in big crowded cities, there is a higher probability that more packets are lost along the way than is usually the case. That is why Wi-Fi will be referred to as an unreliable medium in this bachelor thesis. So to ensure that every recipient receives the complete file, every datagram must be sent redundantly. Disadvantageous however is that data packets, which have already been received and sent again by the sender, do not feature any new useful information. Thus, an unnecessary transmission was performed. Therefore, it would

be desirable to reduce unnecessary transmissions as far as possible by encoding different data parts in such a way that the recipients are able to extract information relevant to them. Of course given that the information was encoded into the datagram. Such an encoding scheme is offered by the *Fountain Codes* (Description of Fountain Codes in 2.1).

1.2 Thesis Overview

This bachelor thesis is focused on designing and implementing an application for the Android platform. The goal of the application is to send and receive files like pictures, music, or code images for software updates over a wireless medium, such as Wi-Fi, using the Fountain Codes. The sender can choose to send the file to every recipient in the same network or to a selected subset of receivers. If the sender chooses to target only a subset of clients in the network, he must encrypt the file accordingly. Only the targeted recipients should then be able to successfully receive the file. Upon reception of a datagram from the sender, the recipient can choose to accept or decline the file. In case of an encrypted file, the receiver must own the secret key to decrypt the file. As the application will offer the distribution of code images which install new applications or update installed applications on the Android Smartphone, the application must offer the option to install those code images. Security aspects such as key agreement, data integrity, and encryption as already mentioned, will also be part of the application.

In bachelor thesis [8] the author also focused on Fountain Codes. But his main focus was on the security aspect of Fountain Codes. This bachelor thesis is about the realization of efficient Fountain Codes for file distribution. An important part of the bachelor thesis is to find a good Degree Distribution so that the overhead of needed packets is kept as low as possible.

1.3 Organizing of Report

The remainder of the bachelor thesis will be organized as follows:
Chapter 2 gives a detailed description of the Fountain Codes and their principle of work. Also LT Codes are explained which are one possible realization of Fountain Codes. At the end of chapter 2 there will be some examples of fields of application where Fountain Codes could be useful.
Chapter 3 is about the analysis of the application which is going to be implemented in this bachelor thesis. Scenarios for use of the application will be given. Based on those scenarios, requirements will be extracted to give an overview of the functionality of the application.

Chapter 4 is about the Android Operating System. The chapter gives an insight into the Android system and the fundamentals of Android application development.
Chapter 5 is about designing the application. The requirements extracted in chapter 3 will be taken into account to see how the application can be designed and structured.
Chapter 6 is about the implementation. The important classes and their function will be explained. Furthermore, their main methods will be listed and described.
Chapter 7 will focus on testing and evaluating the implemented application. The evaluation will focus on the performance of the Fountain Codes and if needed their adaption to work more efficiently.
Chapter 8 will summarize the important aspects of the bachelor thesis and point out topics for future works.

2 Fountain Codes

2.1 What are Fountain Codes

Fountain Codes are a type of erasure codes. They are also known as rateless erasure codes. The term rateless refers to the fact that Fountain Codes do not have a fixed code rate. Fountain Codes are often used when communicating over an unreliable medium such as wireless communication channels. When using Fountain Codes, the sender is able to generate potentially unlimited amounts of code words from any data he wants to distribute. Those code words are broadcasted to any receiver listening. The receiver can obtain the complete source data by decoding any subset of encoded packets equal or slightly larger than the number of source packets. Successfully decoding the source data with only slightly more or even exactly the number of source packets highly depends on how the sender encodes those code words. Choosing an inefficient encoding scheme can lead to unwanted effects. The result is that there is the possibility that the receiver needs almost twice the amount of source packets to retrieve the complete data. In worst cases even more than that. Therefore, the decoder receives a lot of packets that are useless because the information encoded into the packet have already been received and thus no new information is extracted and computing the packet was needless.

Fountain Codes are advantageous due to the fact that they belong to the class of *Forward Error Correction* codes. In contrast to *Backward Error Correction*, where receivers send a request for retransmission to the sender if they receive a corrupt packet or no packet at all, since the data is shared over an unreliable medium, *Forward Error Correction* codes work differently. Instead of listening for retransmission requests and sending packets accordingly, the sender just broadcasts different random linear combinations of the split source data. The receivers can individually extract the missing data from that packet, provided that the missing data chunk was XORed into the encoded packet (Figure 2.1). One detail that should be pointed out is the fact that the receivers could be missing different packets and still be able to extract useful information from one and the same encoded packet. In conclusion, the error correction is done on the receiver's side. In case the missing data is not part of the code word, the receiver tries the next packet. If the communication between sender and receiver was based on some sort of acknowledgement packets (NACK-Packets), informing the sender of missed packets, the sender would be flooded with such information and the

overhead of coping with those information would heavily slow down the communication. Even normal broadcasting without any feedback from the receivers would be less efficient than using Fountain Codes as, due to the unreliable medium, the sender would have to consider data loss and send the source data at least twice to ensure complete reception of the data. With Fountain Codes, and given an efficient encoding scheme, the sender would only have to broadcast about one and a half times the data and actually be sure that with high probability the complete data is assembled at the receiver's side.

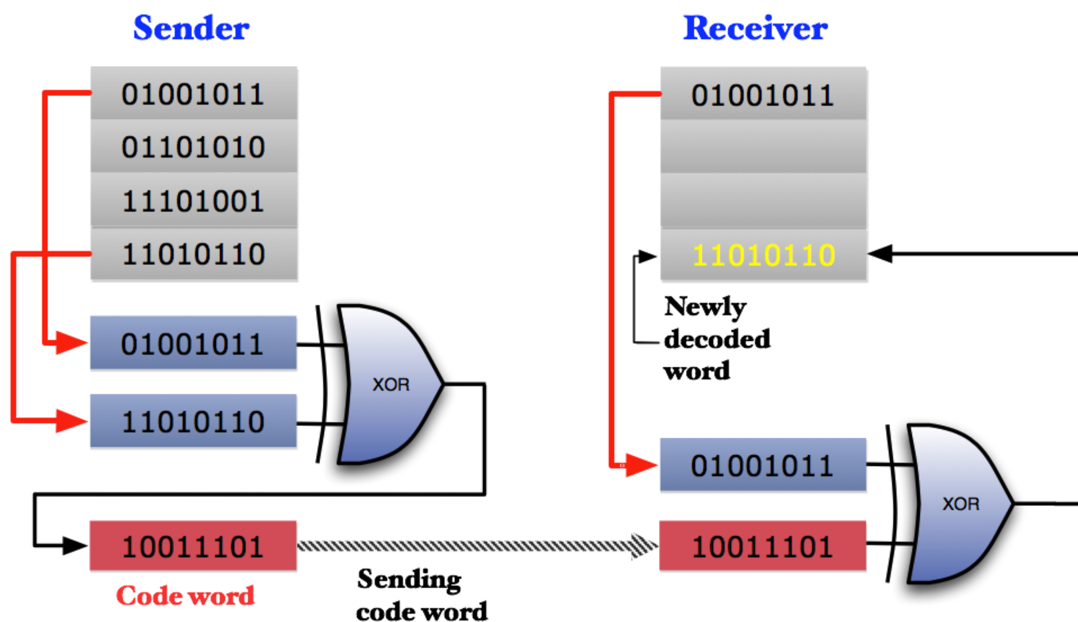


Figure 2.1: Sender generates and broadcasts code word. Receiver gets code word and decodes.

2.1.1 Principle of operation

Encoding process: The code words are generated by splitting the original source data into equal data chunks. Uniformly at random chosen data chunks are then XORed to encode the data. Along with the encoded data, a coefficient vector is generated and sent. The coefficient vector holds the information about which data chunks where XORed into the code word so that the receiver knows which already received data can be used to decode the encoded packet. A detailed description of an encoding process will be given in 2.2.1

Decoding process: The receiver decodes the data by solving the linear equation system $t = A * s$ for s [9]. The vector t contains the received encoded packet. A is a *m x m* matrix where the *m*s are the coefficient vectors which already have been received successfully. The time complexity of the decoding algorithm is $O(k^3)$. The variable k contains the amount of equal data chunks that the source data is split into.

The procedures described above are the basic principles of encoding and decoding when using Fountain Codes. However, as the decoding algorithm is not very efficient, other algorithms have been developed and should be used when working with Fountain Codes.

There are different practical realizations of Fountains Codes such as LT Codes [10] or Raptor Codes. To develop the application in this bachelor thesis, the coding principles of LT Codes will be used to implement the Fountain Codes.

2.2 LT Codes

LT Codes are the first practical realization of Fountain Codes. They were invented by Michael Luby. As is the definition of Fountain Codes, LT Codes are rateless and the number of code words that can be generated are limitless. The main difference to basic Fountain Codes is the way, data is decoded by the receiver (See 2.2.2). While solving the linear equation system $t = A * s$ (Equation explained in Decoding process of section 2.1.1) to decode the data has a time complexity of $O(k^3)$, the decoding algorithm of LT Codes just has a time complexity of $O(k * ln(k/\delta))$ when the receiver is successfully recovering the data with a probability of $1 - \delta$ from $k + O(\sqrt{k} * ln^2(k/\delta))$ received packets. k is the amount of data blocks the data is split into and δ is a small value that determines the probability of failure of the decoding process.

2.2.1 LT encoding process

The encoding process of LT Codes does not differ from the basic encoding scheme of Fountain Codes. First the source data has to be divided into *k* equal parts. In applications using Fountain Codes, *k* is often chosen to be close to the size of a datagram payload to optimally exploit the network protocol and avoid unnecessary traffic. After the source data is divided, it takes the sender three steps to generate the encoded packet.

1. A degree *d* is chosen from a degree distribution *p(d)*. The degree *d* defines how many random data chunks are XORed together to encode the packet. *p(d)* is the probability that an encoded packet has the degree *d*. Designing and finding a good degree distribution is very important regarding the efficiency of the LT Codes and all

other rateless erasure codes. Working with a poorly designed degree distribution can lead to slow or incomplete data distribution as the decoder has to receive almost twice or even more encoded packets to retrieve the complete source data.

2. Out of the *k* data packets *d* parts are uniformly chosen at random.

3. The chosen packets are then XORed together to form the encoded packet.

As mentioned earlier a coefficient vector is needed to make the decoding possible. There are a lot of different possibilities to realize the coefficient vector. Two possibilities are:

1. The coefficient vector is directly communicated to the decoder with every encoded packet, i.e., the coefficient vector is sent along with the encoded packet.

2. The coefficient vector is separately computed on the encoder and decoder side. The computation could be based on the indices of each data chunk encoded into the packet being calculated with a formula. For example, the encoder chooses uniformly at random two integers $a, b \in (1, \ldots, k-1)$ and sends the two integers along with the encoded symbol. The *i*-th index then, is the ($a * i + b$ mod k)-th data part out of the *k* source symbols [11].

Having gone through all necessary operations to complete the encoded symbol, the sender is now ready to broadcast the datagram. The encoding operations can be repeated on the fly and thus the sender can generate potentially unlimited encoded symbols containing different degrees and coefficients.

2.2.2 LT decoding process

The LT Codes decoding process (Figure 2.2) is different from basic Fountain Codes. The decoder does not solve the linear equation system $t = A * s$ but works with two buffers *A* and *B* containing packets which are not completely decoded and packets that have been successfully decoded, respectively. The decoding process described here presumes the coefficient vector being sent along with the encoded symbol. When the decoder receives an encoded packet (*X, C*) he first extracts the code symbol *X* and the coefficient vector *C* and tries to decode the code symbol with the help of buffer *B*. The decoder iterates the coefficient vector and crosschecks every coefficient against buffer *B*. If *B* contains a plaintext p_i at the index corresponding to the coefficient, that plaintext its XORed into the code symbol and the degree is decremented. Afterwards the degree of the packet is checked. If the degree $D(C) > 1$ then the code symbol is not yet fully decoded and the pair *X, C* is stored in buffer *A* for a subsequent try. If the condition is not met and $D(C) = 1$, meaning the codes symbol is a plaintext, p_i is stored in buffer *B*, provided that the *i*-th plaintext was not already stored. After successfully decoding one packet and storing the plainetext p_i in buffer *B*, another

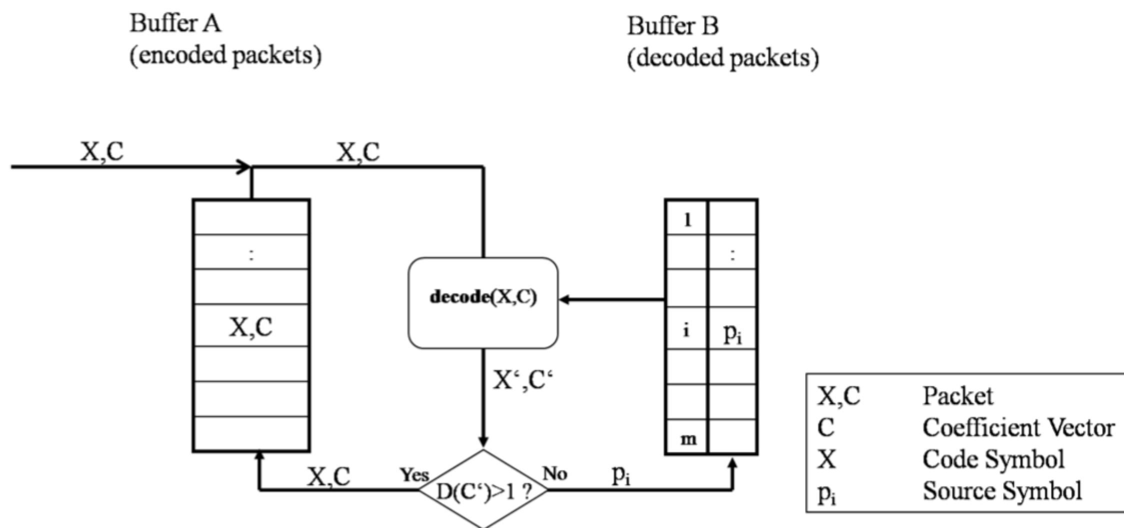


Figure 2.2: LT Codes decoding process [1].

attempt to decode code symbols in buffer *A* is started and repeated until no new plaintext is retrieved and stored in *B*. By the time no new code symbol *X* is able to be decoded, another Packet (*X, C*) is received and processed.

2.3 Use of Fountain Codes

Fountain Codes can be used in many areas. Almost any application field, involving communication over an unreliable medium, can benefit from the characteristics of Fountain Codes. Even if packets are lost to interference on the channel, only slightly more packets are necessary for a successful distribution to every recipient. Furthermore, the sender does not need to keep track of the clients and their status of missing packets. Thus, the use of Fountain Codes can also be advantageous in cases where sending sources change during distribution. There are two important main preconditions that must be met for Fountain Codes to work and be useful to work with. First of all Fountain Codes are only useful in cases where a lot of data is distributed. It makes no sense to use Fountain Codes when a data of 10 Bytes is sent. That data can be sent in one UDP datagram and thus there is no need to encode anything. Secondly the data to distribute must be completely available at the sender before encoding can start. A reason for that condition is, for example, the degree distribution. The degree distribution is crucial for the performance of Fountain Codes and must also be chosen before the encoding starts. If data blocks are encoded during aggregation of data and the

degree is of higher value than the current data size, the Fountain Coding process will fail with an exception.

2.3.1 Wireless sensor networks (WSN)

A wireless sensor network (Figure 2.3) consists of sensor nodes which are deployed around a large area. They monitor the surrounding conditions and collect required data. Sensor nodes

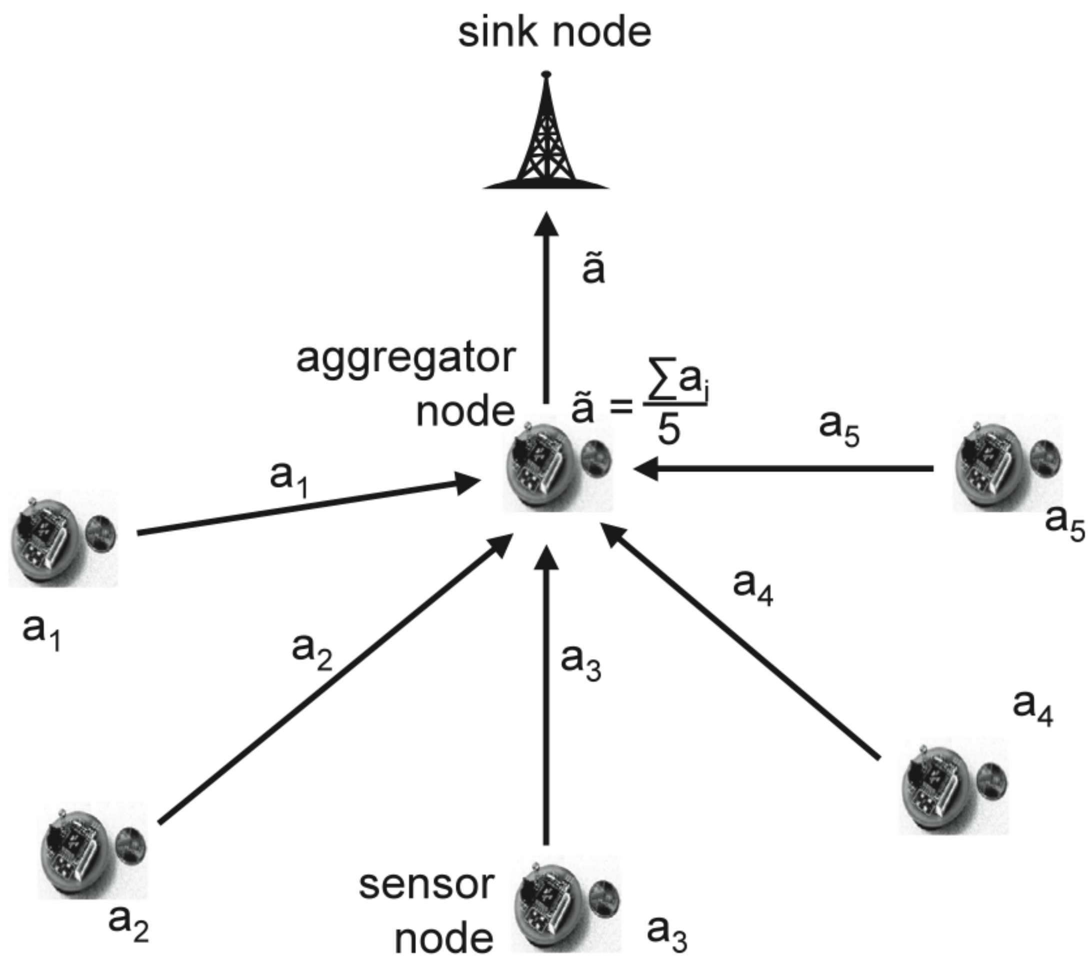


Figure 2.3: Possible realization of a wireless sensor network [2].

often monitor data like temperature, pressure, sound, and other physical or environmental conditions of the area. Sensor nodes have a limited energy source and computing power. They do not analyze the sensed data. Each node solely gathers information and sends his

data to a central aggregator node. That node sends the aggregated data to a central location, called the *sink*. The sink collects the data from all aggregator nodes deployed and analysis the data. The sink can also initiate an update process of the sensor nodes by sending a code update image.
Not every WSN has to have an aggregator node. There are different realizations of WSN. This is just on possible realization.

Use in WSN

Fountain Codes can be used in wireless sensor networks since they are beneficial to almost every aspect of wireless communication done in a WSN. Let us look at the example of an code update process. In this WSN there is no aggregator node. Sensor Nodes communicate directly with the sink.

In wireless sensor networks occasionally there is the need to reprogram the sensor nodes. A code image is loaded into the sensor nodes and they can execute the code image which brings them up to date. But wireless sensor networks often consist of hundreds or thousands of sensor nodes. Loading the code image into all the nodes by hand takes a very long time and is therefore out of the question. Fortunately, the sensor nodes are also able to receive data.

The code update process is initiated by the sink. The sink broadcasts the code image to the sensor nodes wirelessly. But the sink cannot reach every node deployed due to the fact that most of the sensor nodes are spread very widely in the area. The extent of radio waves are limited and if the distance between the sink and a sensor node is too big, the code image never reaches the node. That is why those sensor nodes that do get the code image have to forward the image to the nodes that are in their broadcasting scope. However, doing so will drain their limited power sources. But unfortunately there is no other way to avoid that the sensor nodes are burdened with additional work. The only option is to try to minimize the extra task. By using Fountain Codes for code image distribution, additional processing time can be reduced. When broadcasting in the classical way, the sensor node has to send the code image more than once to assure that the surrounding sensor nodes receive the complete data. But as described in 2.1, when using Fountain Codes, the sender only has to send the code image slightly more than once. In optimal situations the sender even needs to send less than the total amount of data packets. If a sensor node receives the code image from say two other sensor nodes, the two nodes must only send slightly more than half of the total amount, due to the Fountain characteristic (Figure 2.4).

If two senders broadcasted the code image in the classical way, they would have to agree on who will send the first half of data and which one the second half. In that case, unnecessary overhead would be the result.

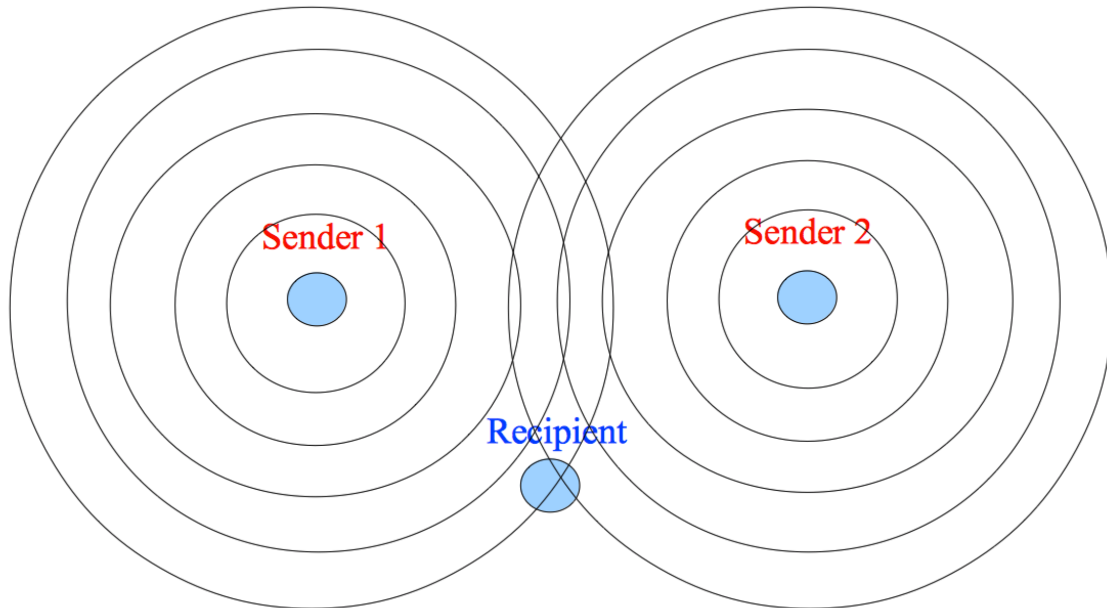


Figure 2.4: One sensor node receiving a code image from two senders.

2.3.2 Automobile Industrie

In recent years automobiles have changed. Automobiles used to be just simple vehicles that transported people from one location to another. Now however, as loads of new technologies in IT have emerged, they find their way into the automobile industry. Probably every higher class limousine from any automobile manufacturer has some sort of computing device integrated into the automobile. Be it just a navigation system or a multimedia system that can play music or even play videos. There are even automobiles that are equipped with built in game consoles. Figure 2.5 for example shows a navigation system of an Audi A8 with integrated Google Earth service.

Software update for moving automobiles

With all the new services integrated into automobiles, there comes a problem. Those services must be kept up to date. Navigation systems manufacturers may release updated maps or the automobile manufacturer wants to add new features to their multimedia systems that come with the car. As mentioned above and shown in Figure 2.5, there are services provided by external corporations like Google. And Google keeps to enhance its Google Earth service continuously. Forcing the customer to check in a garage every few months to update the system is not very service orientated. Many customers would neglect the monthly check

Figure 2.5: Audi A8 navigation system with integrated Google Earth [3].

and drive around with out of date system version. Such service would lead to bad publicity for the automobile manufacturer. So the manufacturer could choose to update the system wirelessly.

Let us assume that the automobile is equipped with some sort of UMTS receiver. System updates are received over that receiver. To deploy the update, the automobile manufacturer has to set up several locations from where the system update is broadcasted via UMTS. Each location covers a limited area due to the limited wave lengths. Since it is the main characteristic of automobiles, they keep moving from one location to another. Therefore, with high probability the automobile will switch between different update locations. If classical broadcasting was used and say the first one hundred out of one thousand packets were received and the automobile switched into another update location area, some negative effects could arise. Assuming the new update location just started broadcasting the system update, the receiver would receive the first one hundred packets redundantly without gaining any new information. If the receiver is really unlucky he switches yet again into another area with a new broadcaster that started the update again. One can see that a different, better distribution solution should be used. And again Fountain Codes are a very efficient solution. Each up-

date broadcasting location is initialized with the same degree distribution and started. They keep encoding packets on the fly and sending them to every client in the network. With very high probability the automobile will receive a data packet that has information useful to him, because the data parts are chosen at random. Now it does not matter that update broadcasting locations are changed during travel. With slightly more packets than the total amount of data blocks, the automobile's multimedia system will be able to execute the update and afterwards be up to date.

Firmware update for sold cars

Another scenario for the use in the automobile field is the firmware update of cars that are for sale. Nowadays cars are not equipped with different engines anymore which have different performance specification like horse power (HP). It is more economical for the automobile manufacturer to produce only one engine of a specific type, like diesel engines, and control the HP and other characteristics with different firmwares. Meaning that one car could be programmed with a firmware which sets the performance to 75 HP and another car with the same engine is programmed to have 110 HP.

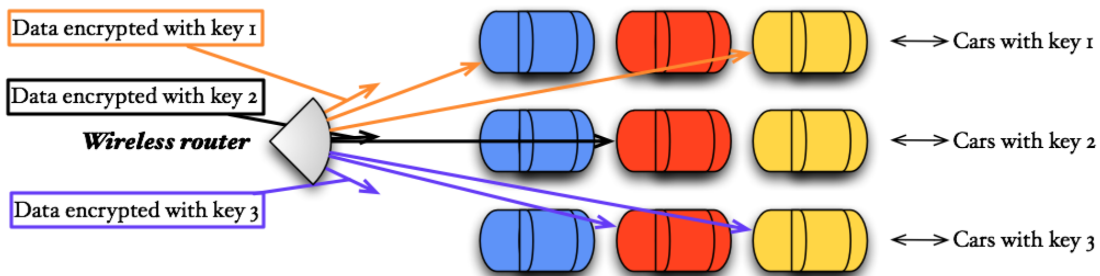


Figure 2.6: Different firmwares deployed to cars using Fountain Codes.

Let us take a car dealership that has a lot of cars of the same type parking on the sales area. All cars are connected to the same WLAN. Using Fountain Codes, different firmwares could be programmed into different cars simultaneously from a central location. To achieve that cars get the targeted firmware they could be preconfigure with a key. If customers buy a car with a specific configuration, that configuration can be deployed by encrypting the encoded packets of the firmware with the targeted key and therefore only the cars with that key are programmed with the firmware (Figure 2.6).

File distribution at home

The two use examples described above are very special and therefore may not be that interesting to some readers. That is why a last short example shall show when Fountain Codes could be used.
The majority of households nowadays have a flatscreen tv, a desktop computer, and maybe also a notebook. And as was already mentioned in the introduction section, the majority of people owns a smartphone. The cameras of the smartphones today keep getting better and better. And a lot of users start taking many pictures or record videos with the smartphone.
At home the user maybe wants to copy the files to the desktop computer, the tv, and the notebook to have multiple copies and if needed, to be able to show them from anywhere. Connecting the smartphone to every device is very annoying. But if the smartphone was able to distribute the files to all of those devices simultaneously and on top of that efficiently no matter how many interfering sources are nearby, the user can save a lot of time. He just has to open an application on his smartphone that offers the service and send it to every device in his home. Figure 2.7 depicts the process.

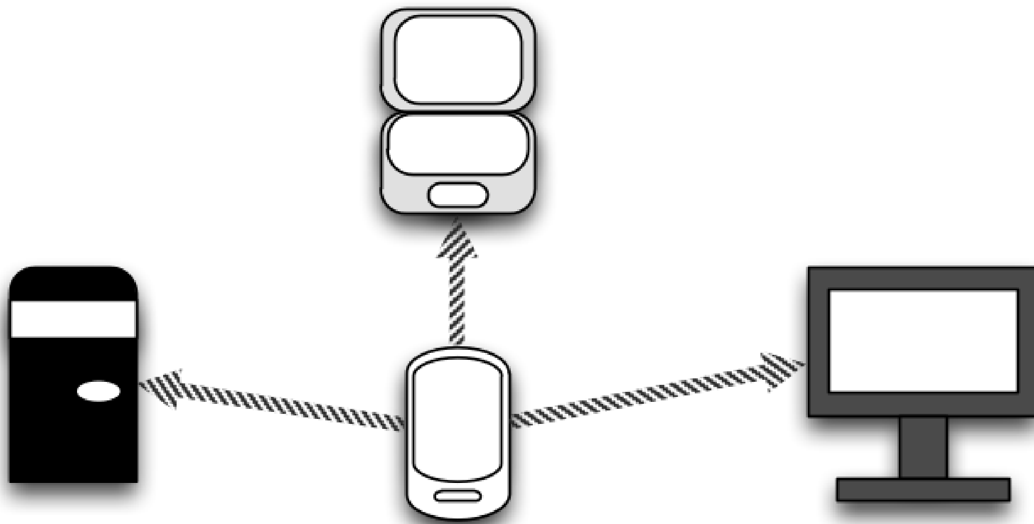


Figure 2.7: Files are being sent from one smartphone to multiple devices simultaneously.

3 Analysis

The goal of this analysis is to examine scenarios where one person tries to broadcast or multicasts data to multiple receivers over an unreliable medium using *Fountain Codes*. Furthermore, on the basis of the scenarios, needed requirements are extracted to help get an overview of components and their characteristics. With the analysis completed, the application required for the bachelor thesis can be designed and afterwards implemented.

3.1 Scenarios

Coping with the unreliable medium

When broadcasting data somewhere outside in public or in a big company, there are a lot of sources that can interfere with the transmission. Especially in big cities where a lot of shops often offer wireless internet connectivity through free hot spots, there are jamming signals that can prevent data to be transmitted successfully. But not only hot spots but other big electrical devices standing nearby can disrupt the sending process. So, unless avoided somehow, the sender must send the data for a very long time to ensure that everybody receives the picture completely. To bypass the long transmission times, Fountain Codes are used to optimize the broadcasting and multicasting in all scenarios.

In the scenarios described below the users want to send a picture and a document. But sending other filetypes, like music, short videos, or application updates, could be possible as well.

3.1.1 User wants to distribute data

Broadcast

Somewhere in the city a user holding an Android handset takes a picture of something interesting. He wants to share the picture with people that might find whatever he took a picture of interesting, too. That is why he needs his Android Smartphone to offer an application that

lets him send the picture to other people holding an Android device. In order to broadcast data, a connection to other peers is necessary. Connectivity can be achieved in two ways. The sender can establish a private ad-hoc network which enables other smartphone users to log into. The second possibility is to connect to a wireless access point. Of course, not every access point can be used to share the file. If the router has disabled broadcasting, the sent packets will be dropped and no receiver will see any packet. Nowadays, probably every open wireless lan in shops disables its broadcasting. It is because activating broadcast can lead to people trying to harm the network by, e.g., rendering the router useless by broadcasting useless data at a very high rate. With too many packets arriving, the router is busy broadcasting them and therefore unable to process the important data. But in the scenarios we assume that broadcasting is enabled.

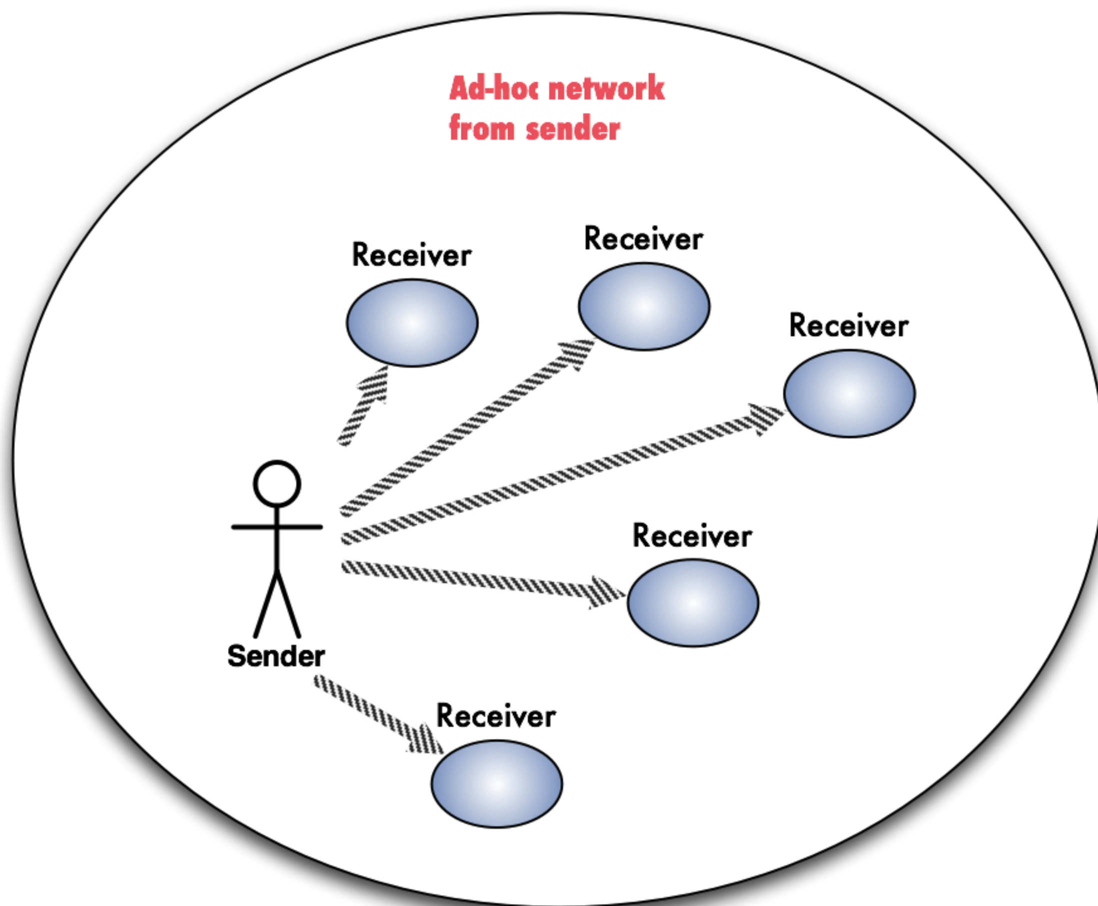


Figure 3.1: Sender broadcasting file to everybody in the ad-hoc network.

The sender activates his Wi-Fi module and establishes an ad-hoc network. Receivers can

log into the ad-hoc network and listen for data. The sender can now choose to broadcast the file to everybody in the same network or to multicast to a chosen subset of receivers (Multicast scenario described in 3.1.1). He chooses broadcast as he wants to release the file to the public. Now the sender needs to select the picture he wants to send. He browses through his smartphone to select the file. Now the user can start broadcasting the file by clicking a send button.
The sender must send an information packet along with the data packet to let the receiver know what kind of file he is about to receive. The Information could include filename, size, and filetype. Furthermore, the sender must label the information packet in some way to avoid multiple reception on the receiver side. In case a receiver finished receiving one file and afterwards wishes to receive another one, he must not be bothered by old file information packets. They need to be filtered out, so that the user is only prompted to accept or decline new files that are being broadcasted.

Multicast

Another scenario is the sender wanting to send some data only to a subset of receivers. Let us assume the application is used in some research project in a company. There are access points dedicated for multicasting researched data. A company can have several teams working on different projects.
In this scenario a researcher of a project wants to send analyzed data to only his project team. To do so, a session key between the team members must be established to ensure that only the selected employees are able to receive and decrypt the data (Detailed description of key agreement in section 3.1.3).

Just like in the broadcast scenario, the smartphone has to be connected to a network. In this scenario the sender connects to a wireless lan that is set up to offer a communication channel dedicated to exchange researched data between the project members. The user chooses the multicast option, offered by the application. He selects the data he wishes to share with his colleagues and pushes a send button to start a key agreement process and afterwards the transmission. First the sender waits until he receives some feedback about who will participate in the session then he starts a key agreement process with the members that gave the feedback. After agreeing on a session key, the data is encrypted and multicasted to the other members of the session.

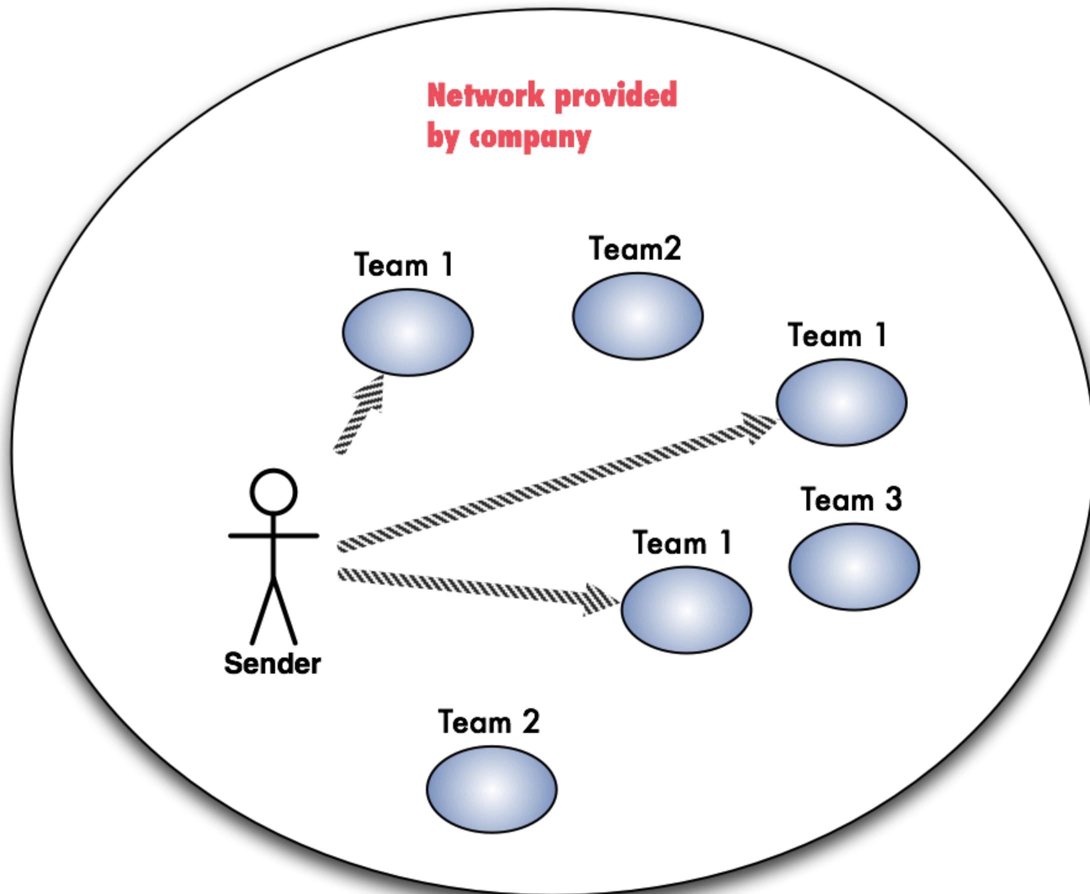


Figure 3.2: Sender multicasting file to members of his project team.

3.1.2 User wants to receive data

Broadcast

A person holding an Android Smartphone wants to listen for data that is broadcasted. The receiver certainly has to be connected to a local area network, to be able to listen for any data. That is why the user activates his Wi-Fi module. He sees the ad-hoc network that was established by the sender in 3.1.1 and connects to it. The user then pushes the receive button and starts listening for data. As the sender is broadcasting a picture, the application gets an information packet. The information packet informs the receiver that the data, he is about to receive, is a picture. It also includes the filename and size. The user can now choose to accept or decline the file. Maybe the phone offers limited memory space and the information

shows that the picture is too big. Accepting the file is illogical because the reception will fail anyways. So in case he declines, the information is stored and further packets belonging to that picture are ignored. But in this scenario the user accepts. The application now listens for the data and assembles the data packets until the whole file is received. Upon successful reception of the data, the application offers the user to open the received picture by providing an open button. In addition, the information about the data packets is stored to avoid multiple reception. When the user clicks the open button a preview is shown along with the file path to the picture or the document. If the file is a code update, the user is asked to allow the installation or to abort.

Multicast

The receive multicast scenario is linked to the multicast scenario of the sender in 3.1.1. The user connects to the same wireless lan as the sender. He starts the reception process by clicking the receive button. Since the sender is sending a file, an information packet is received. The information show that the file will be sent via multicast and that a key agreement is required before the actual send process can begin. In case the receiver accepts, he informs the sender that he will take part in the key exchange and the process is started. After a successful key was agreed upon, the data packets are received, decrypted, and assembled.

3.1.3 Scenarios for security measures

When an application is used by a lot of people and that application distributes data to multiple recipients, there are always people trying to intercept data, manipulate data to spread malware, or to harm users in some other way. That is why it is important to analyze the application in regard to security holes that could be used to harm the sender or receiver.

Data integrity

In this scenario an adversary tries to prevent the receiver from completely or correctly receiving the file by sending corrupt data packages on his own that are identified as packets from the actual sender (Figure 3.3).

The sender takes a picture and wants to share that picture with the public, like described in scenario 3.1.1. The application does not offer any security measures to ensure the integrity of the data packet. Therefore, an adversary has no problem intercepting the packet and manipulating the data as he wishes without the recipient knowing that the data has been

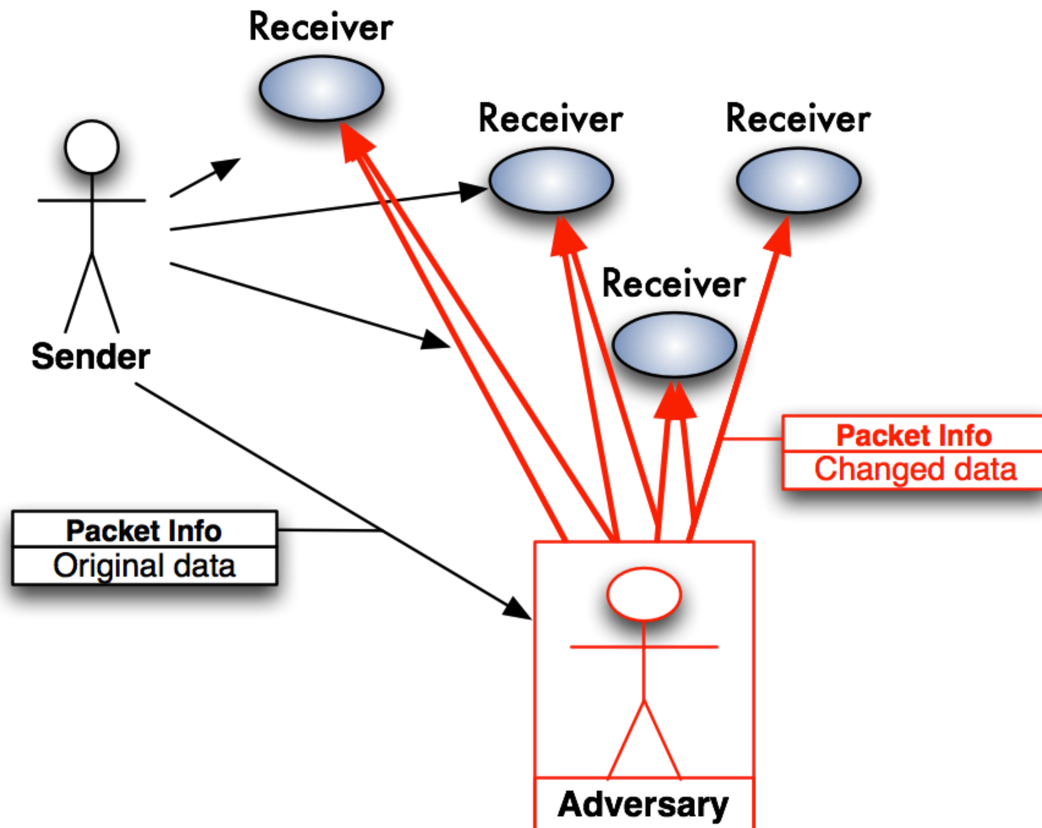


Figure 3.3: Adversary creating own data packet and sending at higher rate.

changed.
Let us assume the adversary listens for a data packet and receives one. He does not need to put in much of an effort to intercept packets as the file is broadcasted and everyone in the network can see it. He extracts the information that are needed to identify the packet, like filename, size and the packet label that specifies which session that packet belongs to. The adversary then computes his own data packet with the extracted information. Finally he appends some faulty random byte code that represents the data and broadcasts the packet. He can keep changing the data to avoid being identified as an attacker, in case the receiver has some kind of primitive algorithm filtering such identical packets. If the network is flooded by the corrupt packets and the receiver assumes that the received packets are genuine, then he will assemble a file that is useless. Depending on other information changed by the interceptor the recipient may not even be able to construct the file and keep listening for genuine data until the user aborts the process having failed to receive any file.

In conclusion, the sender must implement some mechanism that ensures the data's integrity.

Key agreement

Like described in the multicast section of 3.1.1 there are scenarios where files must not be read by strangers. In the multicast scenario the application is used in a company's research project team. Of course, the company and the project team do not want the researched data to be seen by everyone. That is why the data has to be encrypted by the sender. The team members agree on a symmetric session key and encrypt the data with that key. But a security issue has to be considered when agreeing on a key.

Using a strong and secure key is of great importance when encrypting data. To secure the researched data, the project team must posses a key secure enough to multicast the data without having to worry about an adversary intercepting the file and reading it. Thus, the team members have to agree on a secure key. A popular key agreement is the Diffie-Hellman Key Exchange (DHKE) Algorithm. The principle of DHKE is that two or more members agree on two very big numbers, one of them being a prime number. Then each participant chooses his own big secret number and computes an output that he then sends to the other members. A shared secret session key can be generated by computing the chosen secret number and the received numbers. But there is no assurance that only the team members of the project participate in the key exchange. In the multicast scenario above, the receiver participates in the key agreement by accepting the file. That is why an adversary could also accept the file and therefore be part of the key exchange. In conclusion there must be a security measure to exclude strangers from the key agreement process.

One could argue, why not generate a secure key and give it to every team member before they start working. Then no key agreement is necessary and the data can be encrypted with the pre-generated key. But there are some problems that can occur. If the generated key is not as secure as it is supposed to be, then figuring out the key could be very easy. Such a key would probably be used more than one day because handing out a secure key every day and inserting the key into the application is annoying. So if the key is used over a longer period of time, hundred of thousands of data packets would be encrypted with that key. If the adversary intercept enough packets, he could analyze the encrypted data and thus extract the key. Another problem could occur if a few researched data shall be shared with another team. The other team would have to get the secret key to encrypt the files. However, owning the secret key enables the other team to listen to any data subsequently exchanged. Therefore, a new key would have to be generated and inserted into the smartphone after the data has been shared.

3.2 Requirements

This section will list the requirements needed for the application that is implemented for this bachelor thesis. The requirements are extracted from the aforementioned scenarios in 3.1. The requirements are differentiated between requirements for the sending part of the application, the receiving part and the security measures needed to ensure save data communication. The requirements are divided into *functional* and *non-functional* requirements.
Functional requirements specify what functions the system must offer. *Non-functional requirements* specify how the system offers the functional requirements. They are used to list quality details.

3.2.1 Requirements for sender

The sender tries to send a file to multiple receivers (Scenario 3.1.1). The communication should be optimized for communication over an unreliable medium. Security aspects must be considered to ensure flawless file distribution. Services, such as taking pictures, recording videos, or any service creating the files, which are about to be sent, are assumed to be already supported by the Android Smartphone.

Therefore, following requirements are necessary:

Functional requirements

1. **Application should offer a dialog to select a file.** The sender must be able to browse through his smartphone to search for files he wants to share.
2. **Application should offer selection between broadcast and multicast.** Sender can choose if he wants everybody in the network to receive the file or only a selected subset of clients. To guarantee that only a selected subset of clients in the network are able to receive the data, security mechanisms are needed which will be listed later.
3. **Application should be able to establish an ad-hoc network** Very useful, as every sender can create his own private network and spare a public access point to be flooded by his data packets.
4. **Application should encode data before sending.** Due to the unreliable medium an optimized coding scheme is necessary to cope with the data loss.

Non-functional requirements

1. **The Dialog should hide files that are not supported by the application.** If the application does not support, for example, docx-files the user should not be able to select those files. That is why they must not be listed.
2. **Fountain Codes should be used to encode data.** To cope with data loss, Fountain Codes should be used.

3.2.2 Requirements for receiver

When a receiver gets the notification that a file is being distributed, he must be able to see some information about the file (Scenario 3.1.2) and get the chance to choose how to proceed. He must also know how to handle packets that where secured in some way.

Requirements for a receiver are:

Functional requirements

1. **Application should show dialog with information about the file.** Shows filename, filetype, and file size. Other necessary information are possible.
2. **Application should offer dialogs for user input.** The application has to know how to proceed when getting a notification.
3. **Application should be able to decode encoded data.** As the sender will use some coding scheme to encode data packets, the receiver must have an algorithm to process the encoded data packets.

Non-functional requirements

1. **User must be able to accept or decline the reception.** After the user is shown the file information, the application must be directed to accept or decline the file.
2. **Fountain Codes should be used to decode data.** Must be able to decode encoded data from sender.

3.2.3 Requirements for security

There are multiple security measures needed, to ensure save and complete data distribution (Sections 3.1.1, 3.1.2, 3.1.3).

Functional requirements

1. **Application should encrypt and decrypt data.** For secure multicast, the application must encrypt and decrypt data.
2. **Application should offer a key agreement procedure** To encrypt or decrypt data a key is necessary. As the users are in different locations, a secure secret key must be agreed on over the air
3. **Application should ensure involvement of genuine members in key agreement .** To ensure that only chosen users can participate in the key agreement, a security measure should be offered to exclude unwanted participants.
4. **Application should ensure data integrity.** To identify manipulation of data, a security measure is needed to ensure data integrity.

3.3 Summary

In this chapter we created some scenarios which showed how the application could be used and how it could work. Going through the scenarios, we were able to extract some security issues that must be considered and most importantly we were able to extract the requirements for the application. Now that the requirement are set, it is time to design the application. During the design we will see what requirement are possible to be implemented and what requirements should be changed.

4 Android

4.1 What is Android?

Most of the following information is extracted from [4].

Android is an operating system for handheld devices such as smartphones and tablets, hosted by Google. It was developed by Google and the Open Handset Alliance (OHA). The OHA is a business alliance comprised of many successful companies , software developers, and service providers. Its biggest members are companies like Samsung, Motorola, HTC, LG, Intel, Texas Instruments, and NVIDIA. The Open Handset Alliance was formed in 2007. Its target was to think about how to make mobile phones better.
Android is a the first *complete, open, and free* mobile platform there is.

- Complete because the developers took a comprehensive approach to create the Android platform. It has a secure operating system as its base and a robust software framework built on top.
- Open means that the Android platform is provided through open source licensing. The underlaying operating system is licensed under the GNU General Public License Version 2 and the Android framework is licensed under the Apache Software License (ASL/Apache2). Android lets developers have unprecedented access to the features when applications are implemented.
- At last Android is free since developing applications is free to everybody. No licensing or royalty fees have to be paid and also no membership fees are required. Every developer can distribute and commercialize his product in a variety of ways.

The first Android handset was released in October 2008. It was the T-Mobile G1 by HTC. Since then six other Android SDK's have been released. Every Android SDK has a different project name. It is named alphabetically after sweets. The first version was named Cupcake and the latest version of Android is codenamed Honeycomb. As of the second quarter of 2011 Android leads the worldwide shares of operating systems run on smartphones with 43.4% [12].

4.2 Android System Architecture

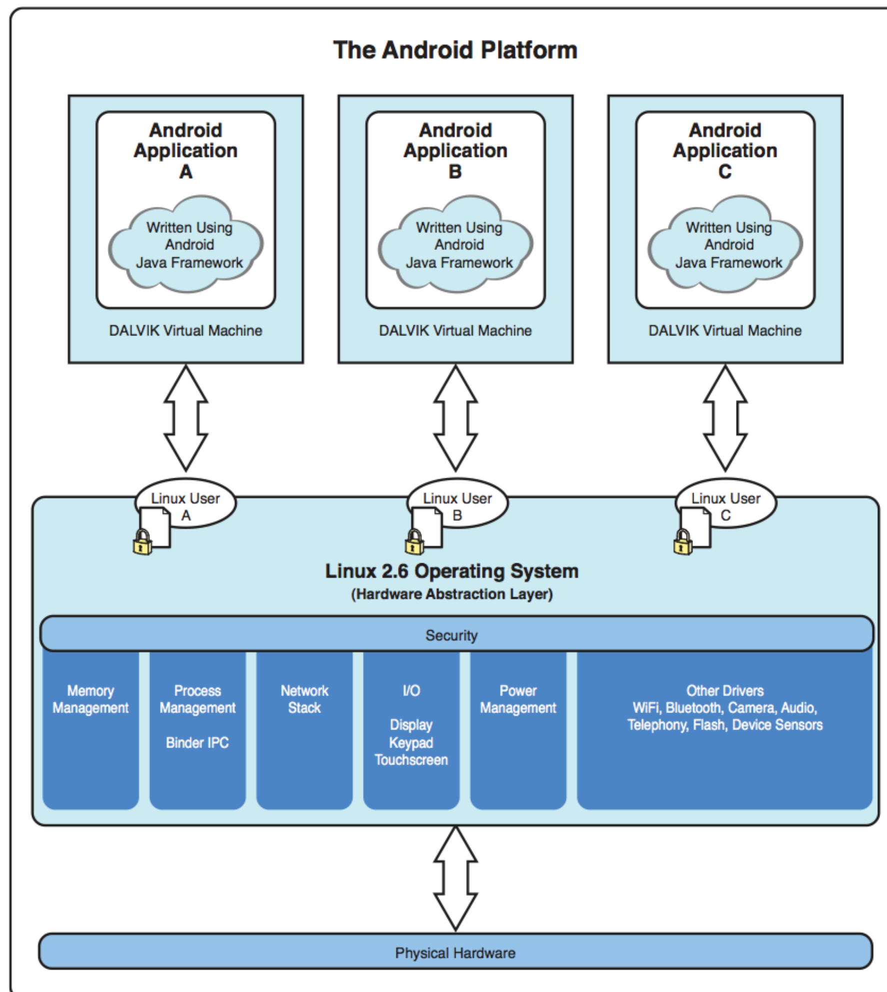


Figure 4.1: Android system architecture [4].

When developing the Android platform, the designers wanted the system to be more fault-tolerant than many of its predecessors. That is why Android runs a Linux operating system upon which Android applications are securely executed (Figure 4.1). The devices running Android are much less likely to crash due to the fact, that each Android application runs in its own virtual machine, the Dalvik VM. The applications are managed code. Therefore, Android devices show fewer instances of device corruption. The Dalvik VM is based on the Java VM. It was optimized to run on mobile devices. The Dalvik VM executes files in the Dalvik Executable format (.dex). The .dex-files are optimized to have a minimal memory footprint, so that multiple instances of the VM running an Android application are executed very efficiently.

The Dalvik VM relies on functions provided by the underlying Linux kernel. Some of the core functions the kernel handles are:

- Enforcement of application permissions and security.
- Managing the memory on a low-level bases.
- Managing processes and threading.
- Handling the network stack
- Managing Display, keypad input, camera, Wi-Fi, audio and flash memory driver.

4.2.1 Android Security

Android tries to maintain its integrity through a variety of security arrangements so that user's data is protected and the device is not infected with malware. But obviously, security measures still need some improvement, since recently a lot of Android devices where infected with different malware downloaded from the Android market.
Every time an application is installed on the Android handsets, a new user is created. That user profile is associated with only that application being installed. By managing applications with their own user profiles, each application has its own private files on the file system, its own unique user ID, and a secure operating environment. The private files cannot be accessed by other applications unless specifically configured. In order to access shared resources on the file system specific privileges are required. Those privileges are called *Permissions*. *Permissions* are registered for needed privileges when the application is installed. Some of these privileges enable the application to make phone calls, control the camera, access the network, or access information such as personal information, contact information, or user's location information.
Another method to secure the system is the signing of applications to built a trust relationship between developer and user. All Android applications are signed with a private key. The private key for the certificate is held by the developer. Self-signed applications are accepted and there is no Certificate Authority (CA) necessary that manages the certificates. If an application is not signed, the Android operating system does not allow the application to be installed. Furthermore, the signatures must always be the same for one application. If an application update has a different key than the current version, the update is not installed.

4.3 Android Programming Fundamentals

Android applications are implemented in the Java programming language. But there are some differences to normal applications developed for personal computers. Android has four important components that most applications implement. The *Context*, the *Activity*, the *Intent*, and the *Service*.

Context

The context is the most important part of an Android application. With the help of the *Context*, application-specific functionalities can be accessed. Retrieving a *Context* object, enables the application to gain access to application resources like layouts or strings that are stored in external files (Application resources will be explained later). Furthermore, application preferences can be retrieved, *Activities* can be launched, system services like the location service can be requested, and *Permissions* can be inspected and enforced, using the *Context*.

Activities

The Android *Activity* class is the core of an Android application. Each screen of an application is mostly implemented extending the *Activity* class. An application can have multiple screens and thus multiple *Activities*. For example, an application can have different tabs for different operations. Every class that extends the *Activity* must override the *onCreate()* method. The *onCreate()* method is like the *main()* method of a normal Java application. It is the first method that is called when a new *Activity* is started.

Intents and Services

Intents and *Services* will not be used in this application. *Intents* are asynchronous messages used to start other *Activities* of other applications. An *Intent* can include data that the called *Activity* can use or data that an *Activity* returns after finishing some computation. As we do not intent to user other applications for this bachelor thesis, *Intents* will not be used.

An Android *Service* is a process that runs in the background of an application. A *Service* is mostly used if some operation takes a long time and therefore should be decoupled from the main *Activity*. Furthermore, it offers the possibility to be repeated periodically. An example for a *Service* is a thread that checks for new mails every 5 minutes.

4.3.1 UI programming

Another difference to a standard Java application is the way the user interface is programmed. In normal Java applications the user interface is implemented programmatically using SWING or AWT. But Android approached the user interface differently. The UI is created using resources. These resources are stored in a separate folder of the application project, named *res*. The basic appearance of an application is defined in a XML file which is stored in */res/layout/*. In this layout XML file, text fields or buttons can be defined and positioned. The *Activity* then uses these XML layouts to create the actual UI.

Strings that are used to fill a text field are mostly included in another XML file stored in */res/values/*. The advantage of this approach is the very comfortable adjustment for different languages. The */res/values/* folder can have multiple subfolders of different countries. Each folder of a country has the same string values only written in their language. The Android handset checks the local preferences and calls the right language folder according to the set language.

Another important folder is the *res/drawable-*/* folder for pictures or icons. Since there are a lot of different Android smartphones on the market which have different display sizes, the application must be adjusted to the display. That is why there is one folder for high definition (/res/drawable-hdpi), one for middle definition (/res/drawable-mdpi), and one for low definition (/res/drawable-ldpi) screens. The smartphone again checks in which category the display falls and gets the right picture.

Programming the UI this way, makes the application highly flexible. The developer can add a new language without any great effort. He just adds a new folder and deploys the update. Other developers that are new to an application know where to go, if the UI must be changed or an icon has to be exchanged.
Despite the mentioned, very comfortable way to create user interfaces in Android, there is also still the possibility to implement the UI programmatically. But it is not recommended.

5 Application Design

In chapter 3, Analysis, scenarios where described which showed an application distributing files through broadcasting and multicasting. The scenarios where devided into three sections. By reference to those scenarios desirable requirements for the application were extracted.

This chapter is about designing the application based on the extracted requirements. In almost the same manner of chapter 3, this chapter will be sectioned into the components of the sender, the receiver, and the aspect of security measures needed to ensure flawless distribution. But before the designing of the three components is started, a short explanation regarding the connection of sender and receiver is given.

5.1 Wireless Local Area Network

Like described in the scenarios in chapter 3, the sender and receiver must be connected to the same network to communicate. There are two possibilities to achieve such connectivity. The first one is that the sender establishes an ad-hoc network. And the recipient can connect to that ad-hoc network. The second possibility is to connect to an access point that allows broadcast and multicast. The ad-hoc network is the preferred solution to establish a network because this way the sender can distribute files at all times. He does not need to depend on an access point being nearby. And as already mentioned in the Analysis chapter, nowadays the free hot spots do not allow the broadcasting of files.
Unfortunately, the Android OS does not support the establishment of an ad-hoc network yet. The only service the Android OS offers, regarding ad-hoc networks, is the ability to connect to an existing network. But the service must be unlocked first to use it. That is why the only way to distribute files with the application developed in this bachelor thesis, is to have an access point nearby and connect to it.
The upcoming Android Version, named *Ice Cream Sandwich*, probably will offer the possibility to establish an ad-hoc network.

5.2 Architecture Overview

Before starting to design the individual components in detail, it is beneficial to look at the application architecture as a whole. From the scenarios we know that the application needs two main components. The sender and the receiver. Other important aspects of the application are the security measure like data integrity, data encryption, and key distribution. And of course, as it is the basis of this bachelor thesis, the Fountain Codes and their integration into the application. Let us take a look at a first possible application architecture based on a sequence diagram (Figure 5.1).

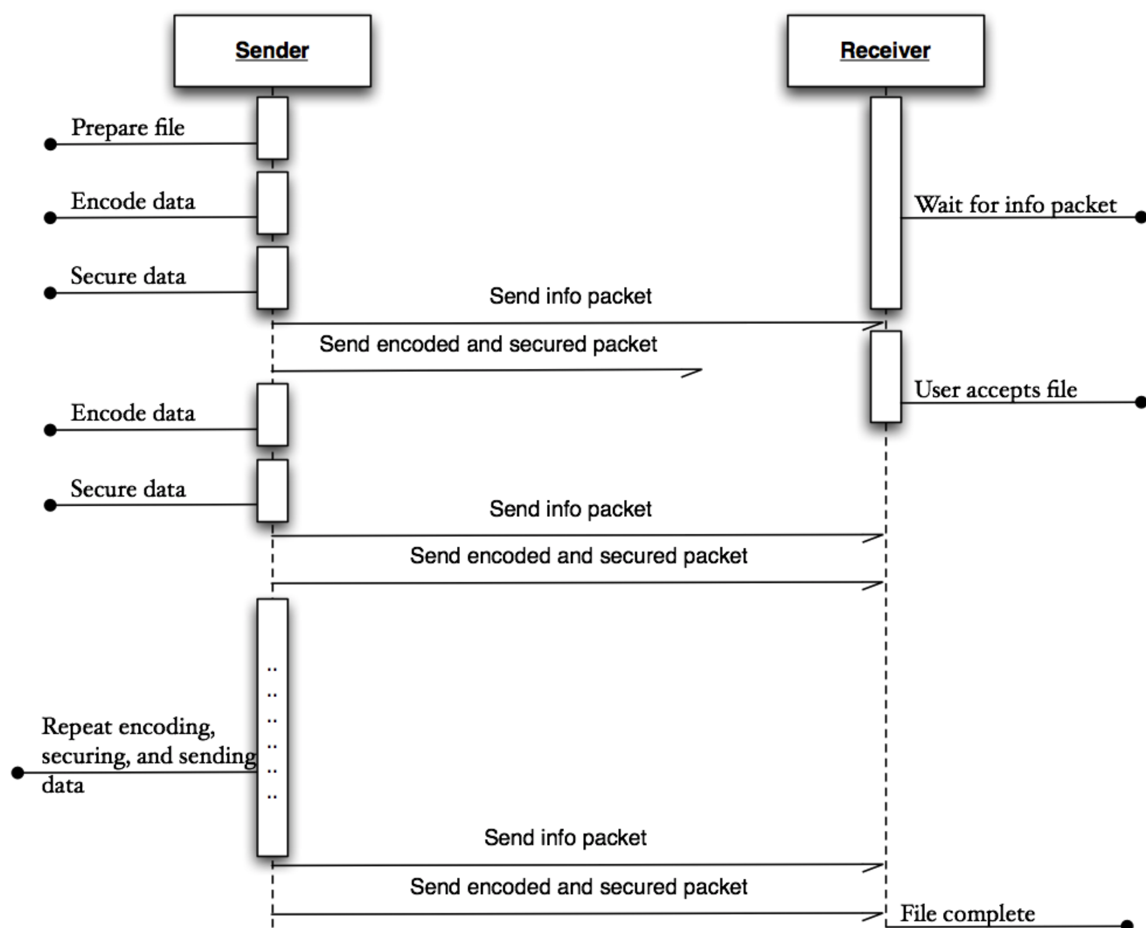


Figure 5.1: First possible application architecture.

In Figure 5.1 one can see that the application has only two components. One component for the sender and one for the receiver. The problem with this simple architecture is that if a new more efficient derivative of Fountain Codes is developed and is suitable for an application like

the one being developed in this bachelor thesis, then implementing it would be complicated. New developers would have to understand the code and integrate the new derivative of Fountain Codes into the code. Deleting the majority and most important part of an already written class and replacing it with core functionalities of a new encoding scheme can result in failure or an unstable application. Furthermore, if sending or receiving is done in the main activity, then user input will not be possible anymore since the sending or receiving process is running. Thus, the operation cannot be aborted. That is why it is a better solution to swap the Fountain Codes and the security measures into a dedicated class. This way the user interface and the user inputs are processed in the main sender activity and are not blocked by continuously running coding procedures. After processing the user input and therefore, for example, knowing if the file is broadcasted or multicasted, the computing and sending of the data is delegated to another object.

Almost the same principle should be used with the receiver's component. But there is another aspect with the receiver that should be considered. When the receiver is listening for a file, the first thing he is interested in is an information packet telling the receiver what kind of file the sender is sending. The information is shown to the user who can accept or decline the file. If he accepts, the Fountain decoding is started. Considering the process sequence, there is one question. If no file is being distributed or if the file is declined, then why should decoding resources be allocated. To avoid unnecessary allocation of memory and other resources, there should be a thread that is started by the main receiver activity to listen only for information packets. If an information packet is received and the user accepts the transmission, then the Fountain Decoder is started. Another advantage of saving resources is that when changing applications in the Android system and therefore pausing this application, the probability that the application is terminated by the Android OS is lower due to minimal resource allocation.

Keeping the disadvantages and their improvement possibilities in mind, the application's architecture and the communication between the individual components could be like Figure 5.2.

Now that an architecture of components is established, it is time to focus on the individual components and the security aspects. In the following first section the design of the security aspects are explored and explained. It is better to start with the security aspects because the sender and receiver use the security measures and therefore there is no need give references to upcoming explanations. After the security aspects, the Sender and its Fountain Encoder are designed in detail. At last the Receiver and its Fountain Decoder and Packet Information Listener are designed.

Figure 5.2: Sequence diagram of architecture to implement.

5.3 Security Design

Now it is time to focus on the design of the main components. Let us start with the security aspects of the application. The requirements extracted from the scenarios in 3.2.3 concerning the security measure needed to secure data in multiple ways are:

- Encryption and decryption of data
- Secure key agreement with only genuine members
- Data integrity

The first security issue that is going to be designed is the assurance of data integrity. Afterwards, the security aspect regarding the key agreement and its problems will be approached. There are some issues that must be considered when agreeing on a key in a broadcast environment. At last, after all members share the same key, the data is encrypted and decrypted by sender and receiver, respectively.

5.3.1 Data integrity

Data integrity means the recipient can be sure that the received data was not altered by an attacker and is therefore the original data the sender sent (See section 3.1.3). Data integrity is achieved by signing the data before distribution. The so called *Digital Signature* can then be verified by the recipient. The *Digital Signature* requires the following steps. First a message *digest* out of the data must be computed with a *hash function*. After generating the *digest*, it must be signed using asymmetric cryptography like the *Digital Signature Algortihm (DSA)*. Let us now turn to the details of *hash functions* and *Digital Signatures* with the *DSA*.

Hash Functions

In cryptography hash functions are widely used cryptographic primitives [5]. The principle of hash functions is, that they compute a digest of a data input. The output is a short, fixed length bit string. The message digest, or *hash value*, can be seen as an almost unique fingerprint of a message. So just like humans, messages can be identified by their fingerprint. Hash values are only *almost* unique as the output is of a fixed length but the possibilities of different inputs is bigger than that. So eventually there are messages that produce the same output. But good hash functions must make it computationally infeasible to find two or more messages that have the same output. Hash functions, in contrary to most cryptographic algorithms, do not require a key. The output is computed only through a save algorithm. Further requirements of hash functions are that they hash any data of any length to a fixed length hash value. So it does not matter if the data is 100 Bytes or 100 MBytes. The output length must be the same. Hash functions should also be performant because the input length can be very big. Furthermore, the hash function must be highly sensitive to input changes. If the message is changed in only a few places, the output must show a big change in the hash value.

Hash functions are necessary for Digital Signatures because of the subsequent cryptographic algorithm needed to completely sign the message, like the DSA. The DSA algorithm only allows a limited input length. 1024 Bits were mostly used in the past. 1024 Bits equal 124 Bytes. And most messages and files are much bigger than that. The message to sign could be split into equal blocks and each block could then be signed individually. But that would take to much time with a message of 1000 MB. In addition to the long processing time, the message overhead must be considered. When sending a signed message, the data will be twice the normal size due to the signed message blocks. And the integrity of the message would not really be given as the attacker could just remove, reorder or add individual blocks and their corresponding signatures. To avoid all of those problems, hash functions are used. The most widely used message digest function is the *Secure Hash Algorithm (SHA-1)*. It belongs to the MD4 family and produces an output of 160 Bits. But as the security of SHA-1 will

probably be broken some day, a new algorithm, the *SHA-2*, was already developed. *SHA-2* has outputs of up to 512 Bits of length (For more details see [5]).

Digital Signature Algorithm - DSA [5]

DSA is the algorithm used in this bachelor thesis to sign and verify the communicated data. DSA is a federal US government standard which was proposed by the National Institute of Standards (NIST). The signature output that is computed is 320 Bits long. DSA belongs to the asymmetric cryptography and therefore works with a key pair. The *public key* k_{pub} and the *secret key* k_{pr}. The key pair is generated by the person signing the data. The *public key* is made public so that everybody has access to the key. The *secret key* however, is kept secret by the one who generated the key pair. He uses the *secret key* to sign the data and the recipient can use the *public key* to verify the data. If the verification is successful, the recipient can be sure that the data he just received is indeed from the sender and that the data was not tampered with because only the sender owns the secret key which is compatible with the public key.

Followingthe DSA will be described in more detail. Figure 5.3 shows the steps, the one signing the data has to go through to generate the key pair k_{pub} and k_{pr}.

Key Generation for DSA

1. Generate a prime p with $2^{1023} < p < 2^{1024}$.
2. Find a prime divisor q of $p-1$ with $2^{159} < q < 2^{160}$.
3. Find an element α with $\text{ord}(\alpha) = q$, i.e., α generates the subgroup with q elements.
4. Choose a random integer d with $0 < d < q$.
5. Compute $\beta \equiv \alpha^d \bmod p$.

The keys are now:
$k_{pub} = (p, q, \alpha, \beta)$
$k_{pr} = (d)$

Figure 5.3: 5 steps to generate the key pair in the Digital Signature Algorithm [5].

The computation and verification is shown in Figure 5.4. According to the standard of DSA, *SHA(x)* has to be the *SHA-1* hash function that computes the message digest of the message *x*. When the recipient receives the signature *(r,s)* he computes *v*. The signature is only valid

if $v \cong r \bmod q$. Otherwise, the signature is not valid and the verification fails. In this case, the receiver can be sure that the data or the signature was manipulation in some way.

Done on sender side

DSA Signature Generation

1. Choose an integer as random ephemeral key k_E with $0 < k_E < q$.
2. Compute $r \equiv (\alpha^{k_E} \bmod p) \bmod q$.
3. Compute $s \equiv (SHA(x) + d \cdot r) {k_E}^{-1} \bmod q$.

Done on receiver side

DSA Signature Verification

1. Compute auxiliary value $w \equiv s^{-1} \bmod q$.
2. Compute auxiliary value $u_1 \equiv w \cdot SHA(x) \bmod q$.
3. Compute auxiliary value $u_2 \equiv w \cdot r \bmod q$.
4. Compute $v \equiv (\alpha^{u_1} \cdot \beta^{u_2} \bmod p) \bmod q$.
5. The verification $ver_{k_{pub}}(x, (r, s))$ follows from:

$$v \begin{cases} \equiv r \bmod q \Longrightarrow \text{valid signature} \\ \not\equiv r \bmod q \Longrightarrow \text{invalid signature} \end{cases}$$

Figure 5.4: Sender signs message and receiver verifies the signature [5].

Signing the data in application

Now that the basis of data integrity is explained, the explicit design for the application can start.

The sender uses Fountain Codes to encode the data. That is why signing only the encoded data is not enough. As was described in section 2.2, much more information is needed to decode the data. The recipient needs to have the degree of the code word, the coefficient vector, and the code word itself. If only the code word was signed, an attacker could just

change the degree or the coefficient vector and thus the receiver would again not be able to decode the data correctly. With a changed coefficient vector, the decoder would XOR false data blocks into the code symbol and consequently get a useless result. The recipient would not know that the result is useless, though. Only after the full reception of the data he would realize that he assembled a damaged file. Therefore, the degree, the coefficient vector, and the encoded data must be signed. The only information the attacker can change is the label linking the packet to a session. But that does not matter. If the label is changed, the receiver filters the packet out anyway.

The next issue regarding the signature is how the public key k_{pub} can be made public so it can be accessed by the recipients. When working with asymmetric cryptography the public keys are often provided in the internet by some server. A user wanting to verify signed data, downloads the sender's public key and verifies the signature. But that approach cannot be taken in this application. There is no assurance that the network, the communicating users are logged into, is actually connected to the internet. If no internet is available, then no verification is possible. That is why the easiest and only solution is to send the public key along with the information packet. After the key pair is generated by the sender, he includes k_{pub} in the information packet. If the receiver accepts the file he can verify all data from the sender with that key.

Conclusion

In this section the aspect of data integrity was addressed. The data's integrity in this application can be achieved by using a hash function like *SHA-1* and signing the hash with the signature algorithm *DSA*. The sender sends an information packet with necessary file information and his public key k_{pub}. The recipient can extract the public key and verify the signatures of incoming data with it.
The question is if data integrity is assured completely. The answer is, not completely. In case the file is *broadcasted* and therefore not encrypted, some users can still be fooled. An attacker could create his own information packet with the information extracted from an original packet and insert his public key. He then generates encoded packets and signs them with his private key. Users who get the adversary's information packet first, will verify the fake encoded packets successfully and the verification of original packets will fail due to the wrong public key they are holding. Affected are user who join the session a little later. But users who already received the original information packet are no longer affected by manipulated data.

5.3.2 Establishing a session key

This section of the security design is about finding a way to agree on a key. The objective is that only selected members share the key and that attackers who wish to obtain the key are excluded. The best known key agreement protocol probably is the *Diffie-Hellman-Key-Exchange (DHKE)* protocol. The principle of DHKE is that two or more members agree on two public parameters. The first parameter is a very large prime number p. The second parameter is an integer $\alpha \in (2, 3, ..., p-2)$. The agreement of the public parameters can be achieved by one member choosing the parameters and sending them to the other members. The following description is for two members agreeing on a key.
Member A chooses a secret integer $x \in (2, ..., p-2)$ and computes a public key $k_{pub,A} = \alpha^x \bmod p$. A then sends $k_{pub,A}$ to member B. B does the process vice versa. He chooses $y \in (2, ..., p-2)$, computes $k_{pub,B} = \alpha^y \bmod p$, and sends it to A. The secret session key k_{AB} is then computed by each member with $k_{AB} = k^x_{pub,B} \bmod p$ for A and $k_{AB} = k^y_{pub,A} \bmod p$ for B. Now A and B have computed the same secret session key.
An attacker intercepting the public key, $k_{pub,A}$, and $k_{pub,B}$ cannot extract the secret key because he does not know which x and y were chosen by A and B. The function that is used for DHKE is a one-way function. That means that $f(x) = y$ is easy to compute but computing the inverse $f^{-1}(y) = x$ is computationally infeasible. It is based on the *Discrete Logarithm Problem (DLP)* (For more information see [5]).

Issues with DHKE

The question is, can this protocol be used in this application. There are some problems with this approach of key agreement. One problem was already explained in section 3.1.3. Without any additional mechanism everyone in the same network could participate in the key agreement. They would just have to choose a secret integer, compute the public key $k_{pub,attacker}$ and send it to the other members. Therefore, his public key would be part of the protocol and he would generate the same key. Another already known issue with DHKE is the Man-in-the-Middle attack (MitM). But the MitM-attack is less likely to be used in this particular application. The attacker does not have to pretend to be to member A or B as he can simply participate in the key agreement.

A far more troubling problem is the communication of the participating members between each other. Each participant must receive the others public keys to generate the same secret key. But the users do not communicate using a reliable protocol like TCP but an unreliable protocol. And on top of that the data is shared over an unreliable medium with possibly many interfering sources. So there is a high probability that some of the agreement members generate a secret key based on incomplete public keys. In addition to just sending the public key once, each participant must send further data if more that two users are taking part in

the key agreement.
When three members are participating, following procedure steps could be traversed - Steps 4 to 6 could differ if the participants compute other public key combinations:

1. p and α must be chosen.
2. Members A, B, and C choose secret x, y , and z, respectively.
3. Everyone computes their public key $k_{pub,member} = \alpha^{chosenSecret}$ mod p and sends the key to the other members.
4. A computes the subkey $k_{AB} = k^{x}_{pub,B}$ mod p, B computes the subkey $k_{BC} = k^{y}_{pub,C}$ mod p, and C computes the subkey $k_{CA} = k^{z}_{pub,A}$ mod p.
5. Now A needs the result of B because the function to generate the shared key k_{ABC} is $k_{ABC} = \alpha^{xyz}$ mod p. Since $k_{BC} = \alpha^{yz}$ mod p, computing k^{x}_{BC} mod $p = \alpha^{xyz}$ mod p. That is why every member has to communicate his computed subkey to another member.
6. B and C do the same, i.e., B: $k_{ABC} = k^{y}_{CA}$ mod p, C: $k_{ABC} = k^{z}_{AB}$ mod p.

One can see that additional communication must be done when several members are involved in the key agreement. Furthermore, additional information is needed. Each recipient must identify himself in case he takes part in the key agreement. The participants must synchronize themselves so that everybody knows which public key they must compute to help other participants. With four people agreeing on a key, A, for example, must get subkey k_{BC} to compute another subkey k_{ABC} which he then sends to D so that he can compute the shared secret key k_{ABCD}. With increasing members the necessary sub keys that must be computed and shared over the unreliable medium increase too. With 20 people involved in the agreement process a lot of data must be shared before the actual transmission of the file can be started. And to make sure that everybody has received the necessary keys from the other members, the key agreement must go on for a very long time. In conclusion, using the DHKE with an unreliable connection such as UDP over an unreliable medium is very inefficient and thus not recommendable.
Another solution to make sure that every genuine member holds the same secret key must be found.

Design of key distribution

Keeping the issues described above in mind, a design must be found that avoids those problems.
A solution to the problem concerning the authenticity of session members could be to simply add encryption to the basic DHKE. The public parameters and public keys could be

encrypted. A password is used as a key and the public values are encrypted with that password. That way, only users who are in possession of the key are able to decrypt the public parameters and keys. As the amount of public values that must be sent over the air is not very much, the password does not need to be very strong. An attacker cannot extract the password with only a few packets to analyze. Of course provided that the password is not too simple.

Regarding the problem with the communication burden created by the DHKE protocol, another way to distribute the secret session key is used. In cases where all members actively take part in the key agreement, there is no way to avoid additional communication. That is why an *agreement* of a key is not optimal for this application. Instead, a more centralized solution is the smarter way to go. The sender generates a strong and secure key $k_{pr,session}$. He adds the key to the information packet and signs the information and the key, Afterwards, the information, the secret session key, and the public key for the signature verification are encrypted with the password. Now the information packet is ready to be broadcasted. Only members who know the password are able to decrypt the information of the file and the two keys. As the sent information packet is always the same for one session, an attacker can only do crypto analysis on one packet. That is surely not enough to extract the password.

5.3.3 Data encryption

The last part of the security aspect is the encryption and decryption of data. It is the simplest part of the security measures because all basics for securing the data are already provided in the previous section. The data is encrypted and decrypted with the secure key $k_{pr,session}$ generated by the sender. Each member who receives the information packet and is in possession of the password, which was used to encrypt that packet, is able to extract $k_{pr,session}$ and thus decrypt the incoming data. The only question left is which parts of the encoded packet must be encrypted. Luckily not much. It is sufficient to just encrypt the coefficient vector of the encoded packet. The data itself is already encrypted since it was XORed together with other data blocks. And if a recipient does not know which data blocks are included in the encoded symbol, he cannot decode the packet correctly. To encrypt and decrypt the data the *Advanced Encryption Standard (AES)* should be used. It is still very secure and executes the cryptography procedures efficiently.

5.4 Sender Design

This section is dedicated to the design of the sender. The sender consists of two main classes. More than these two classes will be needed to build the complete sender but the

following two classes are the important ones. The fist class, the Sender, is the main class. It is the activity that controls the sender. It accepts user inputs and starts the Fountain encoding. The second class is the Fountain Encoder class. It executes the encoding, secures all necessary data like described in the previous section, and it sends the data. The task is stopped by the main activity when the user aborts the file distribution.

5.4.1 Sender

The first aspect to design is the user interface. From the analysis chapter and the security design section the following is known:

- User must be able to enter session name.
- User must be able to choose between *broadcast* and *multicast*.
- User must be able to browse through file system.
- User must be able to enter password for secure key distribution.
- User must be able to start the send process.
- User must be abet to abort the send process.

The user interface could look like Figure 5.5.

If the option broadcast is selected, the password text field is not used even if a password was entered. A password is only needed in multicast sessions. In case multicast is selected and no password is entered, a dialog must inform the user to do so. Without the necessary information the sending process is not started.
As soon as the mode and its necessary inputs are chosen, the user can start the sending process by clicking the *Send* button. Clicking the *Send* button starts the Fountain encoding and all connected procedures, like signing the data. The main sender activity and thus the user interface is decoupled from the Fountain Encoder. Therefore, if the user aborts the send process by clicking a button, the application is able to react to that user input and cancel the Fountain encoding process.

5.4.2 Fountain Encoder

The Fountain Encoder class is the most important class of the sender. The Fountain Encoder should be implemented as a thread because it executes long running operations like encoding data, signing data, encrypting data, and sending necessary information for the receiver.

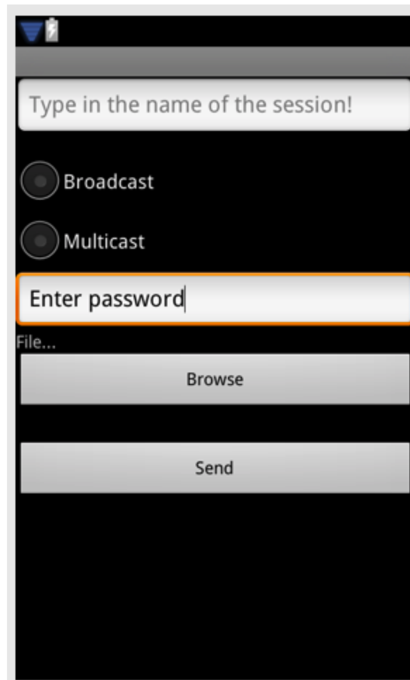


Figure 5.5: User interface for application.

Information Packet

There are two kinds of information packets needed. The first one is a simple information packet. It is created and send when the file is being *broadcasted* to everybody in the same network. The information packet holds the *session name*, *file name*, *file size*, *file type*, and the public key which is needed to verify the signature of the encoded packets. This information packet does not include any security measures. Its content is not encrypted in any way.
The second information packet is for multicast sessions. It holds all the information that the first one holds. But on top of the basic information, the generated secure session key to encrypt the data is included in the packet. Furthermore, all the information is signed with the included public key and encrypted with the password that the sender entered into the password field of the user interface.
One of these two information packets is sent periodically to all listening receivers depending on the mode selected.

Fountain encoding process

Now it is time to focus on the important part of the Fountain Encoder. The actual Fountain encoding.

Before the sender can start encoding the data, it has to be split into equal data blocks. The data cannot simply be split into one byte blocks because doing so would take too long to distribute the data. We will choose a data block of about 1000 Bytes. The router will probably have an MTU(Maximum Transmission Unit) of maximal 1500 Bytes because we use the wireless lan to connect the devices. We could choose a bigger block size but that would require fragmentation of the encoded packet since an ethernet frame can only hold as many bytes as are available through the MTU. If the channel is noisy and part of the fragmented encoded packet gets lost during transmission and must be retransmitted, the receiver has to wait until he has the whole encoded packet. With a block size fitting into one frame, the receiver instantly has data he can work with. The block size cannot be the complete MTU since some space must be available to add the additional data like the coefficient vector which is sent along with the encoded symbol. The split data blocks should be represented by their own class to make it easier to handle them. In conclusion, a class *Data* is needed which represents the data blocks.

After the data is split into appropriately big data blocks and available for encoding, the encoding can start. To encode the data blocks, from now on referred to as *data*, a *degree distribution* and a scheme to uniformly select *data* at random is needed.
The encoding process is done according to the LT Encoding process in 2.2.1. A degree *d* is taken from the *degree distribution*. Corresponding to the value of the degree, *d data* are chosen at random and encoded. A coefficient vector holding the index of each *data* picked is created simultaneously. Now the encoded symbol and the coefficient vector are ready to be sent. The flowchart in Figure 5.6 shows the procedure. To send both objects in one UDP datagram another class is needed that encapsulates them. The *EncodedPacket* class. A *EncodedPacket* object includes the *coefficient vector*, the *data*, and the *degree*. Furthermore, a session name is needed to link the packet to a session.

Before the *degree distribution* is described, the encoding and sending sequence should be dealt with in more detail. As described in 2.1, Fountain Codes are rateless and therefore unlimited amounts of code words can be generated. As a consequence the sender can generate new encoded packets on the fly and keep sending them until the user aborts the process. But one must keep in mind that the application is run on a smartphone. And unfortunately the smartphones today barely run two days with one full charged battery. The average smartphone user must even recharge his handset everyday. Certainly, the encoding itself does not take that much computation time and effort but the encoded symbol must be signed and, when chosen, encrypted. Traversing all the necessary steps for an encoded packet hundreds of thousands of times over and over again, can heavily strain the battery and even damage it. That is why it is smarter to pre-generate a sufficient amount of encoded packets and send them iterating through that packet pool. So only the send process of encoded packets is done in a loop.

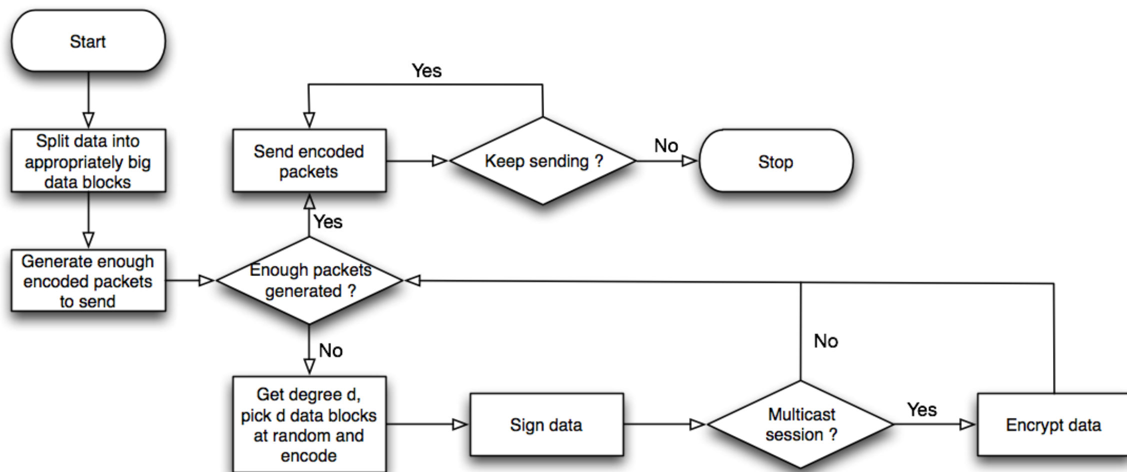


Figure 5.6: Flowchart showing procedure of the Fountain Encoder.

Degree Distribution and random selection of data blocks

Like described in section 2.2, the *degree distribution* is of utmost importance. Without a good *degree distribution* the Fountain decoding can take a very long time because the receiver could need much more encoded packets to decode the data than the actual amount of data blocks the complete data is split into. That is why a *degree distribution* has to be found that leads to efficient encoding so that the decoding process is completed with only slightly more received encoded packets than the amount of data blocks. In addition to that, finding the right scheme to select the data blocks is very important as well. Having found an optimal *degree distribution* but choosing the data blocks unluckily can be as inefficient as choosing a bad distribution. If the data blocks are selected at random without any system, the same data blocks could be selected several times. As a result the decoder would receive an encoded packet which he decodes and subsequently discards since the information encoded into the packet were already received through a previous packet.

In [6] the authors of the paper optimized their *degree distribution* for small messages. Using an importance sampling approach they tried to find a *degree distribution* that is optimal for messages that are split into 16, 32, 64, or 128 blocks. As one can see in Figure 5.7 the value of the degree starts with 1 and is doubled until half the data block amount is reached. Each degree is given a probability p_i of occurrence. The probability of the degree decreases with the increasing degree value which results in an average degree that is rather low than high. A low average degree is beneficial since it reduces the amount of operations that must be performed. Taking the *degree distributions* in Figure 5.7, leads to an overhead of only 25%-40% which is very good.

K →	16	32	64	128
p_1	0.221	0.212	0.161	0.187
p_2	0.457	0.351	0.400	0.339
p_4	0.188	0.288	0.256	0.275
p_8	0.134	0.101	0.101	0.101
p_{16}	-	0.048	0.045	0.046
p_{32}	-	-	0.037	0.031
p_{64}	-	-	-	0.021
$\mathbb{E}[N]$	22.6	43.6	82.7	158.7
Std. $\sigma(N)$	4.4	6.4	9.1	11.4
$\mathbb{E}[N]$ in [15]	22.5	43.6	81.9	159.8
Std. $\sigma(N)$ in [15]	4.2	6.8	7.7	12.1

Figure 5.7: Table showing found degree distributions and their results [6].

We will take the findings of [6] as the basis for the *degree distribution* of the application. The question probably is, why consider results that are based on a maximal length of 128 data blocks. The files distributed in the application are much bigger than that. But as one can see in Figure 5.7 there is a pattern in the *degree distribution*. Small degrees are distributed more often than the bigger degrees. We will take the *degree distribution* of K = 128 and adjust the probabilities for the degrees and combine them with an appropriate selection scheme to see if such a pattern can be used to distribute data with different sizes efficiently. After the implementation is completed the performance and efficiency of the *degree distribution* will be evaluated.

5.5 Receiver Design

The receiver is comprised of the three classes as was elaborated in section 5.2, Figure 5.2. The three classes are the *Receiver*, the *Packet Information Listener*, and the *Fountain Decoder*.

5.5.1 Receiver

Let us start with the *Receiver* class. Just like the class *Sender* in the previous chapter, the *Receiver* handles the user inputs and starts another thread, the *Information Packet Listener*. The receivers initial user interface is very simple because only one button is needed to start the listening process (User interface and dialogs shown in Figure 5.8). If an information packet arrives, a dialog is created that shows the information and waits for the user to accept the file or decline it. In case the information packet is encrypted, a dialog asks the user to enter the password before the accept dialog is shown. After a file is received, a new view item is added to the initial user interface. It shows information about the file and a button to open the file in a preview if the file is a picture or music file, or install the application update if the file is a code image.

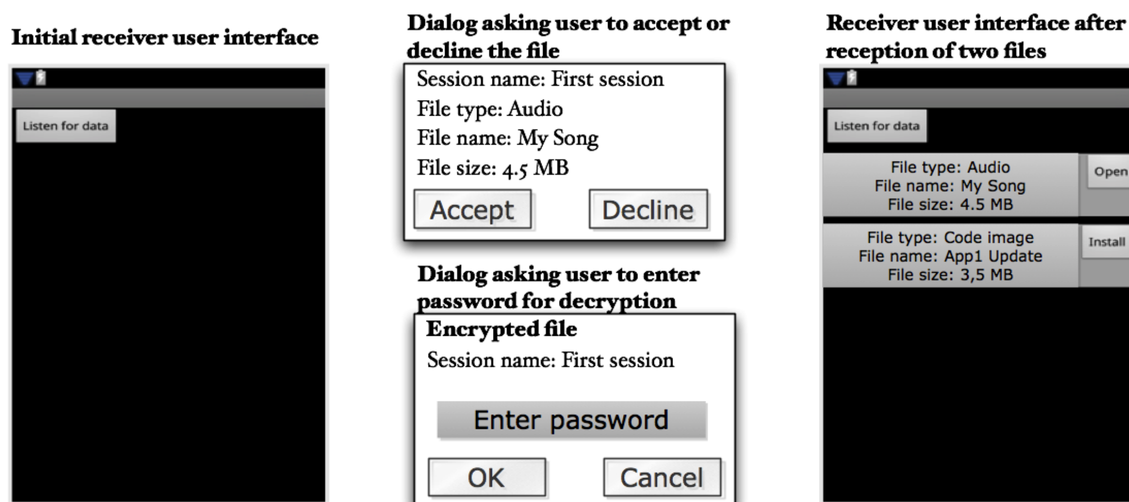


Figure 5.8: User interfaces and dialogs for receiver.

5.5.2 Information Packet Listener

The *Information Packet Listener (IPL)* is the task that is started by the *Receiver*. Its purpose is to listen for information packets, analyze them, and update the user interface accordingly. When an information packet arrives, the *Information Packet Listener* first checks if the same information was received earlier. If it was received, meaning that the session name is already stored, the information is discarded and the listening process continues. But if the information is new, the *IPL* starts analyzing the information. First it checks if the file is encrypted. In case of an encrypted information packet, the IPL updates the user interface with the password dialog of Figure 5.8. After the correct password is entered, the IPL must verify the

information's signature to assure the integrity of the information packet. The user is informed of an incorrect password submission if necessary and the password dialog is shown again. Unless the verification fails, the accept dialog is presented to the user and the packet's information is shown. Otherwise the user is informed that the verification was unsuccessful and the IPL starts listening for new information packets again. When the user finally accepts the file, the *Fountain Decoder* is started. The flowchart in Figure 5.9 visualizes the procedure of the *IPL*.

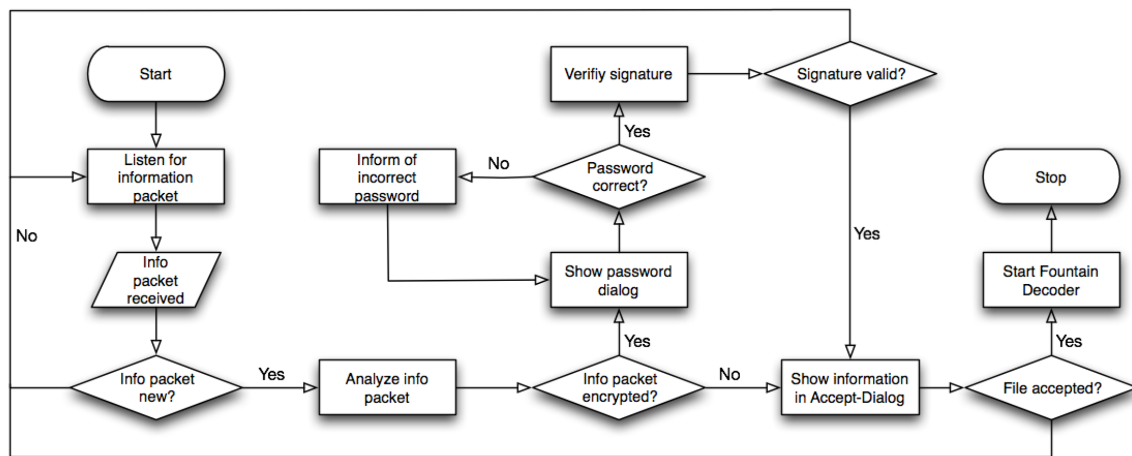


Figure 5.9: Flowchart of Information Packet Listener.

5.5.3 Fountain Decoder

The *Fountain Decoder (FD)* is the last class of the receiver. Objects created from this class listen for encoded packets and decode them using the *LT Codes* decoding algorithm (See Section 2.2.2).

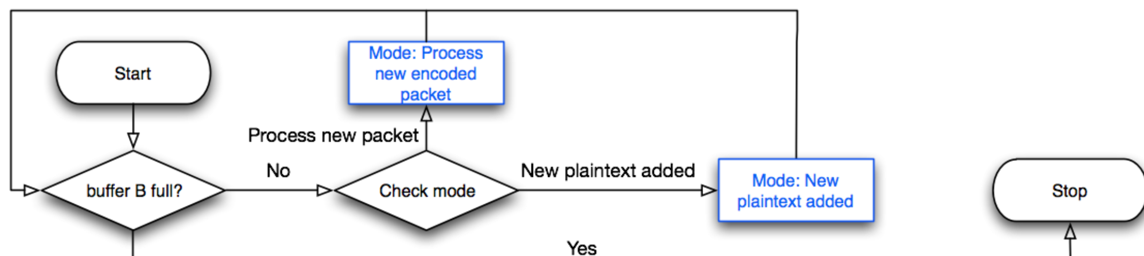


Figure 5.10: Flowchart of Fountain Decoder as a whole.

The *Fountain Decoder* works in two different modes which are marked blue and shown in Figure 5.10. The first mode, the *Process new encoded packet* mode , is the initial mode that the *FD* works in (Figure 5.11). The *FD* listens for encoded symbols from the sender. As soon as an encoded packet is received, the *FD* checks if the packet belongs to the actual session. In case of an packet belonging to another session the packet is dropped and the listening process is started again. Otherwise, the encoded packet can be processed. First, the signature is validated. Only if the verification was successful the analysis is continued. After the verification, the packet is checked for encryption. If the packet is encrypted, the *FD* uses the session key, which was generated by the sender and extracted from the information packet, to decrypt the data. Now that the plain encoded packet is available the decoding is started (Detailed decoding process described in section 2.2.2). After the encoded packet is decoded as far as possible, the packet degree is checked. A degree bigger than one indicates that the packet was not fully decoded yet and is therefore stored in buffer *A*. A degree that equals one indicates that the packet was completely decoded. Thus, the plaintext is stored in buffer *B*. Furthermore, the mode is switched to *New plaintext added*.

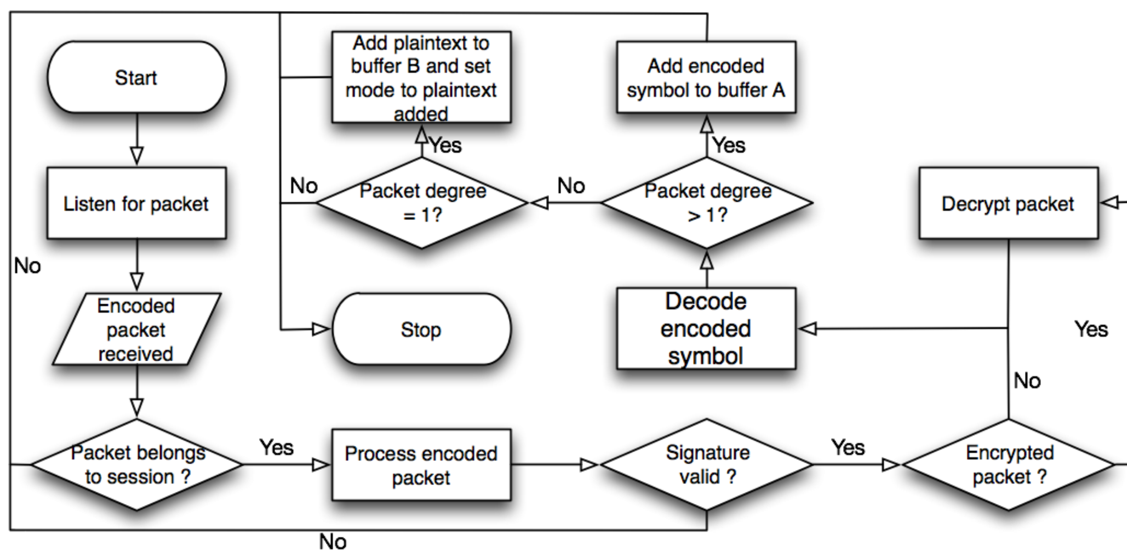


Figure 5.11: Detailed flowchart of mode *Process new packet*.

The *FD* in mode *New plaintext added* works as depicted in figure 5.12. First the mode is set to *Process new packet*. Before the decoding of encoded packets in buffer *A* can start the *FD* must assure that there indeed are packets stored in buffer *A*. With no packets stored in *A* the mode is switched back to *Process new packet*. But if the buffer is not empty, the decoding procedure can start. The *FD* iterates through the buffer one by one. He takes a packet and tries to decode it. If the degree equals one, the plaintext is stored in buffer *B*. Of course, the plaintext is only stored if it was not completely decoded in a previous decoding

step and therefore already in buffer *B*. Furthermore, the packet is removed from *A* and the mode is set back to *New plaintext added* again. In case the packet degree equals zero, the packet is removed from buffer *A*. Having completely iterated the buffer *A*, buffer *B* is checked for completeness of contents (Figure 5.10). If plaintexts in *B* are still missing the procedure of figure 5.10 is repeated.

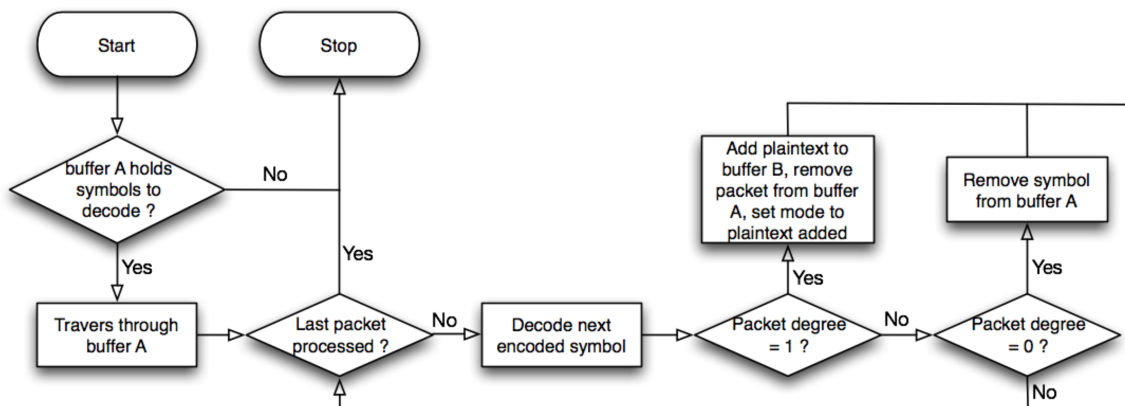


Figure 5.12: Detailed flowchart of mode *New plaintext added*.

5.6 Summary

In this chapter we designed all necessary components of the application. On the one hand we focused on the security aspects of the application and evaluated their feasibility. That is why, for example, we found out that using the *Diffie-Hellman-Key-Exchange* protocol in this particular application would not be very efficient and thus we approached the key distribution differently. On the other hand we designed the main two sides of the application. The Sender and the Receiver. We acquired that swapping the Fountain Coding to an external class is much clearer and more maintainable for later developers extending or changing the application. We also realized that some resources should only be allocated when necessary. With the help of the flowcharts the particular classes were visualized to better understand the procedures described.

Now that the design is finished, the implementation of the application can finally start.

6 Application Implementation

This chapter is about the implementation of the application. The implementation is based on the design from chapter 5. The implemented classes, their member fields, and the important methods will be explained. UML class diagrams are shown in figure 6.1 and figure 6.2.

The application is made up of two tabs. One tab is for the sender and the other tab is for the receiver. Only one tab can be chosen at a time. Each tab is an *Activity*. Like described in the Android chapter, an *Activity* is an instance that is running actively when a new view is shown in the Android OS.

6.1 Sender

6.1.1 SenderTab

The main sender class is named *SenderTab*. The *SenderTab* extends the *Activity* class. An object created from this class is processing user inputs, updating the user interface, and starting the Fountain encoding. Its main two methods are *sendMessage* and *onCreateDialog*. The *onCreateDialog* method is an inherited method from the super class *Activity* and must be overridden. It is called when the *showDialog* method is called. By reference to the parameter given from the *showDialog* method, a different dialog is created. There are three different parameters and thus three different dialogs that are created. The first dialog is for showing the file list of the smartphone. The user can browse through his smartphone using this dialog. The second dialog is for showing a progress window. Showing the progress dialog ensures the user that the application is currently sending the file. The last dialog is shown when the user presses the *Back* button. The dialog asks the user if he really wants to abort the send process or not. The *sendMessage* method is called when the *Send* button is pressed. In this method all necessary information are extracted from the user inputs and the Fountain encoder is started with the extracted information as parameters.

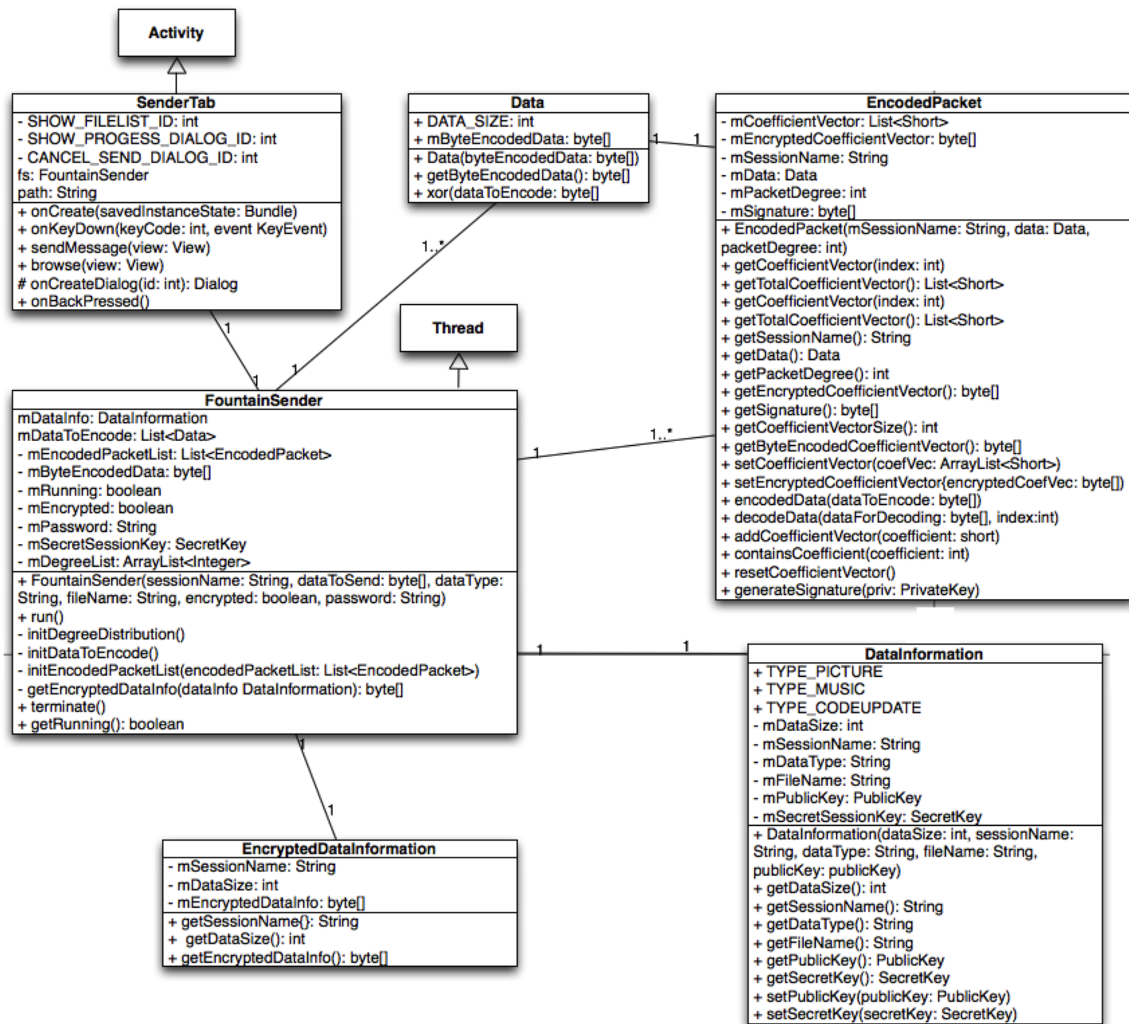


Figure 6.1: UML diagram of the sender.

6.1.2 FountainSender

The Fountain Encoder class is named *FountainSender*. It extends the *Thread* class as is must run simultaneously to the *SenderTab*. The *FountainSender* has four important methods. The *run*, *initDegreeDistribution*, *initDataToEncode*, and the *initEncodedPacketList* method. The *run* method is the method that is called when the thread is started. The procedure done in this class is the same as depicted in Figure 5.6. In *run* the following methods are called.

The *initDegreeDistribution* is the first method that is called. It initializes the degree distribution. The second method call is the *initDataToEncode* method. This method splits the file into

equal data blocks. The data blocks are represented by the class *Data*. The class *Data* offers the method *xor* which encodes two data blocks. The last method, *initEncodedPacketList*, initializes an ArrayList that holds all encoded packets represented by the class *EncodedPacket*. The encoded packets are then taken in the *run* method and sent. In the *initEncodedPacketList* method the encoded packets are signed and, if chosen by the user, encrypted. The signature is created by using the *SHA-1* message digest and the *DSA* algorithm. The key pair created for signing the data is 512 Bits long. Some may think that 512 is too short and thus insecure but taking a bigger key results in longer signing and verification times. And as a new key pair is generated with each file send, 512 Bits are sufficient. The encryption is done using an *AES Key* and an *AES Cipher*. The cipher works in the *ECB (Electronic Code Book)* mode with PKCS5Padding and encrypts only the coefficient vector. After the necessary initializations are completed the *FountainSender* sends the encoded packets and data information packets.

6.1.3 Other classes

EncodedPacket

The *EncodedPacket* has five member fields. Most of the member fields are information that are needed for the Fountain Codes to work like the coefficient vector or the degree. But there are two member fields that need some additional explanation. The first is the *mSignature* variable. It is the signature created by the method *generateSignature* from the same class. To sign the encoded packet the coefficient vector, the data block, and the degree are used to create the message digest. Afterwards the digest is signed and the signature is stored in *mSignature*. Another member field is the *mEncryptedCoefficientVector*. When the file is multicasted and therefore encrypted the encrypted coefficient vector must be stored in a different member field since the result of the encryption done by the cipher is a byte array. The plain coefficient vector must be reset before the encoded packet is send as it otherwise is not an encrypted encoded packet anymore.

DataInformation and EncryptedDataInformation

Like described in section 5.4.2 two different data information classes are needed. One that is not secured in any way and another that is encrypted using a password. The *DataInformation* class is the class for the plain data information. It holds all the information mentioned in 5.4.2. One difference to the design is that the secret session key for decryption of the encoded packets is included in this class. If the data information is not encrypted, making the secret key unnecessary, the member field is left out. Otherwise the secret key is set.

The encrypted data information is represented by the class *EncryptedDataInformation*. It has three member fields. *mSessionname* and *mDataSize* are necessary as the receiver must get some information about the file being send. The third member field *mEncrypted-DataInfo* holds a password encrypted instance of the *DataInformation* class which in this case does hold an secret key, set and generated in the *run* method of the *SenderTab* class. The *DataInformation* object is encrypted in the *run* method as well.

6.2 Receiver

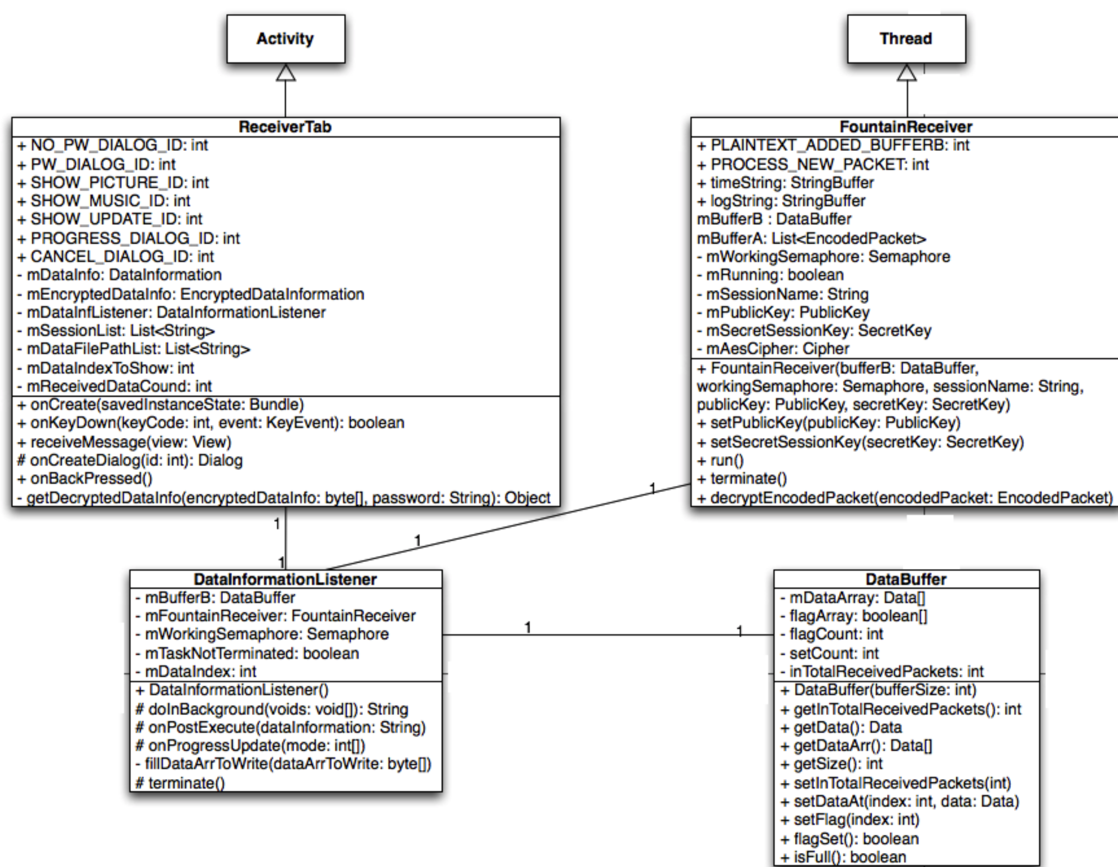


Figure 6.2: UML diagram of the receiver.

In section 5.5 the receiver was designed to be made up of three classes. One Receiver class, one Data Information Listener class, and one Fountain Decoder class. That is the way the receiver is implemented in this application.

6.2.1 ReceiverTab

The receiver's Activity is the counterpart of the sender's Activity and is named *ReceiverTab*. It is the first object instantiated when the receiver tab is selected. Just like the *SenderTab* the *ReceiverTab* overrides the *onCreateDialog*. There are seven dialogs that can be created within the *ReceiverTab*.

1. **No password dialog:** Shows the file information extracted from the plain data information packet. Starts the Fountain Decoder when the file is accepted.
2. **Password dialog:** Shows the session name and file size and prompts the user for a password. Starts the *no password dialog* when the correct password was inserted. Takes the encrypted data information packet and decrypts it with the *getDecryptedDataInfo* method from the same class. If the password is incorrect the same dialog is shown again. On abort the dialog is removed and the receive process is aborted.
3. **Show picture dialog:** Shows the received picture in a preview and the storage path on the smartphone. Closed with the close button
4. **Show music dialog:** Plays the received music file in a preview and shows the storage path on the smartphone. Closed with the close button
5. **Show update dialog:** Shows a dialog that asks the user if he really wants to install/update the application. If accepted, the application is installed. If declined the dialog is removed.
6. **Progress dialog:** Shows a progress dialog that ensures the user that receiving and decoding is still in progress.
7. **Cancel dialog:** Is created when the user presses the back button while the file is being received and decoded. Clicking the *Yes* button aborts the Fountain Decoding and the *No* button removes the cancel dialog and the process dialog is shown again.

The important method in the *ReceiverTab* class is the *receiveMessage* method. It is called when the *Listen for Data/Stop* button is clicked by the user. When the method is called the Data Information Listener is started if the button was set to *Listen for Data* or the Data Information Listener is aborted if the button was set to *Stop*. The method switches the button from *Listen for Data* to *Stop* and vice versa.

6.2.2 DataInformationListener

The class *DataInformationListener* is implemented as an inner class of the *ReceiverTab*. It had to be implemented as an inner class since the *DataInformationListener* must update the user interface of the application. And the user interface can only be updated if the object has access to the *Activity* object of the *ReceiverTab* and the *onCreateDialog* method. Furthermore the *ReceiverTab's Context* must be accessed to acquire a multicast lock which enables the receiver to receive multicast packets. Otherwise they are filtered and dropped. Another feature that makes the *DataInformationListener* spacial is the fact that it does not extends the class *Thread* but an Android specific class named *AsyncTask*.

AsyncTask

The *AsyncTask* class is a generic class that is able to perform background operations like a normal thread [13]. On top of that the *AsyncTask* is able to change the user interface of the application. An *AsyncTask* works a little different from a normal thread. Unlike threads, it is started by calling the *execute* method and not the *start* method. An *AsyncTask* has four methods than can be called and worked with. The *onPreExecute*, *doInBackground*, *onProgessUpdate*, and the *onPostExecute* method. The *onPreExecute* method is not used in this application as it is mostly used to setup a task by showing a progress bar which is not needed. When the task is started the first method that is called is *doInBackground*. In this method all necessary computation is done. In case of this application, the listening and processing of the data information packets. After analyzing the data information the *No password dialog* or the *Password dialog* is created depending on the information packet. Creating a dialog can be achieved by calling the *publishProgess* method which in turn calls the *onProgressUpdate* method. In that method the dialog is created by calling the *showDialog* method. Figure 5.9 shows the procedure of the *DataInformationListener*.
After the *doInBackground* method is completed *onPostExecute* is called automatically. In *onPostExecute* the user interface of the receiver is extended with the view that shows information about the received file and offers a button to open the file in a preview, like shown in Figure 5.8. But the view can only be updated if the reception of the file was completed. That is why the Fountain Decoder acquires and locks a binary semaphore which the *doInBackground* method blocks on. When the file is received completely, the semaphore is released and *doInBackground* can finish.

6.2.3 FountainReceiver

The Fountain Decoder class is named *FountainReceiver*. Like the *FountainSender*, the *FountainReceiver* extends the Thread class to run concurrently to the *ReceiverTab*. The principle

of operation of the *run* method is shown in Figures 5.10, 5.11, and 5.12. According to the *FountainSender* the receiver verifies the signature by generating a message digest with the *SHA-1* and verifying the hash value with the *DSA* algorithm. The decryption of the encoded packet is achieved by calling the method *decryptEncryptedPacket*. A cipher working in the *AES/ECB/PKCS5Paddin* mode is initialized with the secure key extracted from the decrypted *DataInformation* packet.

7 Test and Evaluation

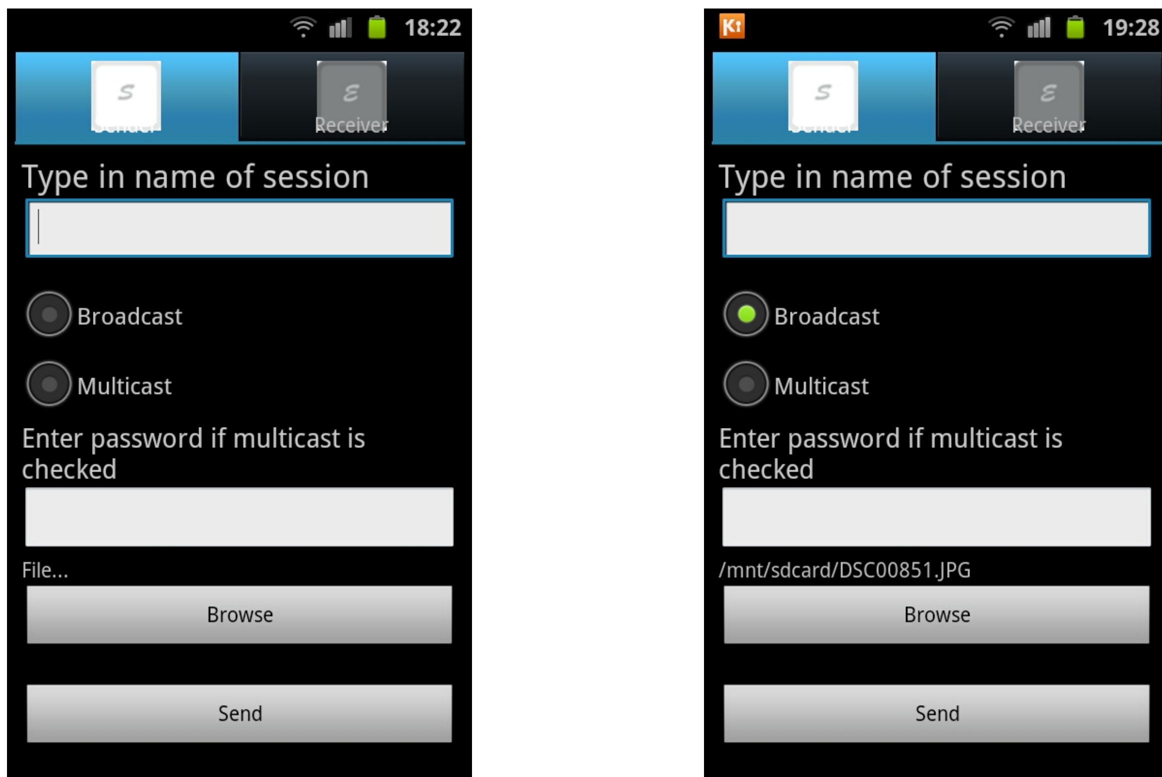
7.1 Test

After the application was designed and implemented, it is now time to test and evaluate the application. The scenarios from the analysis in chapter 3 will be applied to see if the requirements are fulfilled and that the application works properly. The hardware used to test the application is a *Samsung Galaxy S2* smartphone running the Android OS and a *MacBook Pro*. The MacBook can be used for testing since the application was implemented in Java. Therefore the implemented *FountainSender* and *FountainReceiver* classes can be reused in a separate Java application to send files to the smartphone and receive files sent from the smartphone.

7.1.1 Sending files

First we will test the application in regard to the send process.
There are two possibilities to send a file. The first possibility is to send the file by broadcasting it to every user in the same network. The second possibility is to choose multicast and encrypt the file using a shared secret session key which the sender generates and sends along with the data information. In the following test scenarios the MacBook will serve as the receiver listening for files.

In Figure 7.1 picture *a* the initial user interface of the sender tab is shown. At the top the user can enter a session name. Under the session name text field two radio buttons are available for *Broadcast* or *Multicast* selection. Only one option can be chosen at a time. Selecting one option deselects the other option automatically. The radio buttons are followed by a text field for the password which encrypts the *EncryptedDataInformation* (See 6.1.3), a button to browse through the smartphone, and a button to start encoding and sending the file. The *Send* button is deactivated at first. It only is clickable if a file was chosen.



(a) Initial UI for the sender tab.

(b) UI after broadcast selection.

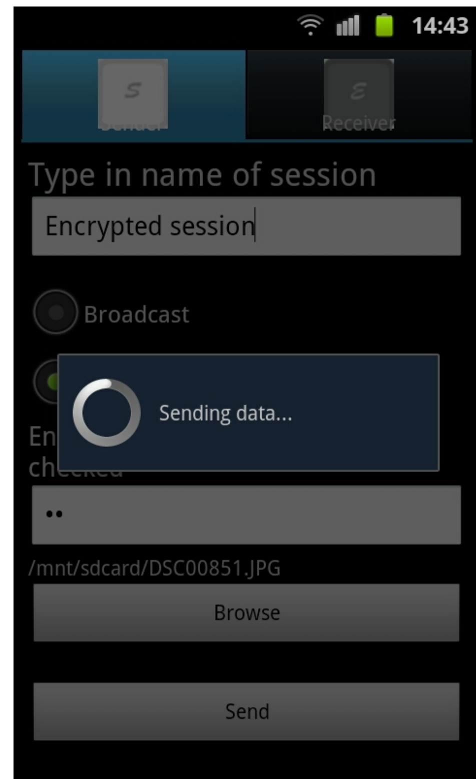
Figure 7.1: Sender tab configuration UI.

Broadcast

If the user wants to broadcast a file to everybody in the network, he selects the *Broadcast* radio button like depicted in Figure 7.1 picture *b*. Like described in the *Design* section, the user does not need to enter a session name when the file is broadcasted. If no session name is entered, the filename is chosen instead. Now the user must choose a file which he wants to send. Clicking the *Browse* button opens a browse dialog which is shown in Figure 7.2 picture *a*. The dialog lists all the files that the current folder stores. The folders are not filtered but the files are. One of the non-functional requirements of the sender is to filter out files that are not supported by the application (See 3.2.1). That is why only files with *.jpg*(picture), *.png*(picture), *.mp3*(music), *.m4a*(music), or *.apk*(Android application) suffixes are listed in the dialog. When a file is chosen, the path to the file is shown in the user interface above the *Browse* button (Figure 7.1 picture *b*). Furthermore the *Send* button is set to be clickable since all necessary user inputs where identified. Figure 7.2 picture *b* displays the dialog shown to the user when the *Send* button is clicked. The dialog ensures the user that the application is currently distributing the file. In Figure 7.3 picture *b* you can see the

information shown on the MacBook application. As already mentioned, the MacBook is the receiver in this test scenario. The session name shows that the file chosen in Figure 7.1 picture *b*, which is currently being sent by the sender, is the one the MacBook is receiving. After the user accepts the file by entering *Y*, the receiving process is started and after a while completed.

(a) Browse dialog.



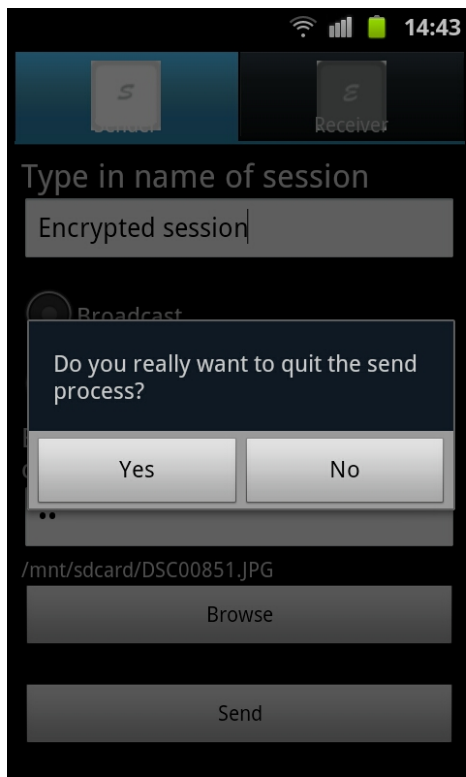
(b) Progress dialog showing that application is sending.

Figure 7.2: Screenshot of the browse and progress dialog.

If the user wants to abort the send process because every recipient received the file successfully or he wants to change the configuration, the *Back* button can be pressed. Every Android handset offers such a *Back* button. A dialog asks the user if he really wants to abort the process or not (Figure 7.3 picture *a*).

Conclusion

The test scenario for broadcasting shows that the application sends the file successfully to the receiver. The application offers the option to select *Broadcast* so that every recipient is



(a) Abort dialog.

```
------Listening for data information------
Data information caught!
Session name:-=DSC00851.JPG=- Size: 4257629
Do you want to accept? -> Y | N:
Y
File: DSC00851.JPG successfully received!
794 packets in total received!
```

(b) Receiver on MacBook receives data information from sender.

Figure 7.3: Screenshot of abort dialog and receive dialog from MacBook receiver.

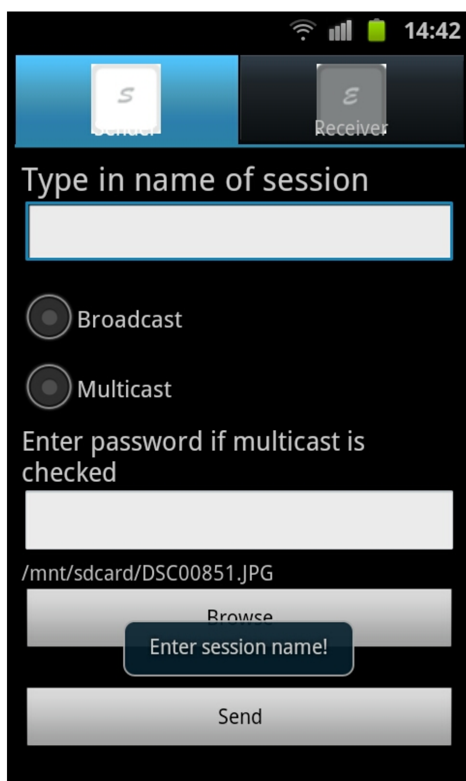
able to receive the file. A session name can be entered and the smartphone can be browsed using the browse dialog. The *Send* button starts the send process and pressing the *Back* button, while sending is in progress, displays a dialog that offers the user to abort the process or to keep sending the file.

Multicast

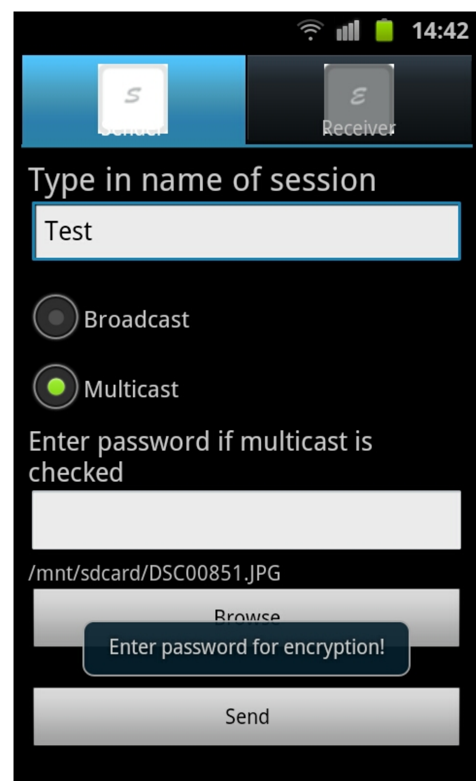
After the broadcast scenario was successfully tested is it time to test the multicast scenario. The user must be able to encrypt the file using a password and only recipients who know the password must be able to receive the file.

The initial user interface is the same. The only difference is that the *Multicast* option is selected instead of the *Broadcast* option (Figure 7.1). There is no difference in the way the user browses through his smartphone. Clicking the *Browse* button, opens the browse dialog (Figure 7.2 picture *a*). The difference to the broadcast scenario is shown after a file is

selected. In contrast to the broadcast scenario the user cannot just start the send process after a file was selected. When the *Multicast* option is chosen, the user must enter a session name and a password. Otherwise the send process is not started. The session name is necessary because in this case the data information is encrypted. But the receiver needs some information about the session to link the session to the sender. Taking the filename would reveal too much about the file. That is why a session name is needed. A password is needed to encrypt the data information as explained in section 5.3.2. Entering no password, would not make any sense. Therefore, if no session name and/or password was entered, the user is informed by the application. Figure 7.4 picture *a* shows a *Toast* notification that pops up if no session name was entered. Figure 7.4 picture *b* shows the *Toast* notification informing the user that no password was entered.
A *Toast* notification is a message that is only displayed for a limited time.



(a) Toast message hinting that session name is missing.



(b) Toast message hinting that password is missing.

Figure 7.4: Toast messages informing the user that information is missing.

After the user enters all necessary information into the application and chooses a file, the send process can start. Figure 7.5 picture *a* shows that the recipient receives an encrypted data information packet. The information was encrypted using the password *password*. You

```
------Listening for data information------
Encrypted data information caught!
Session name:-=DSC00851.JPG=- Size: 4,257629 MB
Insert password:
wrong password
Password incorrect!
```

(a) Wrong password entered.

```
------Listening for data information------
Encrypted data information caught!
Session name:-=Test=- Size: 4,257629 MB
Insert password:
password
Password correct! Receiving and decoding started...
File: DSC00851.JPG successfully received!
845 packets in total received!
```

(b) Correct password entered.

Figure 7.5: MacBook receives encrypted file.

can see that the password *wrong password* was entered on the receiver's side. The application informs the user that the password was incorrect. Figure 7.5 picture *b* depicts that the user enters the right password. Thus the reception and decoding is started until the complete file is received. You can also see that at first only the session name and the file size can be extracted from the encrypted data information.

Conclusion

The test scenario shows that the *Multicast* option works properly. The user is notified by a *Toast* notification if the session name or the password are missing. Furthermore, the test scenario shows, that the file can only be received by recipients who know the password. In case a wrong password in entered, the file cannot be received as the secret session key that is included in the data information cannot be extracted. But if the right password is entered, the file is received successfully.

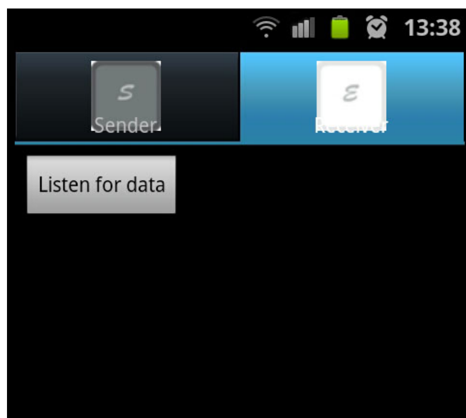
7.1.2 Receiving files

Now that we know that files can be successfully distributed using *Broadcast* or *Multicast*, it is time to test the receiving part of the application.

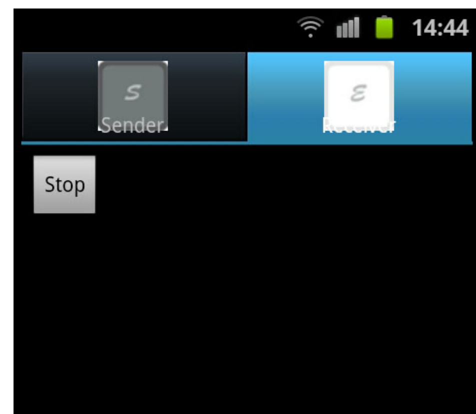
Session not encrypted

In this test scenario the file is being distributed without any encryption. Every user in the network can receive the file.

Figure 7.6 picture *a* depicts the initial user interface of the ReceiverTab. By pressing the *Listen for data* button the receiver starts listening for files. The button changes to *Stop* as is shown in Figure 7.6 picture *b*. If the user wants to stop listening the *Stop* button can be pressed.



(a) Initial UI of ReceiverTab.



(b) Receiver listening for data information.

Figure 7.6: Initial ReceiverTab user interface and UI when receiver is listening for data information.

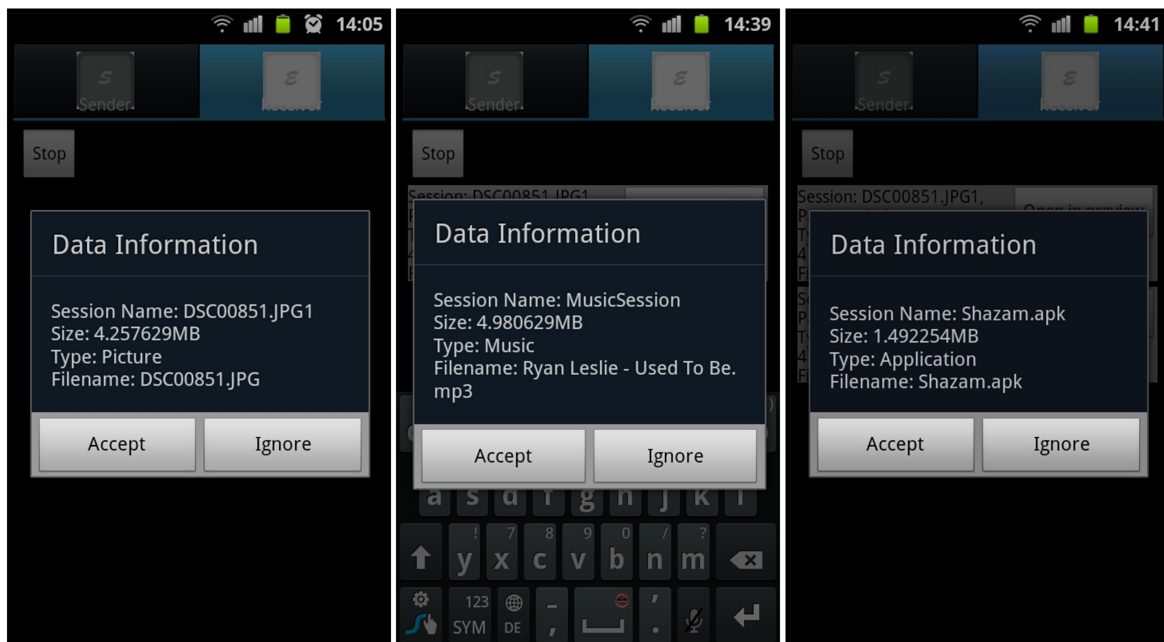
If a data information is received, a dialog pops up that displays the information about the file. You can see in Figure 7.7 that the dialog shows the session name, the size of the file, the file type, and the file name. The user can accept and thus start the receive and Fountain decode process by pressing the *Accept* button or decline the file by pressing *Ignore*. In case the user wants to ignore the file, the dialog is removed and the user is never informed about that file again until the application is restarted.

Accepting the file, opens the receive progress dialog which indicates that the receive process is in progress (See figure 7.8 picture *a*). The reception can be quit by pressing the *Back* button. The quit dialog asks the user if he really wants to quit (See figure 7.8 picture *b*). Figure 7.8 picture *c* shows the user interface after a picture, a music track, and an application were successfully received. You can see that the information about the file is displayed along with a button that can be pressed to open the received file in a preview or to install the received application.

Figure 7.9 depicts the individual dialogs that are opened depending on the file type. A picture or a music file are opened in a preview showing or playing the received file. In case of an application, a dialog is shown asking the user if he really wants to install the application.

Session encrypted

In case the session in encrypted, the application displays a dialog that informs the user that a password is needed to decrypt the data information. Figure 7.10 depicts the dialog shown



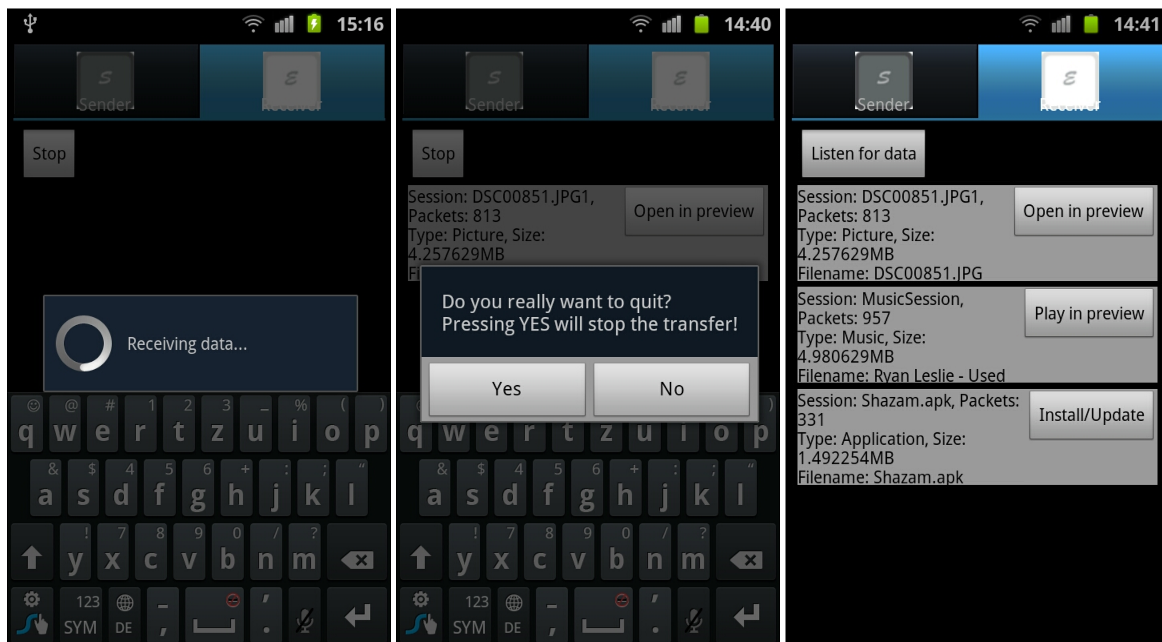
(a) Data information for picture. (b) Data information for music. (c) Data information for application.

Figure 7.7: Data information dialogs for supported file types.

to the user. If the user enters an incorrect password the dialog is shown again. If a correct password was entered an accept dialog is shown like in figure 7.7.

Conclusion

The tests show that the application is able to receive broadcasted and multicasted files and assemble them correctly. The user is informed about the file that is about to be received through a dialog. If the file is being multicasted, the user is prompted with a password dialog. The user can accept or decline the file. The UI is updated so that the user can open the received files. The pictures and music tracks can be opened in a preview and the applications can be installed while being in the application.



(a) Receive progress dialog.

(b) Dialog asking user if he really wants to quit.

(c) Changed UI after reception of several files.

Figure 7.8: Receive progress dialog, abort dialog, and changed UI after reception of files.

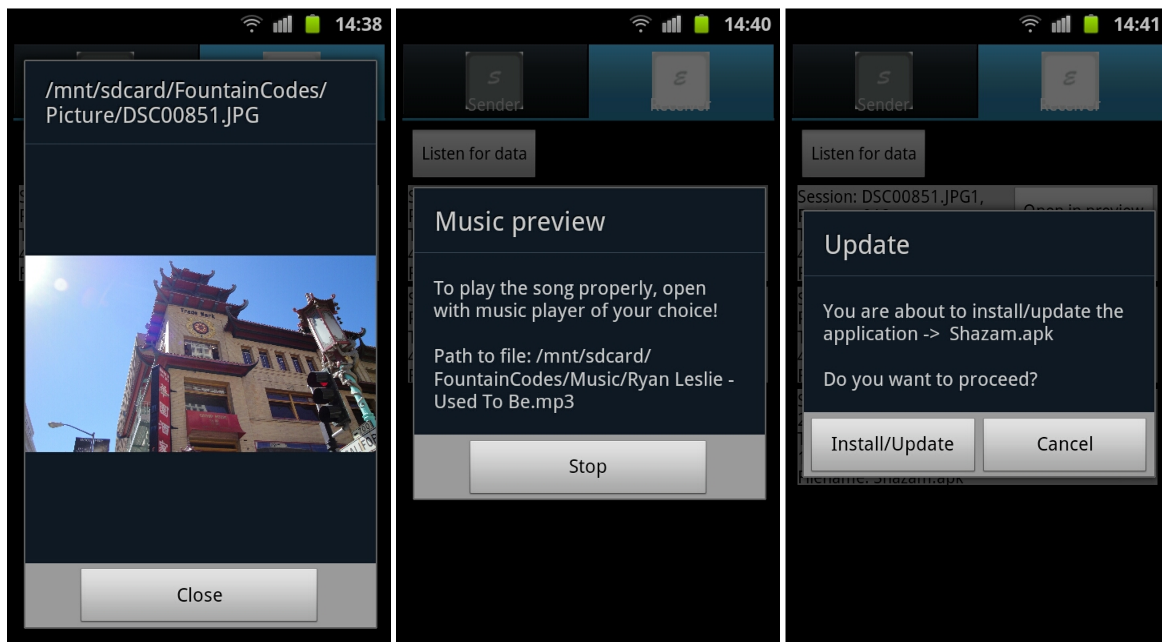
7.2 Evaluation

7.2.1 Fountain Codes

This section of the *Test and Evaluation* chapter is about evaluating the implemented Fountain Codes. The objective of the evaluation is to find out how efficient the Fountain Codes work in this application. To evaluate the Fountain Codes a picture of about 4.257 MB is sent. As each data block has a size of 1000 Bytes (See 5.4.2), the file is split into 4257 data blocks.

The efficiency of the Fountain Codes is influenced by the *degree distribution* like explained in chapter 2. For the initial implementation of the application the *degree distribution* from [6] was chosen. Table 7.1 shows the *degree distribution*.

Implementing the application using the *degree distribution* from Table 7.1, results in the receiver needing almost twice the amount of data blocks to decode the file completely. Such a result, of course, is not acceptable. Changing the *degree distribution* does not change the efficiency of the application. That is why there has to be another reason for the miserable results.



(a) Received picture opened in preview.

(b) Received music track played in preview.

(c) Dialog asking user to proceed with installation or cancel.

Figure 7.9: Individual dialogs shown depending on file type.

Degree	p(d)
1	0.19
2	0.34
4	0.27
8	0.10
16	0.5
32	0.3
64	0.2

Table 7.1: Initially implemented degree distribution.

The reason is the time the receiver needs to compute the received encoded packet. When an encoded packet is received the decoder traverses through the coefficient vector to see if there are decoded data blocks which could help to decode the current encoded packet. And that takes some time. It takes the sender much less than a millisecond to send a new encoded packet if the packets are sent in a simple while loop. Therefore, the receiver misses a lot of encoded packets. The special characteristic of Fountain Codes certainly is the fact that packets can be missed and the recipient decodes the complete data very fast nevertheless. But the assumption with Fountain Codes is that the majority of the encoded

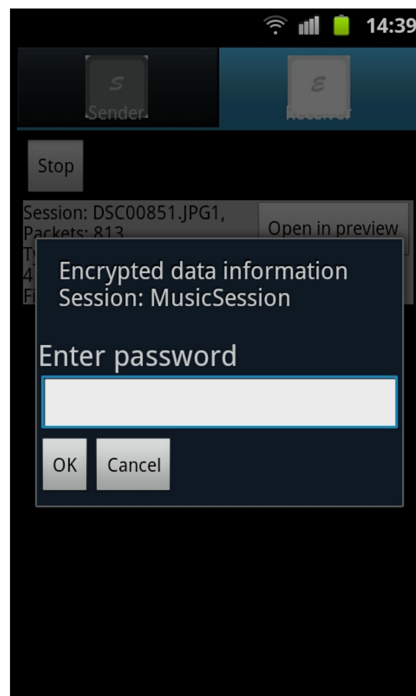


Figure 7.10: Dialog promting user for password.

packets do reach the recipient. That means if too many following packets get lost, the actual *degree distribution* is not applied anymore. That is why the sender must wait for a short period of time between each send process to assure that the receiver is ready to receive the next packet. But how long does the sender have to wait? To find out a small addition is implemented into the application that measures the time the receiver needs to decode one encoded packet.

Table 7.2 shows a snippet of a time listing which was created by the application. The time listing continues in almost the same pattern. There are not many discrepancies. You can see that the receiver mostly needs less than 25 milliseconds to process one encoded packet.

Keeping the result in mind, the implementation of the *FountainSender* is changed. Between each send process the thread waits for 25 milliseconds. With the *FountainSender* changed, the Fountain Codes work much better. You can see in table 7.3 that the *FountainReceiver* now only needs about 60 % more encoded packets to decode the complete file.

60% is already a big improvement to the previous 90-100% off additional encoded packets needed to decode the complete data. But it still is not satisfying enough. Now that the timing issue is fixed, a better performance can probably be obtained by finding a better *degree distribution*. That is why the initial *degree distributions* has to be changed and tested.

Packet Nr	Time in ms	Packet Nr	Time in ms	Packet Nr	Time in ms
1	17	8	11	15	23
2	11	9	10	16	97
3	7	10	33	17	7
4	7	11	19	18	10
5	1	12	35	19	21
6	13	13	18	20	15
7	36	14	14	21	22

Table 7.2: Time listing of receiver.

Packets in total	Additional rate	Packets in total	Additional rate
6861	61%	6810	60%
7003	65%	6593	61%
6950	63%	6912	62%
6605	55%	6757	59%
6744	58%	6930	63%

Table 7.3: Listing of encoded packets needed to encode the file. File is split into 4257 data blocks.

After a couple of changes, the *degree distribution* in table 7.4 shows the best performance yet as you can see in table 7.5. That is why that *degree distribution* is implemented in the final version of the application.

Degree	p(d)
1	0.23
2	0.27
4	0.19
8	0.12
16	0.8
32	0.6
64	0.5

Table 7.4: Implemented degree distribution

We were able to reduce the additional packets to about 50%. A lot of different combinations of *degree distributions* were tested but we were never able to go under 50%. The question is how to further optimize the Fountain Codes.

As was described in section 5.4.2, the way encoded packets are chosen at random is another

Packets in total	Additional rate	Packets in total	Additional rate
6314	48%	6526	53%
6405	50%	6422	51%
6484	52%	6376	50%
6453	52%	6449	51%
6399	50%	6552	54%

Table 7.5: Listing with new degree distribution from table 7.4.

factor that influences the efficiency of Fountain Codes. The initial implementation was to jump a fixed length with each new encoded packet and then jump, depending on the degree d, $d-1$ times a random length with the *Math.random* method from the standard Java library to the next data block. Figure 7.11 shows the process.

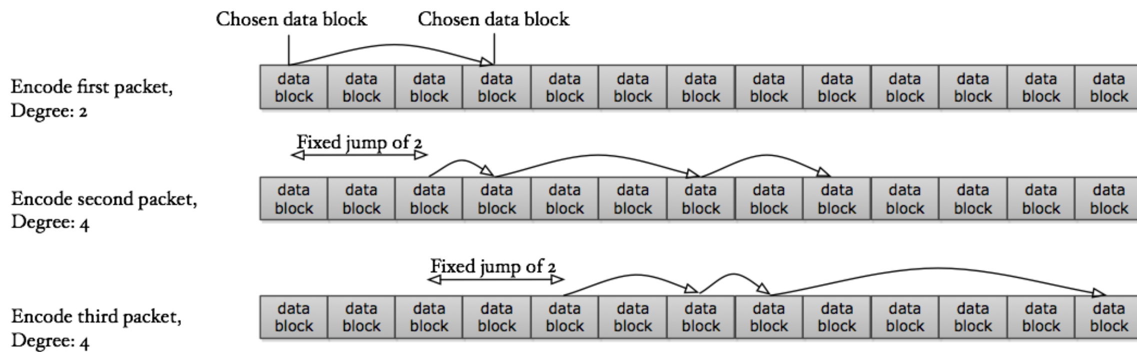


Figure 7.11: Initial scheme to select random data blocks.

There is one little issue with the selecting scheme explained above. No assurance is given that the data blocks overlap. The idea of Fountain Codes is, that encoded symbols are decoded with plain data blocks that were already completely decoded. Meaning, that encoded symbols include data blocks that are included in other encoded symbols as well. The receiver cannot combine three encoded symbols, if the first one includes data blocks (1,3,5), the second (2,6), and the third (4). But if the first one included (1,3,5), the second (3,4), and the third (4), then the decoder would be able to combine the symbols so that (3) and (4) are extracted successfully and the first symbol is reduced to just (1,5). Keeping that in mind the following selection scheme was designed and implemented.

The first index of the following encoded symbol is the second at random chosen index from the previous encoded symbol. That way it is ensured that at least one data block does overlap with a data block of the following code symbol. Figure 7.12 visualizes the scheme.

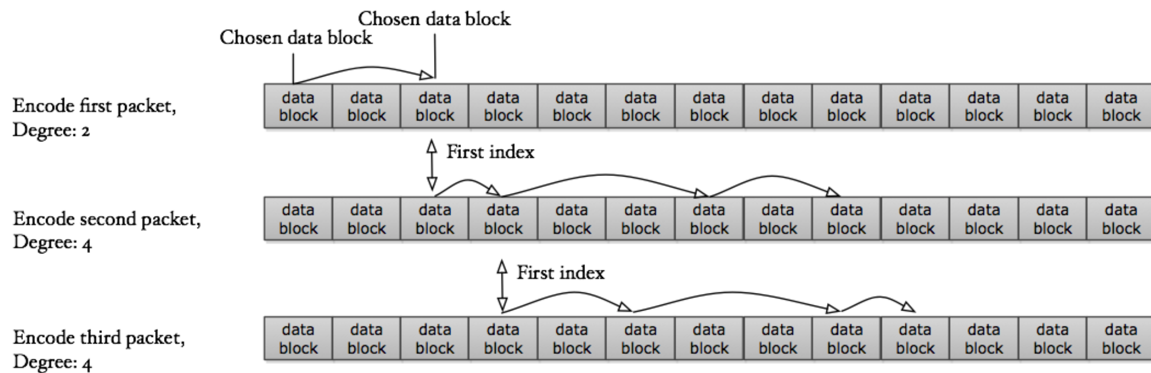


Figure 7.12: Improved scheme to select random data blocks.

With the new selection scheme implemented, the application is tested again. Table 7.6 shows the results of the test.

Packets in total	Additional rate	Packets in total	Additional rate
5879	38%	5795	36%
5745	35%	5866	38%
6018	41%	5722	34%
5763	35%	5822	37%
5946	40%	5888	38%

Table 7.6: Listing with the new degree distribution from table 7.4 and the new selection scheme from figure 7.12.

You can see that using the new selection scheme results in a big drop of additional packets needed, again. Now the receiver needs only 34% more encoded packets to decode the complete file in the best case scenario and about 41% in the worst case. Certainly, there will be cases where the additional packet rate will be higher than that. But on average the results from table 7.6 will be applicable.

Some may think that 35-40% of additional packets are pretty much considering that not a lot of packets will be lost due to the reliable WLAN. But as was already mentioned, the issue with the decoding time could still occur with some received packets. Remember that a delay of 25 milliseconds was implemented into the sender so that the receiver has time to decode the encoded symbol. When you look at table 7.2, you can see that there are four packets that needed more than 25 millisecond of decoding time (Packet nr. 7, 10, 12, 16). That means that just considering these 21 packets almost 20% of the packets are already missed by the receiver. Keeping that in mind the 35-40% do not seem too bad. Furthermore, its worth mentioning that even files bigger than 8 MB and smaller than 2 MB were distributed with the

application. And the additional packet rate was almost round about 35-40% with every file. In conclusion the implemented *degree distribution* and *selection scheme* is efficient enough no matter what file sizes are distributed.

7.2.2 Application performance

Time

This section is about evaluating the performance of the application in a whole. In the previous section we saw that the Fountain Codes work very efficiently. But now we must evaluate the usability of the application in regard to the overall computing time. How long does it take to distribute one file?

In section 5.4.2 the application was designed to split the file into 1000 Byte blocks. When the file from the test scenarios is send, it will be split into 4257 data blocks. Distributing the file now takes the application 2-3 minutes to receive the whole file. Waiting 2-3 minutes for one picture is unacceptable from the user's point of view. The design choice to use 1000 Byte blocks was made since we wanted to cope with the possible noise on the transmission channel. But first of all, the current test environment does not have such a noisy channel and second of all, looking at the test results it probably is faster to split the file into bigger data blocks and rather let the Ethernet protocol do the retransmission then split the file into small blocks and let the application do the computation.

Choosing a data block size of 7000 Bytes and thus splitting the test file into 568 data blocks, results in a distribution time of only 20 seconds. Waiting 20 seconds is much more acceptable than 2-3 minutes.

File size

Another performance limitation that should be mentioned is the supported maximal file size. The current realization of the application only allows files that are smaller than 50 MB to be distributed. Choosing files that are bigger than that results in an *Out of memory error* exception since the heap space of the virtual machine is overloaded. But the application was designed to distribute photos, music tracks, and applications. And such files seldom are bigger than 10-15 MB.

8 Summary and Future Work

8.1 Summary

The goal of this bachelor thesis was to design and implement an application that allows users of Android Smartphones to distribute files to multiple recipients simultaneously. The foundation of the application are the Fountain Codes. They are beneficial in cases where the data is shared over an unreliable medium or a channel that is very noisy. Due to the characteristic of the share medium, there is a higher probability that data is lost along the way to the recipient. To compensate the data loss the data is Fountain encoded. At random chosen data blocks are XORed together to make up an encoded packet that holds multiple information in one packet. Each recipient can extract the information he needs individually. Even if the recipients miss different packets, one encoded packet could hold information for all of them.

When developing an application, the first thing to do is to find out what the application should be able to do and where it could be used. That is why we first started to analyze the application. The goal of the analysis was to find requirements so that we could design the application and subsequently implement it. With the help of three scenarios we were able to extract the requirements. The first scenario helped us get the basic requirements for the application like the possibility to browse through the phone or the need of an information packet that lets the recipient know what kind of file is being shared. The second scenario was about a user wanting to share the file with only a subset of the connected users. It helped us realize that some sort of encryption is needed to ensure that only targeted users are able to get the file. Furthermore, the need of a key agreement was extracted. The last scenario was about security holes that must be considered. It pointed out that a mechanism was required that ensured the data's integrity and the assurance that only genuine members take part in the key agreement.

The next step was to design the application with the requirements extracted from the analysis chapter. We first looked at the application as a whole. By analyzing the processes the sender and the receiver have to go through, we were able to find the most important classes the application should implement. We elaborated that the Fountain Codes should be swapped to an external class to prepare the application for future development. Furthermore, we saw

that some resources should only be allocated when really needed. Afterwards we focused on the three main components of the application. The sender, the receiver, and the security measures both use to ensure flawless file distribution. The work on the sender and receiver was focused on their user interfaces and the steps each one has to take to send files or receive them. With the help of flowcharts the procedures were visualized. While designing the security measures we noticed that agreeing on a key for encryption was not the optimal way for this specific application. Too many information had to be exchanged between the participants to make the agreement possible. And that was a problem. So we decided to let the sender generate a secret session key and let him spread the key by encrypting it with a password. Only members who were in possession of the password were able to receive the file.

The last step of the bachelor thesis was to test and evaluate the application. Especially the performance of the Fountain Codes were an important part of the evaluation.
First the application was tested to see if all requirements were implemented correctly. The sender was able to send files by broadcasting them or multicasting them to a subset of users who were in possession of the password. Testing the receiver, showed us that the reception of files was successful. Several dialogs offered the opportunity to interact with the application. If the file was meant for only a subset, and thus was encrypted, the user was prompted with a password dialog. Inserting the right password resulted in extraction of the secret session key and the decryption of the encoded packets. After the successful reception, the user was able to open the files in a preview or install the application that was received.

8.2 Future Work

When the Fountain Codes were evaluated in the preceding section, we saw that there are a lot of aspects that influence the performance of the Fountain Codes. First we found out that the sender had to delay his sending cycles since too many packets were missed on the receiver's side. The receiver was still decoding a packet and thus was not able to listen for the next packet. Delaying the sender, resulted in the Fountain Codes working much better. But not good enough. That is why the degree distribution and the selection scheme were optimized. After editing the aforementioned aspects, we were able to optimize the Fountain Codes to the point that only 35-40% additional packets were needed to completely decode the file.

35-40% additional packets are not bad considering that some packets do get lost due to the decoding time on the receiver's side. But certainly the Fountain Codes and the application can be further optimized.
First of all there are a lot of different ways to realize the degree distribution. In this bachelor thesis we took the findings of [6] as the basis for the degree distribution and adapted

them. But going a different way could be and probably is more efficient. Finding a degree distribution that works efficiently with all data sizes, is an interesting topic for future works. Furthermore, an efficient selection scheme must be found that works well together with the degree distribution. The goal is to find a degree distribution and a suitable selection scheme with which only 10-20% of additional packets are needed to decode the data.
Future work in regard to the application's performance could be to optimize the decoding process. As was already mentioned, decoding one packet takes round about 10-30 milliseconds. Optimizing the decoding algorithm, could lead to shorter decoding times and thus to a faster distribution of files. Another enhancement could be the file size. Right now only files that are smaller than 50 MB are allowed. It would be beneficial if the application could efficiently share bigger videos that were recorded with the smartphone.

Bibliography

[1] M. Luby, "Information additive code generator and decoder for communication systems," US Patent No. 6,373,406, April 16 2002.

[2] D. Westhoff, J. Girao, and A. Sarma, "Security solutions for wireless sensor networks."

[3] [Online]. Available: http://www.netzwelt.de/news/81417-audi-a8-navigation-google-earth.html

[4] S. Conder and L. Darcey, *Android Wireless Application Development*, 2nd ed. Addison-Wesley, 2011.

[5] C. Paar and J. Pelzl, *Understanding Cryptography*. Springer, 2010.

[6] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A. F. H. III, and M. Zorzi, "Synapse: A network reprogramming protocol for wireless sensor networks using fountain codes."

[7] [Online]. Available: http://www.comscore.com/ger/Press_Events/Press_Releases/2011/1/Google_Android_Shows_Fastest_Growth_Among_Smartphone_Platforms_in_Germany

[8] H. Perrey, "Untersuchung der vertraulichkeit nach obfuskierung durch fountain codes," Master's thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2010.

[9] J.-M. Bohli, A. Hessler, O. Ugus, and D. Westhoff, "Security enhanced multi-hop over the air reprogramming with fountain codes."

[10] M. Luby, "Lt codes."

[11] C. Harrelson, L. Ip, and W. Wing, "Limited randomness lt codes."

[12] [Online]. Available: http://www.gartner.com/it/page.jsp?id=1764714

[13] [Online]. Available: http://developer.android.com/reference/android/os/AsyncTask.html

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, November 2, 2011	
Ort, Datum	Unterschrift