UiO : Department of Informatics
University of Oslo

Toolset for NFC tag development

Design, implementation and evaluation

Aage Dahl
Master's Thesis Spring 2014

Toolset for NFC tag development

Aage Dahl

2nd May 2014

Abstract

NFC tags follow a range of standards and communication protocols. Support for communicating with these tags is limited, and few empirical studies investigate their performance. Therefore it is challenging to develop tags for specific user scenarios.

This master thesis aims at designing, developing and implementing two apps and one library, which are supposed to reduce the effort of testing NFC tags. The library enables seamless communication with a range of different tags, through a singular interface. The library functions as a tool for the two apps that are developed in the course of this work. The first app enables tag developers to assess their proprietary communication protocols, whereas the second app offers flexible automated benchmarking procedures that can ultimately be used to compare tags.

To confirm that the apps fulfill their purpose, extensive testing is performed to compare a set of five tags based on five different standards in order to assist in the development of one proprietary tag. The results of this study indicate that there is no ultimate standard that perfectly fits any scenario and that the selection of the standard both restricts and enables design choices.

The present research reveals that the communication performance between the NFC reader and the NFC tag is not significantly influenced by the increase in distance between them from zero to 30mm, not even when a body of saline solution separates them. My research also provides some unexpected results; the measured energy consumption for powering the tag is about 30% higher than that for writing to the tag. Hence, the energy consumption seems to decrease when the reader is communicating with the tag, as opposed to when the reader is merely powering the tag.

Preface

Prior to writing this thesis, I was not distinctly familiar with the term NFC and had little knowledge of RFID and how it worked. However with Ali Zaher's introduction of the NFC based tag that could potentially impact the lives of people suffering from diabetes, it became clear that a thesis exploring NFC technology is both needed and desired.

I would like to thank my primary supervisor, Thomas Plagemann for both introducing me to this subject and for guiding me through the development of my thesis. And especially for providing the input I needed when I needed it, which has truly been essential for the thesis to give results. It has helped me to focus on the main issues at hand and to make the right decisions at the critical points in time.

I would also like to thank my second supervisor, Ali Zaher, who has helped me throughout this thesis with his relentless understanding and lengthy discussions. I could not have asked for better guidance.

To my family, I would like to thank them for being patient and having faith in me. It has made it much easier to tackle this task.

And last but not least, I would like to thank my dear wife for believing in me and providing me with support, love and understanding. There would simply be no thesis without her.

Contents

List of Figures

List of Tables

Listings

Abbreviations

The abbreviations used in this document are presented in Table 1.

Abbreviation	Description
IT	Information Technology
NFC	Near Field Communication
IC	Integrated Circuit
I/O	Input/Output
EEPROM	Electrically Erasable Programmable Read-Only Memory
RFID	Radio Frequency Identification
SSP	Bluetooth Secure Simple Pairing
GPS	Global Positioning System
RDT	Record Type Definition
NDEF	NFC Data Exchange Format
OSI	Open Systems Interconnection
LLC	Logical Link Control
MAC	Medium Access Control
bps	Bits Per Second
BPS	Bytes Per Second
UI	User Interface
UUID	Universally Unique Identification
3s.f.	Three significant figures
S.D.	Standard deviation
RF	Radio frequency
CRC	Cyclic redundancy check

Table 1: Abbreviations

Part I

Introduction

Chapter 1

Introduction

Information technology has exploded during the last few decades and what started out as being a tool for IT professionals has quickly become a source of information that is readily accessible even by the youngest members of our society. My very own 12 months old nephew is now a proud user of the iPad and has through apps developed for this platform acquired the ability to recite the English alphabet in the form of a song. As Mark Weiser advice:

> "Machines that fit the human environment, instead of forcing humans to enter theirs, will make using computers as refreshing as taking a walk in the woods." [62]

This advice is taken seriously by the development community [27, 33] and it leads to consumers completely surrendering to the automation and ease supplied with the power of computing and creativity of developers. For example, to take a picture and render it available for millions of users enjoyment can be accomplished with a few clicks of a button through mobile apps such as Instagram [28]. Simply the process of rendering this picture would 15 years ago easily take a week for the amateur photographer, as this process would have to be outsourced to a professional printing service. This magnificent improvement is reflected in vast other areas as well, technologies such as content analytics is trying to close the gap between natural language and structured information, allowing computers to make sense of our tweets and Facebook comments as well as mimicking humans understanding of visual and audio contents [2, 39]. Artificial intelligence is yet another area that tries to give machines the ability to reason and learn like humans. A common theme amongst these topics is that they strive to populate the virtual world with a virtualized version of the real world. A final topic that strives for the same result is ubiquitous computing where computing appears to be everywhere and operating seamlessly under our noses to understand and fulfill our needs and desires without our conscious interaction with the devices.

The gap between the real world and the virtual world is growing closer by every new technology and software solution made available on the market, and one of these technologies is *Near Field Communication* (NFC). NFC makes the virtual world reachable by the gesture of a touch whilst still

retaining a high level of security. By moving two NFC enabled devices into close proximity of each other, the two can exchange information. This is done without the need of pressing a single button.

Imagine placing your phone over the palm of your hand and being able to see what goes on inside your body. Allowing information from integrated sensors to flow to a transceiver in the palm of your hand, and from there being able to transmit that information to powerful software solutions made available with your mobile phone. It is a future that is well within our reach through the use of existing technologies such as NFC.

The focus of this master thesis is closely aligned with the description in the previous paragraph. Through the use of implanted sensors connected to implanted *integrated circuits* (ICs) with an NFC interface, vital signs can be monitored with the simple gesture of touching the phone to the location of the IC located underneath the skin. An illustration of this is depicted in Figure 1.1. This technology enables the transfer of both power and information through a single wireless interface, potentially allowing sensors and circuits to run without the need of separate implanted power supplies such as batteries. For patients with chronic illnesses such as diabetes type 1, this system could perform continuous monitoring of vital signs, allowing information to be collected and processed with minimal invasion into the patient's daily activities. The processing could further be used for early warning systems to alert both the patient and the medical staff of upcoming dangers and thereby prevent or reduce the impact of upcoming disasters.

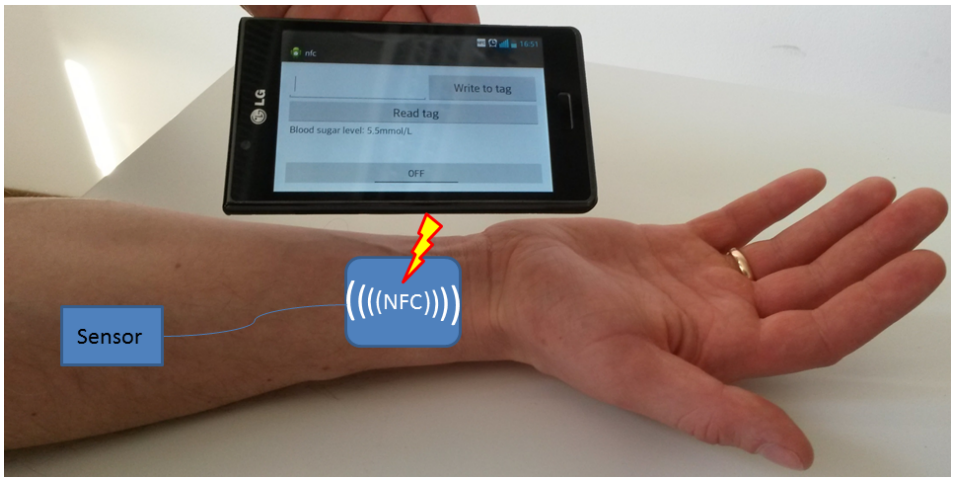


Figure 1.1: Micro implant connected to a sensor with a smartphone as the reader

1.1 Problem statement

NFC is a term used to refer to a collection of short-range wireless technologies for exchanging data between devices. A subset of these devices is passive NFC tags (see 2.3.1.2) that are used for a range of

different applications. Different protocols operate amongst these tags and result in different characteristics and behavior, however little research has been conducted that informs us about their performance. There exist studies of how NFC technologies generally perform [34]. There even exist studies that prove that NFC tags are able to communicate through biological tissue[23]. However, no published studies exist that compare the performance achieved by these different tag types. The performance is in this context related to battery consumption, bit error rates, communication range and communication throughput. Information about these metrics can assist developers in choosing the right tag type for their specific use-case scenario.

Technical specifications for each individual tag exist, but there are no empirical studies that show how these relate to real life scenarios. When developing tags that are used for specific scenarios such as those mentioned in the introduction section of this chapter, how the tags perform can be imperative to the tag type selection. Some tag protocols have large overheads related to data transfers and might therefore be inefficient when repeatedly transferring small samples of data. Other tag protocols have small overheads that make transfers of small data efficient at the cost of making transfers of large data less efficient. Furthermore, how these I/O operations are influencing battery consumption can have design implications for systems involving NFC. The environment or setting, in which the tags are used, can influence data integrity and communication range, but it is not known how much or what implications this has on systems that are developed that incorporate NFC technology. This thesis aims at producing software that enables answering of these questions as well as designing and conducting an experiment that provides insight into the performance of some tag technologies in a specific scenario.

1.2 Contribution

Existing research has mapped out general usage of NFC as a technology, ranging from studies into user interaction with NFC devices [4, 40, 51] to security aspects [31], covering the usage of NFC in medical devices [3, 23, 58], for payment and ticketing applications [24, 44] etc. Also further improvements of existing NFC protocols are discussed, [52] briefly recognizing limitations of current NFC protocols and propose future NFC protocols inspired by the TCP/IP protocol stack and discuss how power consumption for polling can be reduced to 1% of current consumption[36].

Through this thesis, I hope to answer the questions posed in Section 1.1. There is no ultimate NFC protocol that fits all use-case scenarios, but there are always protocols that are better suited for certain scenarios. One such scenario is related to the works of Zaher, in which a tag is developed that forward sensor values from inside the human body. The tag is planned to be implanted under the skin and from there connected to implanted sensors. It uses the inductive capabilities of NFC to power itself and the connected sensors as suggested in [43]. The sensor values are monitored

and offloaded to NFC readers that are within close proximity of the tag and queries for this information.

To make an informed decision about what tag technology to base Zaher's work on, it is important that the different capabilities are mapped out. The tag is required to communicate through tissue, if this significantly increases the bit error rate, it is reasonable to design the tag to transfer data in small chunks to minimize repeated transfers. If the throughput is small, it limits the update rate for the sensors or possibly impacts the granularity of the sensor values. Battery consumption impacts how the tag can be used, if repeated communication is expensive, the design needs to negotiate between an acceptable update rate and an acceptable battery consumption cost. And finally, if the phone needs to be touching the skin to communicate, a wrist watch located directly above the tag, with an NFC interface and Bluetooth capabilities might be compulsory to achieve the desired user experience. Information is then relayed through the watch to an external device such as the mobile phone as depicted in Figure 1.2.

Figure 1.2: Micro implant powered by NFC-watch

1.3 Methods

Different methods are applied to different stages of this work. Initially, a requirement analysis is done to map out areas where information about NFC is lacking. Then, as an overall procedure, a design-implementation-evaluation method is used to prepare an overall understanding of what needs to be done, how it is implemented and evaluated. The implementation of the software is done in accordance with the lean software development methodology, which belongs to the iterative software development method family [7]. Such a method is imperative to the success of the project, as it enables it to grow dynamically as new needs are identified.

1.4 The implant

This work is a prerequisite to the study of protocol selection for an Android application that communicates with a *Micro-implant* that incorporates an NFC interface. The communication technology of the implant is heavily based on tag technology; therefore, by studying the tag properties, we hope to identify what requirements should be set for the implant. As identified when comparing tags in Section 2.4, the different tags hold different properties, and we need to consider what properties are important for our use case. The following subsections discuss these properties.

1.4.1 Power source

The *micro implant* is supposed to be implanted into the wrist of patients and is connected to different sensors located inside the body. One such sensor is the *GlucoSense*, which measures blood sugar levels. The patients applicable for this *micro implant* are principally those that have diabetes type 1. For such patients, it is important to measure the insulin level of the blood regularly, which requires a blood test. With a *GlucoSense* operated in under the skin and indirectly talking to a *reader* through the *micro implant*, the blood sugar level can be monitored with much greater ease and at higher frequency. The sensor is also extracting power from the electromagnetic field created by the *reader* and therefore can operate without its own power source as illustrated in Figure 1.1. This is a key feature as changing battery on such a sensor requires surgery.

The external power supply is also a problem, because no operations can be performed when the power source is out of range and therefore no data can be collected form the sensors. To resolve this issue, a strap could be developed that is worn on the wrist and, at fixed intervals, power up the system to enable measuring of the glucose values. This records historical data in the memory of the implant and allows a *reader* to extract this information at a later point in time. Such a feature requires more memory.

1.4.2 Memory

All tags have a small part of non-volatile memory, usually an EEPROM memory. This memory defines the amount of space to make available for storing information on the tag. The way the memory is structured is tag dependent and defined in the tag specifications. However, although the tag specification does pose some restrictions to the memory structure, it also gives the designer some degree of flexibility to have some parts of the memory available for customized solutions.

As proposed in the preceding Subsection, it is possible to power the tag without performing I/O operations with it. This is expected to preserve the battery power of the reader, but requires parts of the memory to be available as a buffer to store a window of sensory values. In such a design, it is important to know how much memory is required to buffer the sensor values. This depends on the sampling rate of each connected sensor and

how often the values are offloaded from memory, *e.g.*:

$$S_m \geq B \geq \sum_{x=0}^{n} \left(\frac{S_x.S_s \times S_x.f_s}{f_o} \right), where \left| \begin{array}{lcl} S_x & = & Sensor \\ f_s & = & Sampling\,frequency \\ f_o & = & Offload\,requency \\ S_m & = & Memory\,size \\ B & = & Buffer \end{array} \right.$$

For example, a single sensor that is updated every 5 minutes, has a sampling size if 10 bits per sample and that is offloaded every 24 hours, requires a buffer of

$$\frac{\frac{1}{5} \times 10}{\frac{1}{24'60}} = 2880bits = 360bytes$$

Whilst a sampling frequency of once every 5 minutes might be suitable for collecting data about the blood sugar level, this frequency is useless for measuring the heart rate. In such a situation, the sampling rate needs to be multiple times per second and requires vast more memory or a higher offload frequency.

Using a buffer also creates complexity related to sampling frequency of the different sensors, because the frequency will vary depending on the sensor type. When multiple sensors of variable sampling frequency and varying sampling size are connected to the tag, it must be able to store these without overwriting existing samples.

An alternative to using a buffer is to simply transfer on demand. When an NFC interface is available, it can post commands and thereby read values directly from the sensors. Although this is a much simpler design, it is predicted that it consumes much energy, because each collected sample has to be transferred to the reader at once.

1.4.3 Communication speed

With the estimated recordings of 360 bytes per 24 hours, this does not put a great deal of strain on the bandwidth requirements. If 360 bytes are transferred at the rate of 106Kbps, then the information should arrive in: $\frac{360 \times 8}{106000}$ = 27ms. Hence the communication speed is not an issue in this scenario. However, if sampling the heartrate at a frequency of 200 Hz, which is required to get a mostly error free sampling [61], and with a 8 bit accuracy, the results are: $200 \times 8 \times 60 \times 60 \times 24$ = 138240000bytes. This takes: $\frac{138240000 \times 8}{106000}$ = $10400(3.s.f)$seconds or about 3 hours and requires about 134MB of memory. Such a system is required to be rethought, in this scenraio the values might have to be streamed directly to the reader for storage.

1.4.4 Read/write

It is imperative that it is possible to read from and write to the tag. This is because the tag is used to relay information from the sensors to the reader

and thus, the sensors need to write their data to the tag and the *readers* need to be able to read this data from the tag. Additionally, it is important to allow the *reader* to adjust operating parameters for the sensor. This could be stored at specific locations in the memory that the sensor reads from before performing its measurements.

1.4.5 Anti-collision

Anti-collision measures should be taken when multiple NFC interfaces are operating close to each other. However, it is unlikely that multiple NFC interfaces are interfering with each other in this specific scenario. Hence, for this tag, it is not of great importance that anti-collision is supported.

1.4.6 Computation

Performing computations on a tag requires the tag to constantly be powered by the reader. If the computations require a lot of processing, it might be more efficient to transfer the information to the reader for processing instead. In situations where the reader is a smart phone, it has vastly more processing power and is able to process larger amounts of data more power friendly and faster than computation done directly on the tag.

1.4.7 Security

The final point to consider is security. This is medical information that is made available wirelessly, without any encryption. It could be important for legal reasons that medical information needs to be secured and this area needs to be investigated. However, this is left as future works, because the security aspect of NFC is a thesis topic in itself and a lot of research in this field has already been done.

1.5 Thesis structure

This thesis is divided into three parts and 11 chapters. The first part aims at identifying why this thesis is needed and supplies the background information required to understand some of the concepts, technologies and ideas of this thesis. Especially chapters 2 to 5 should be used as a reference for technical information, as they do not introduce any knowledge directly related to the project.

Part one has five chapters. Chapter 1 introduces the thesis and briefly discusses the project. Chapter 2 gives an introduction to NFC. The tags used in this thesis and their memory structures and the command sets relevant for this thesis are introduced in Chapter 3. Chapter 4 describes the Android Operating system, whereas Chapter 5 describes some key design principles followed in the software development.

Part two describes the design and implementation of two apps, a library and an experiment. It is divided into four chapters where chapters 6 and

7 describe how the two apps and the library are developed, designed and implemented in this work. Chapter 8 discusses how the experiment was conducted in this thesis. Chapter 9 presents the data collected from the experiment.

Part three concludes this thesis and consists of two chapters. Chapter 10 presents the key results collected through the experiment and evaluates it. The final chapter, Chapter 11, evaluates the developed software and suggests the topics for further development.

Chapter 2

Identification systems

This chapter starts by introducing *Automatic Identification Systems* and shows how it is related to *Radio Frequency Identification*. The chapter continues with *Near Field Communication* and introduces some tag types that implement this technology.

2.1 Automatic identification system

Automatic identification systems have existed for decades. The most common and widespread is one that we interact with every day, barcodes. Barcodes allow alphanumeric information to be represented as an arrangement of wide and narrow bars. These bars are optically read and interpreted by an optical laser scanner. The barcode is usually used to identify the object, and information about the object is typically stored elsewhere. The reason for storing data elsewhere is the limited amount of information that can be represented by this technology. However, other automatic identification systems exist such as RFID that can store a lot more information. This makes it possible to store not only identification, but also information about the object itself, which helps to create more generic applications that require less prior knowledge about the object to be identified.

2.2 Radio frequency identification

RFID is an automatic identification system, the similarity to the barcode is obvious: they contain information about the object they are associated with. However, they communicate their information in a very different way. While barcodes store and send their information optically, RFID stores the information electronically and sends it through the use of *radio frequency* (RF). Since barcodes are optically stored, they can be written to only once, upon creation. *RFID tags*, on the other hand, can be written to and read from multiple times, as their information is stored electronically.

The operating frequencies of RFID range from the *low frequency band* up to the *Microwave band*. These different frequencies allow

Frequency band	System type	Communication range						
		3cm	10cm	30cm	1m	3m	10m	>10m
LF	Passive							
HF	ISO 14443							
	ISO 15693							
UHF	Passive							
	Active							
Microwave	Passive							
	Active							

Figure 2.1: RFID communication range [14]

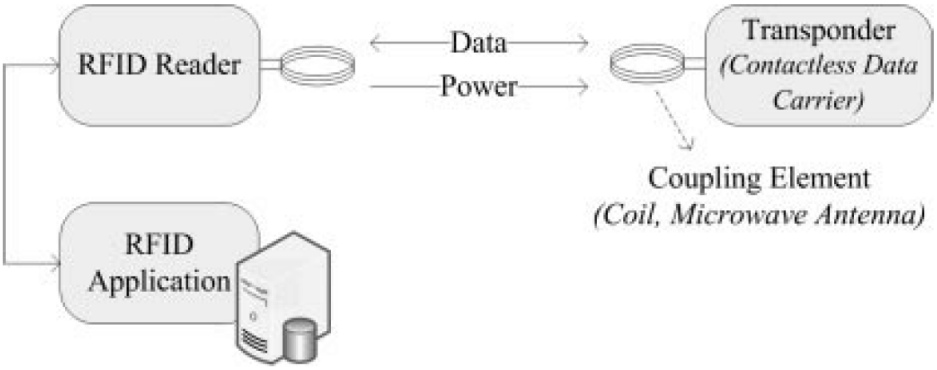


Figure 2.2: Main components of the RFID system[59]

communication over different distances ranging from 10cm up to 10 meters as depicted in Figure 2.1.

The main use of RFID is as a critical tracking and control technology. It primarily consist of two components: the *interrogator* (or reader) and the *transponder* (or RFID tag). The task of the *RFID tag* is to, upon request from a reader, send the information stored in the tag to the *reader*. The task of the *reader* is to identify the presence of a *RFID tag*, extract information from it and for some tag types, to power it. The main components of the RFID system are depicted in Figure 2.2.

2.2.1 Transponder (RFID tag)

The *RFID tag* is an integrated circuit that can hold small applications and small amounts of data. Similar to other electronic devices, *RFID tags* also need a power source. There are two methods of powering an *RFID tag*, and these two methods require two different types of tags, namely *passive tags* and *active tags*. The *passive tags* are powered by the electromagnetic field

created by the *reader,* whereas the *active tags* have their own embedded power supply. In terms of communication range, it makes sense to use the *active tags*. However, these tags are more expensive, and when the power source is consumed, the power source (or tag) needs to be replaced. A *passive tag* is more appropriate in situations where replacement is not a viable option such as in implanted *RFID tags*, or when the communication range is not an issue such as in product registration in shops.

2.2.1.1 Passive transponder (passive tags)

Passive tags consist of a *transponder chip* and an *antenna*. The *transponder chip* generates energy from the electromagnetic field created by the *reader* to operate the internal circuit. The antenna is used to transmit the information back to the *reader*.

2.2.1.2 Active transponder (active tags)

Active tags consist of an embedded *power source*, a *transponder chip* and an *antenna*. The power source is used to power the *transponder chip*, which generates the signal to be transmitted by the *antenna*. Since it has an embedded power source, the *active tag* can produce a much stronger signal and therefore operate over a much longer distance.

2.2.2 Reader

The *reader* continually polls for nearby *RFID tags* and creates an electromagnetic field. When a tag is within range, the tag responds to the signal created by the *reader* by modulating the electromagnetic field. By interpreting the modulated electromagnetic field, the *reader* may extract the information. This is illustrated in Figure 2.2.

2.3 Near field communication

NFC is a close cousin of *RFID* that extends the capabilities of RFID tags. It uses the bandwidth 13.56 MHz for communication, which gives it a maximum communication range of up to 20 cm [15, p. 57]. The extension to RFID enables NFC to operate in three modes, *read/write operation mode*, *card-emulation mode* and *peer-to-peer mode*. The *read/write operating mode* enables the NFC device to replicate a RFID reader, to enable reading from and writing to RFID tags. Through the properties of *peer-to-peer* mode, the NFC device adopts data sharing abilities similar to *wireless communication devices* such as Bluetooth, Wi-Fi and infrared.

One of this technology's prime advantages is its implicit security due to the short communication range. This property makes it more resilient against eavesdroppers and makes it a suitable candidate for transferring sensitive data. This property is used for implicit pairing of Bluetooth enabled devices. With the use of *Bluetooth Secure Simple Pairing* (SSP),

the devices can exchange authentication information over a NFC link [16] to establish a Bluetooth connection without the need of any user interaction. SSP uses NFC as the secure element in initiating data communication between two Bluetooth enabled devices. This eliminates the need for manual pairing.

The gesture of moving the device into close proximity of a target is unlikely to happen by accident, and it can therefore be assumed that the user wants to perform some type of action. This feature is extensively used in the Android operating system where NFC tags can be associated with the launching of apps and configuration of the phone. For example, entering the car and touching the phone to a NFC tag mounted somewhere in the car may configure the phone to connect to the car's Bluetooth handsfree set and launch the GPS app on the phone.

2.3.1 Communication modes

For two NFC devices to be able to communicate, the devices can take on either an active or passive communication mode.

2.3.1.1 Active

In active mode, both NFC interfaces generate a magnetic field and send their data by modulating it. They do so in an alternating fashion to avoid collision. The advantages of this mode are that it operates over a longer distance and the power consumption is shared between the devices.

2.3.1.2 Passive

In passive mode, the NFC interface that initiates the communication provides the magnetic field. The target can draw energy from the magnetic field, but cannot create its own. The initiator sends data by modulating its magnetic field and the target sends by load-modulation. The transfer speeds available with this communication mode are 106, 212, and 424 Kbit/sec [59, p. 102], depending on the type of tag it communicates with.

2.3.2 Operating modes

There are three main smart devices in NFC: NFC enabled mobile phones, NFC readers and NFC tags. The valid NFC device combinations are summarized in Figure 2.1. To allow the NFC enabled phone to communicate with all three device types, it needs three different operating modes: *read-/write*, *peer-to-peer* and *card-emulation*. The *read/write* mode allows the NFC phone to communicate with tags by simulating the role of a *reader*. The *card-emulation* mode allows it to communicate with NFC readers by simulating a smart card. In *peer-to-peer* mode, it can communicate with other NFC enabled mobile phones.

Device	Mobile phone	Reader	Tag
Mobile phone	X	X	X
Reader	X	-	X
Tag	X	X	-

Table 2.1: NFC combinations

2.3.2.1 Read/write

An NFC device can operate in reader/writer mode, when there is another NFC device in range that is in card emulation mode. This allows the reader to transmit commands to the NFC device in card emulation mode and receive the response. This is used when NFC phones interrogate NFC tags, as described in Section 2.4.

This mode is used to write data to, or read data from, a NFC tag. From the NFC tag's side, the phone is perceived as a *reader*.

2.3.2.2 Peer-to-peer

This mode is used to allow NFC enabled mobile phones to exchange information between each other. This mode is used when performing SSP to allow NFC enabled devices to implicitly establish a Bluetooth connection.

2.3.2.3 Card-emulation

In this mode, the mobile phone simulates the properties of a smart card or a tag so that the phone can be read by a *reader*. This mode is useful for contactless payment applications, because it gives payment issuers greater control over their payment strategies as the credentials can be stored on a remote server rather than on the mobile device.

2.4 NFC tags

NFC tags are passive RFID tags that are used as small data containers and that respond to certain commands that are defined by the standard they belong to. They enable communication with *readers* and, depending on type, can allow *readers* to write information to their memory. Four types have been defined by the NFC forum and are compared in Table 2.2. As the Table shows, the tags have varying properties related to memory size, communication speed, cost, read/write abilities, anti-collision support, security and standards that they are based on.

Memory defines the available space for storing information on the tag. It is important to define what memory space is required as large memory space increase complexity and cost of the tag to be developed. For example, a sensor can write its sensed data in timed intervals to the tag and allow the tag to buffer this data. Once a reader is available for offloading the information from the tag, the tag is free to clear the memory and make

Type	Tag 1	Tag 2	Tag 3	Tag 4
Memory(B)	<= 1K	<= 2K	<= 1M	<= 64K
Com. speed(Kbps)	106	106	212/424	106/212/424
Cost	Low	Low	High	Medium/High
R/W	Yes	Yes	Variable	Variable
Anti-collision	No	Yes	Yes	Yes
Security	16/32 bit Signature	Insecure	16/32 bit Signature	Variable
Standard	ISO-14443A	ISO-14443A	JIS X 6319-4	ISO-14443A or B

Table 2.2: Tag type properties

space for new readings. In this scenario, the size of the sensed data, frequency of recordings and the frequency of offloading are key factors to consider when deciding on what tag to choose.

Similarly the communication speed puts requirements to the hardware, making it more expensive to develop and manufacture. Generally, the amount of data transmitted from the tag to the *reader* does not require a large bandwidth. For example, if the tag is updated with insulin blood levels at intervals of five minutes, and each recording is one byte long, the required transfer speed is way below 106Kbps and hence tag type 1, 2 and 4 are suitable in relation to speed. However, transfer speed becomes a key factor in situations where real-time monitoring of sensor data, such as heart rate, is stored on a tag and needs to be continuously transferred to the *reader*. To decide on the type of tag, one needs to consider the frequency of transfers, the size of the data to transfer and how the transfer speed affects the user experience.

The cost is always an issue, both in development and in production. The cost reflects the complexity of the system, and therefore the complexity should be kept to a minimum.

All the tags presented have read/write capability, however, a subset of them (*e.g.* tag type 1, 3 and 4) also have memory areas that can be protected by switching them from read/write to read only memory. This can be done at manufacturing time or in some tags by the user. The advantage is that the tag cannot be tampered with whether it is intentional or not.

Anti-collision ensures that when multiple sensors are sharing a medium, they negotiate who transfers information and when. This is important if the tag operates in an environment surrounded by other tags. The introduction of anti-collision does incur a significant increase in complexity to the hardware of the tag and the need should therefore be carefully considered. When multiple sensors are making recordings in a confined area and storing the information on tags, it could be sensible to include anti-collision measures to ensure that information can flow fluently. However, in the case that is considered at the end of this paper, only one tag is used and therefore this situation does not occur.

<table>
<tr><th>Layer</th><th>With NDEF</th><th>Without NDEF</th></tr>
<tr><td>Application</td><td>NDEF application</td><td rowspan="3">Non NDEF applications</td></tr>
<tr><td>ISO-14443-4</td><td rowspan="2">NFC Forum Mandated Tag Operations</td></tr>
<tr><td>ISO-14443-3</td></tr>
<tr><td>ISO-14443-2</td><td colspan="2">Digital protocol</td></tr>
<tr><td>ISO-14443-1</td><td colspan="2">Analog</td></tr>
</table>

Figure 2.3: Protocol stack for R/W mode

Security issues related to tags are covered by earlier studies[12, 31]. It prevents eavesdroppers from being able to study the information available on the tag. The NFC forum has standardized RDT, a digital signature system that enable secure transfer of NDEF packets through the use of private and public keys. The information transmitted between the NFC interfaces is then being encrypted with the public key and can only be decrypted with the private key. Hence, when transmitting, the transmitter encrypts with the recipients public key and the recipient decrypts with its private key. Anyone listening in, is unable to decrypt the information without the private key.

2.5 Protocols

The area of interest for this paper is the communication between an NFC enabled device, primarily a mobile phone, and *NFC tag*s. Therefore the emphasis on protocols are related to protocols for *read/write* mode. Furthermore the paper focuses on the implementation offered by Android's application framework [25]. Essentially this offers two forms of communication with NFC enabled devices: through *NDEF messages*, which specify protocols based on tag type, and through *non NDEF messages*, where the protocol stack is implemented by the application developer directly. The two protocol stacks are illustrated in Figure 2.2.

The *NFC Forum Mandated Tag Operations*-layer delivers NDEF messages to the NDEF application and allows the NDEF application to send NDEF messages to the target without the need to know the specifics of the implementation. This layer identifies the tag type and performs operations accordingly, leaving the application with a simplified interface for communicating with all NFC mandated tag types.

In non NDEF applications, there is no reliability in the data transmitted. Simply raw bytes are transferred from the *Digital protocol layer* to the application. Due to this, if we want to implement such a communication, we need to define a set of protocols that handle communication with the device. To acquire reliable connection with the device, we need to define a link layer protocol for transmission of data back and forth.

In the OSI model, the data link layer handles communication between two nodes in a network. This layer is divided into two sub-layers, the LLC and MAC. The LLC layer is the uppermost of the two and multiplexes between communication protocols.

The link layer offers a set of possible functionalities for using the communication channel. Firstly, if the communication channel is shared, there is a need for a medium access control scheme. This scheme negotiates a solution to reduce collisions when multiple devices are trying to communicate simultaneously. Secondly, if the communication channel is not error free, and validity of the data is essential, we need to have a way of detecting and possibly fixing errors in the data transmitted. Lastly, to enable variable size data transmission, we need a flow control system.

2.6 Conclusion

NFC is a flexible and expandable standard that builds on top of RFID and offers a large range of uses. To help standardizing the implementation of tags, the NFC Forum has defined four tag types that have some pre-defined features. However, as becomes apparent in the following chapter, the way the tags are defined leaves the developer open to customize the tag and extend the tags functionality.

Chapter 3

NFC tags

This chapter consists of five sections, each of which introduces a new tag type together with its basic physical characteristics such as communication speed, memory size and anti-collision support. Each section includes three sub-sections that explain the general memory layout of the tag and how reading and writing operations are performed. At the end of each section, the implementation of a tag that complies with these specifications is presented together with its physical attributes. The tags presented are used for testing in Part II. It must be noted that the description is compressed to the essential knowledge required to understand the implementation of these protocols in the developed software introduced in Part II. As such, the full flexibility and all possibilities of the tag types are not explored. However, reference to relevant documentation, where the full description is available, is supplied in the text.

3.1 NFC Forum Type 1 Tags

This section introduces the NFC Forum Type 1 tag, which is based on the ISO-14443A [29] standard. A more complete description can be found in the official *NFC Forum Type 1 Tag Operating Specifications* [17]. This tag type communicates with a speed of 106Kbits/s and has an expandable memory of 64B up to 2KB, but it does not have any anti-collision support [41].

3.1.1 Memory

The NFC Forum Type 1 tags have two memory structures: a static memory of 120 bytes and a dynamic memory stretching beyond 120 bytes. The memory is considered to be divided into blocks of 8 bytes, and the division of these blocks is illustrated in Figure 3.1. As can be seen, the first block contains manufacturing data, whereas block 13 and 14 are reserved for internal use and locking bits. Locking bits control read and write access to blocks, and once it is set, it cannot be changed. More information about locking bits is available from the official NFC Forum Type 1 tag operating description [17]. The dynamic memory is further divided into segments,

where each segment consists of 16 blocks. The address space for both static and dynamic memory is flat.

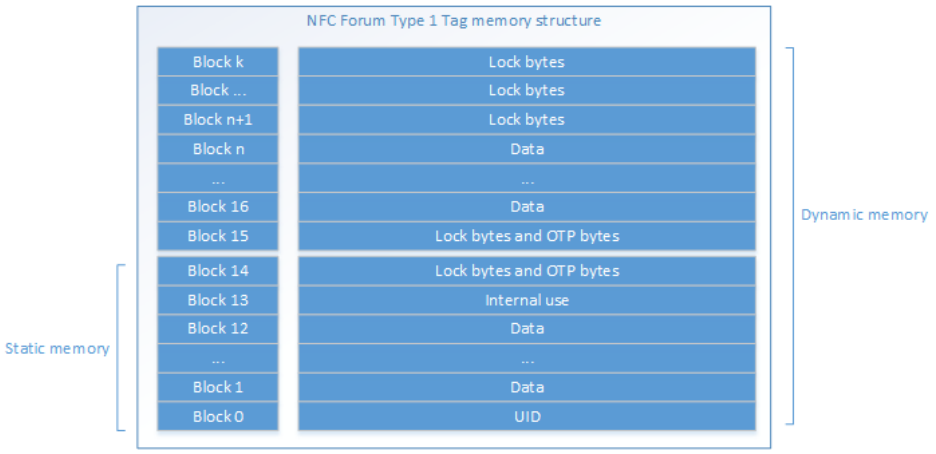


Figure 3.1: Memory layout of NFC Forum Type 1 Tag

3.1.2 Read command

There are four commands available for reading NFC Forum Type 1 tags, *read*, *rall*, *rseg* and *read8*. The *rall* and *read* commands are required for tags with static memory, whereas the *rseg* and *read8* commands are required for tags with dynamic memory. The *read* command contains the number of the byte to read together with the first four UID bytes of the tag. The tag responds with the referred address and a single byte. The *rall* command includes the first four bytes of the tag UID, and the tag responds with a copy of the first 128 bytes of its memory. The *read8* command contains the number of the block to read from and the first four bytes of the tag UID. The tag responds with the bytes available from the referenced block. The *rseg* command contains a segment number and the first four bytes of the tag UID. The tag responds with the bytes found in the referenced segment.

3.1.3 Write command

Similar to reading, there are four commands available for writing to NFC Forum Type 1 tags: *write-e* and *write-ne*, which are required for tags with static memory, and *write8-e* and *write8-ne*, which are required for tags with dynamic memory. The *write-e* command contains a byte, a byte address and the first four bytes in the tag UID. The tag proceeds with updating the byte with the referenced address with the given byte. This command gives access to blocks 1 to 12 and thus prevents overwriting any settings bits. The *write-ne* gives full write access to blocks 0 to 14, but does not reset the byte it writes to and any bits that are set in the byte it writes to, remain set after the operation completes. The *write8-e* is identical to the *write-e* command, with the difference that it works with blocks instead

of bytes. The *write8-ne* is identical to the *write-ne* command, with the difference that the operation is performed on a full block instead of a single byte.

3.1.4 Discussion

NFC Forum type 1 tags have very simple read and write commands with a low overhead that transfer one or eight bytes at a time. This makes it efficient for transferring small loads. However, it also makes it expensive to transfer larger chunks of data. This tag also specifies a single *rall* command that reads 128 bytes of the tags memory with a very low overhead. This low overhead makes reading the full memory efficient. However, this does scale well for medium sized data transfers.

3.1.5 Topaz 512

This is a tag that operates according to the specifications of NFC Forum Type 1 tag. It has a dynamic memory that expands to 512 bytes and it supports commands for both static and dynamic memory access. However the static memory access is limited to the first 128 bytes. The app described in Chapter 6 only supports commands required for the static memory model. This tag is throughout the remainder of this work referred to as Tag1.

3.2 NFC Forum Type 2 Tags

This section introduces the NFC Forum Type 2 tag, which is based on the ISO-14443A [29] standard. A more complete description can be found in the official *NFC Forum Type 2 Tag Operating Specifications* [18]. It has anti-collision support, communicates with a speed of 106Kbits/s and has an expandable memory of 64B up to 2KB [18, 41].

3.2.1 Memory

Tag 2 has two defined memory layouts, static and dynamic. The static layout has 64 bytes of memory, whereas the dynamic layout is larger in size. The page size of this tag is four bytes and is referred to as a block. Bytes contained in a block cannot be individually modified without overwriting the remaining bytes in the block. Blocks are grouped in sectors, which are defined as 256 contiguous blocks. The blocks are addressed relative to the sector they belong to.

As is illustrated in Figure 3.2, the first four blocks contain manufacturing data and tag settings. Special attention should paid not to overwrite any of this data unintentionally. The two MSB (most significant bytes) of the third block contain locking bits for the first 15 blocks. The information on how the locking bits work can be found in [18, p.5].

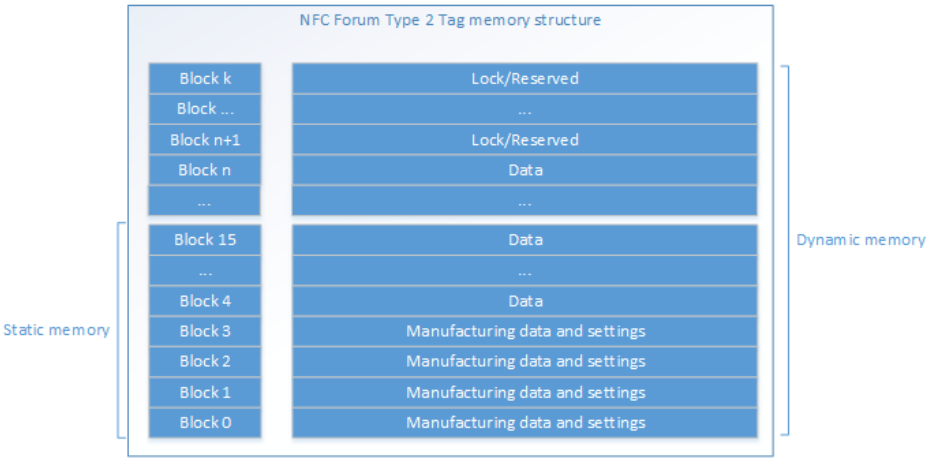


Figure 3.2: Memory layout of NFC Forum Type 2 Tag

3.2.2 Read command

The *read* command of NFC Forum Type 2 tags define the starting block to read from. The tag responds with 16 consecutive bytes starting from the first byte in the block defined in the query. This command is not able to select the sector to read from and therefore uses the currently selected sector. To select a different sector, it is necessary to issue a *sector select* command [18, p.18].

3.2.3 Write command

The *write* command of NFC Forum Type 2 tags defines the block to write to together with the four bytes that are written to this block. If a subset of the bytes in a block is to be updated, it is necessary to perform a *read* operation of the designated block and alter the desired bytes before issuing the *write* command. Similar to the *read* command, this command is not able to select the sector to write to and so this has to be accomplished through a *sector select* command [18, p.18].

3.2.4 Discussion

NFC Forum type 2 tags have very short commands, which create small overhead with efficient use of the NFC. This is optimal for reading and writing smaller amounts of data. The tag does not have any built-in security and information stored on it is, therefore, available for anyone wnh has a chance to scan it.

3.2.5 NTAG203

An existing NFC Forum Type 2 tag is the *NTAG203* from NXP Semiconductors. This is a tag with a dynamic memory layout and 144 bytes of user memory [53]. The software described in Chapter 7 has been tailored to

work with this memory size. This tag is throughout the remainder of this work referred to as Tag2.

3.3 NFC Forum Type 3 Tags

This section introduces the NFC Forum Type 3 tag, which is based on the JIS X 6319-4 [32] standard. A more complete description can be found in the official *NFC Forum Type 3 Tag Operating Specifications* [19]. It has anti-collision support, variable memory up to 32KB per service[1] and supports communication speeds of 106, 212 and 424Kbits/s[19, 41]. The concepts of reading and writing information to this tag are covered in the preceding sub-sections.

3.3.1 Memory

The memory of a NFC Forum Type 3 tag has a page size of 16 bytes that is referred to as blocks in this standard. Memory management on this tag differs from the two preceding tags in that it does not supply direct access to the memory chip. Instead, it maintains a type of file system, where services supply access to a subset of memory blocks and where these memory blocks are addressable relative to the service they belong to. Thus, the operating system on the chip takes care of mapping the block addresses in the service to the physical location on the chip. An illustration of the memory division is displayed in Figure 3.3. Each service is uniquely identified by a service code, which also determines the type of access this service grants to its associated blocks. The access type is *read only* or *read and write*.

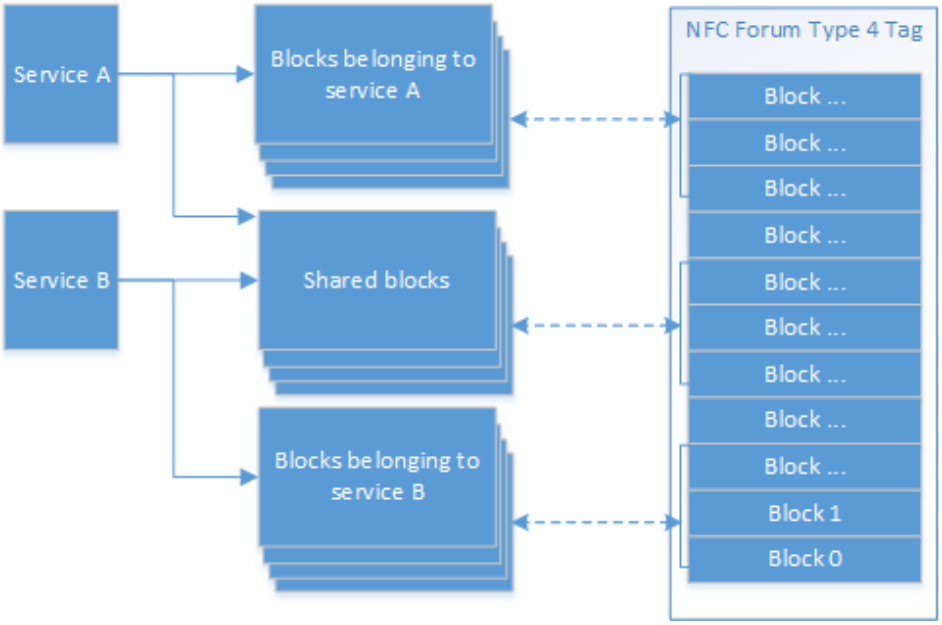


Figure 3.3: Memory layout of NFC Forum Type 3 Tag

[1] A service grants access to a memory segment.

3.3.2 Read command

This command contains the full eight byte UID of the tag, a list of all services to work with and all blocks to read from. It is up to the manufacturer to determine how many services can be active simultaneously and how many blocks that can be read simultaneously. Since each block is uniquely addressed, the blocks read do not need to be situated contiguously in memory.

3.3.3 Write command

The *write* command contains the full eight byte UID, the start block, the end block and the data to transfer. The standard is designed to allow up to four blocks of data to be written to simultaneously. However some tag suppliers only support the writing of one block at a time.

3.3.4 Discussion

The flexibility of the read command enables the protocol to cater for variable needs without excessive overhead. Since blocks are individually addressable in this standard, they do not need to be contiguously located in memory, which improves the flexibility when designing how the memory is used. The disadvantage of reading block sized chunks is that if an individual byte is to be altered, the memory block needs to be "swapped in" (read from the tag) before it is changed with the desired byte and then written back to the tag.

3.3.5 FeliCa Lite-S

An existing NFC Forum Type 3 tag is the FeliCa Lite-S tag developed by Sony. In this subsection, the specifics relating to the implementation of the FeliCa Lite-S are introduced. The tag grants access to blocks through two services: one, which gives read access and is referred to as the *read service*, and the other, which gives read and write access and is referred to as the *read-write service*. The *read service* gives read access to the set of read-only memory blocks and the set of read-write memory blocks, whereas the *read-write service* only grants access to the read-write memory blocks. The memory layout of this tag is illustrated in Figure 3.4.

The structure of the FeliCa Lite-S memory is such that the first 14 blocks are defined as user blocks. User blocks are read-write blocks and do not affect the internal settings of the tag. They are therefore available for the user to use as he/she sees fit. In the Nfc.Communication framework, reading and writing access is limited to these 15 user blocks. This is done to abstract the user away from knowing the internal memory structure of each individual tag and simply view the tag as a storage device.

In this tag, a read command enables reading of up to four individually addressable memory blocks. The size of the command varies depending on the number of blocks that are read. As a response, the tag collects the bytes available in the desired blocks and returns this as a response.

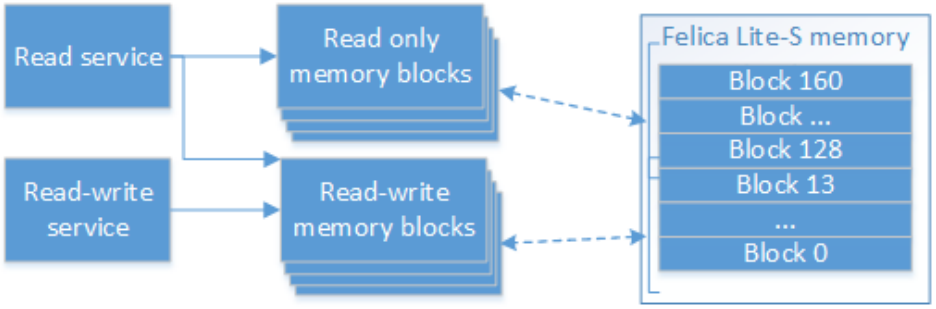


Figure 3.4: Memory layout of FeliCa Lite-S

A write command with FeliCa Lite-S enables the user to write to up to two blocks at a time. The command expects the number of bytes to be written to reflect the number of blocks to write to. Therefore, even if only one byte is to be written, the remaining 15 bytes needs to be padded into the array in order for the command to be accepted.

This tag is throughout the remainder of this work referred to as Tag3.

3.4 NFC Forum Type 4 Tags

This section introduces the NFC Forum Type 4 tag, which is based on the ISO-14443-4 [30] standard. A more complete description can be found in the official *NFC Forum Type 4 Tag Operating Specifications* [20]. This tag supports the ISO/IEC 7816 APDU message structure and this section presents some of the key elements of this standard.

3.4.1 Memory

The memory management of this tag resembles that of NFC Forum Type 3 tags as described in subsection 3.3. It also employs a file system for managing the memory of the tag. The file system consists of a set of applications, where each application grants access to a subset of files. These files grant read and write access to pre-defined areas of the memory. Before any IO operations can be performed, an application needs to be selected and with it the file that it should operate with. Once the file is selected, IO commands can be issued for updating or reading contiguous parts of the file. The standard does not pose any limitations on the amount of bytes that can be updated in one command. The layout of the memory is illustrated in Figure 3.5.

3.4.2 Read command

Before any read operations are executed, it is necessary to select an application and a file associated with this application. The read command consists of an offset into the file and the number of bytes to read. If the number of bytes to read exceeds 59 bytes, it is split into multiple transfer operations.

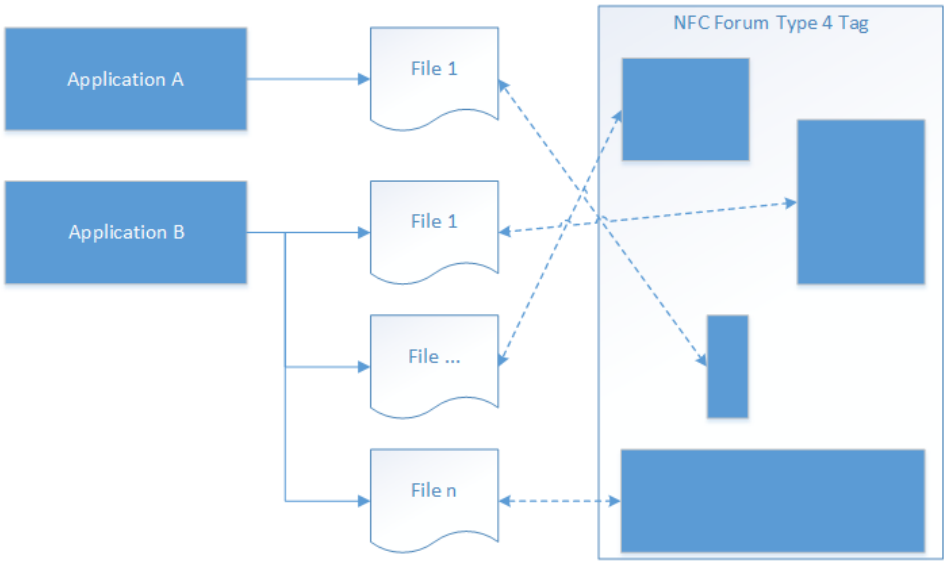


Figure 3.5: Memory layout of NFC Forum Type 4 tag

3.4.3 Write command

As with the read command (see subsection 3.4.2), the write command also requires an application and the desired file to be selected. Similar to the read command, the write command consists of an offset into the file and the number of bytes to write. If the bytes to write exceed 52 bytes, it is split into multiple write commands, where the first gives information about the size and the offset into the file, and the consecutive commands simply transfer bytes.

3.4.4 Discussion

NFC Forum type 4 tags have very flexible read an write commands that scale well with the size of the packet being transmitted when reading from or writing to the tag. However, it also has some overhead when writing as it needs to select the correct application and file on the tag. Additionally, all read and write operations have to be performed on contiguous memory.

3.4.5 Mifare Desfire

An existing NFC Forum Type 3 tag is the MifareDesfire tag developed by Philips Semiconductors. This tag has 4kB of non-volatile memory. The tag has an extension to the NFC Forum Type 4 tag command set, which allows it to create and manage applications and files on demand on the chip. More details about such procedures can be found in the official Mifare Desfire datasheet supplied by Sony Semiconductors [54].

In the application described in Chapter 7, an application, and a file associated with this application, is created on this tag unless they exist from before. The file is created with a size of 512 bytes of memory.

This tag is throughout the remainder of this work referred to as Tag4.

3.5 MIFARE Classic

MIFARE Classic is a proprietary tag based on the ISO-14443A standard [29]. It has anti-collision support, variable memory size and a communication speed of 106Kbits/s [41]. The memory size varies from 1KB to 4KB. Compared to the preceding tags, this one also supplies secure reading and writing to sectors[2] of the tag. This section introduces the concepts of I/O operations, security and the memory for this tag.

3.5.1 Memory

The memory in MIFARE classic tags is divided into pages with a size of 16 bytes. These pages are in this standard called blocks. Each block is individually addressable by a block number and belongs to a sector, which is addressable by a sector number. Access to a block within a sector requires authentication with the sector. The authentication process is described in Section 3.5.2. The memory is illustrated in Figure 3.6 and shows that the first block contains manufacturer data such as the UID of the tag and is therefore ROM. The last block of each sector contains two keys and access bits, which are used in the authentication process. It is important to take care not to overwrite these values unintentionally, as they can render the sector inaccessible. Not all MIFARE Classic tags have sectors with 16 blocks. However the larger tags such as the 4KB sized tag does, and it is therefore necessary to cater for this when designing a framework for manipulating the tags memory.

3.5.2 Security

Secure access to the memory area of a Mifare Classic tag is gained through the division of the memory into sectors. Each sector controls access to a set of either 4 or 16 blocks. When the interrogator requests authentication with a sector, it needs to specify which key it desires (A or B) together with the six byte key. This key is compared with the key found in the last block of the given sector. Furthermore, the “Access” bits, which are illustrated in Figure 3.6, determine what type of access is granted. The authentication procedure is illustrated in Figure 3.7. More information on authentication can be found in [57].

3.5.3 Read command

Reading from this tag is performed through a read command that specifies the block number to read. The tag responds with the 16 bytes found in this block.

[2] Sectors refer to predefined segments of memory on the tag

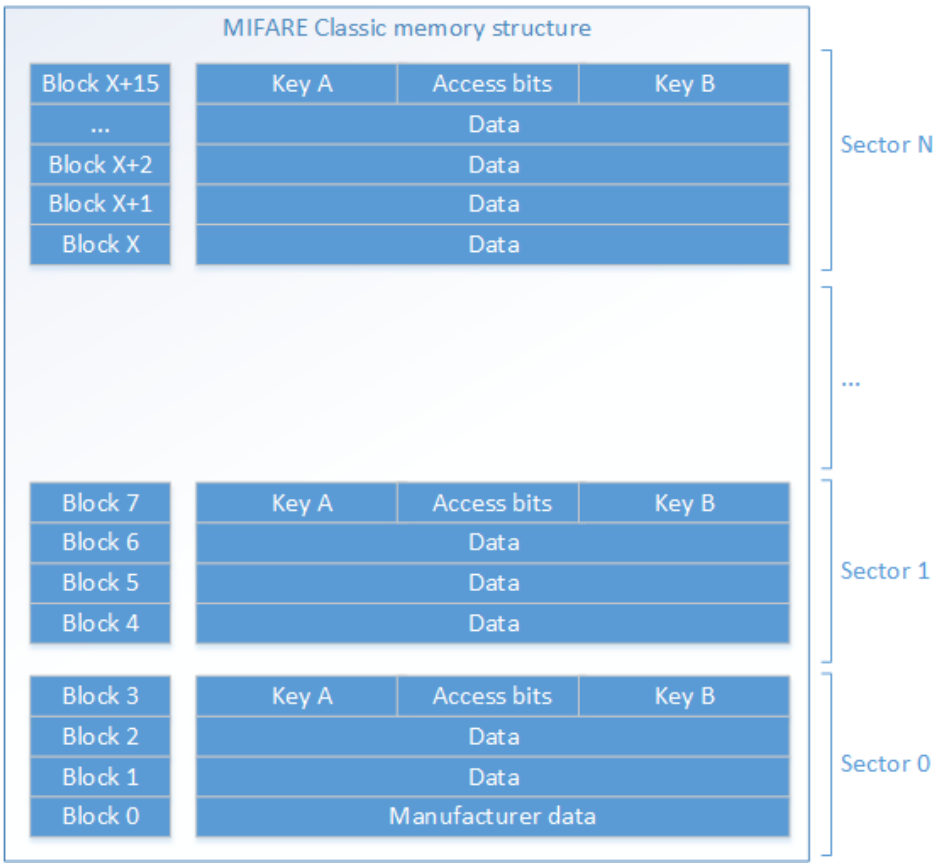


Figure 3.6: Memory layout of MIFARE Classic

3.5.4 Write command

Writing to this tag is performed through a write command that specifies the block number to write to and an array of 16 bytes to write. Because this tag writes a full block of 16 consecutive bytes, it is required to read the block first, if one desires to update a subset of the bytes residing in the block.

3.5.5 Discussion

The built in security feature of MifareClassic gives some protection to the data stored, but it also requires some overhead with data transfer such as when authenticating sectors. Additionally, read and write commands transfer large amounts of data, which incurs a large overhead when transferring smaller packets to the tag.

3.5.6 MF1S50

MF1S50 is a Mifare Classic tag with 1kB of memory distributed as displayed in Figure 3.8. As can be seen from the illustration, all sectors on this tag have four blocks comprising of 16 bytes of memory each. In total this renders 752 bytes of read-write memory space.

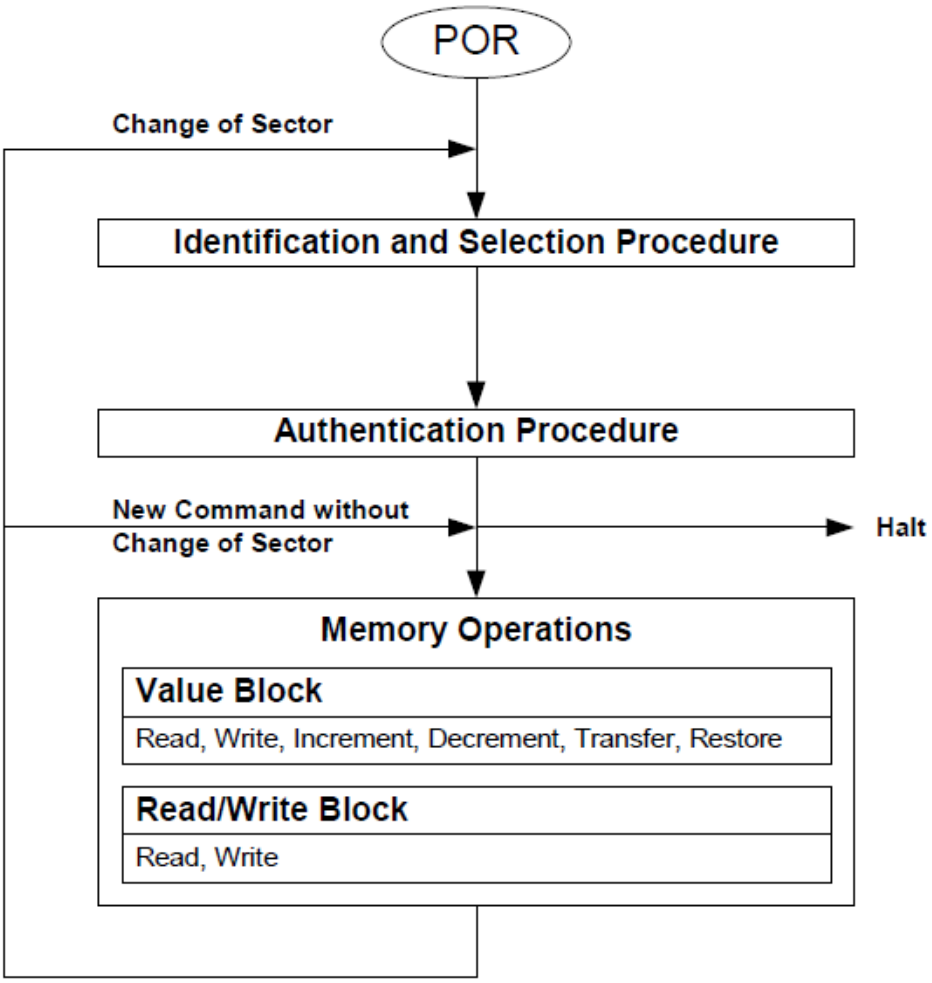


Figure 3.7: MIFARE Classic authentication procedure[38]

This tag is throughout the remainder of this work referred to as MifareClassic.

3.6 Conclusion

To conclude, the flexibility of the different tag type standards leaves great flexibility to the manufacturer of the tag. This requires application developers to have knowledge about both the tag type standard and the actual tag implementation. This flexibility is both a blessing and a curse, as it enables tag manufacturers to develop tags tailored to specific scenarios, which in turn makes it harder to develop generic systems for handling all tags.

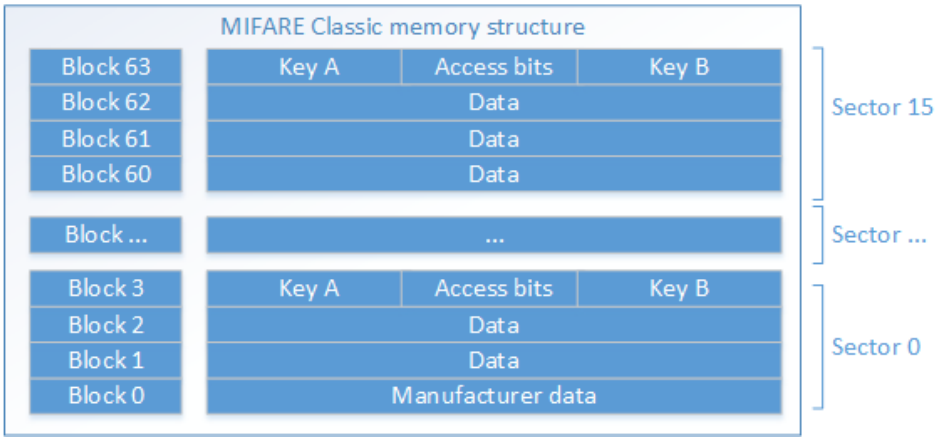


Figure 3.8: Memory layout of MF1S50

Chapter 4

Android OS

This chapter describes the Android operating system, some of its history, its structure and app development for it. The chapter touches on some key components and concepts related to developing apps on the Android platform such as the Activity and Fragments, xml layout files and Intents. To conclude, it introduces and discusses some IDE's available for helping developers with creating apps.

4.1 History

Android Inc. is the start-up company that developed Android, which is an Open Source operating system that is based on the Linux kernel. This company was bought by Google in 2005, only 22 months old [55]. In 2007, the Open Handset Alliance[6, 42] is formed, which results in a significant amount of intellectual property being added by its key members to the Android Platform.

In 2008, the first Android driven mobile phone is released by T-Mobile, the G1[26]. It is the start of an exponential growth of devices running the Android operating system, which now cover 72% of the global smartphone marked [63].

4.2 Android architecture

The Android OS is composed of a set of layers as depicted in Figure 4.1. At the core is the *Linux Kernel* and above lies the *Native layer*, which contains native libraries that are hardware specific. The *Application Framework* is the next layer, and it contains the building blocks that developers' apps interact with. At the top lies the *Application Layer*, and this is where user-developed apps reside.

4.2.1 Linux Kernel

The Linux kernel residing at the core of Android is a derived version of the vanilla flavored Linux Kernel [63], patched to fit the needs of Android. As

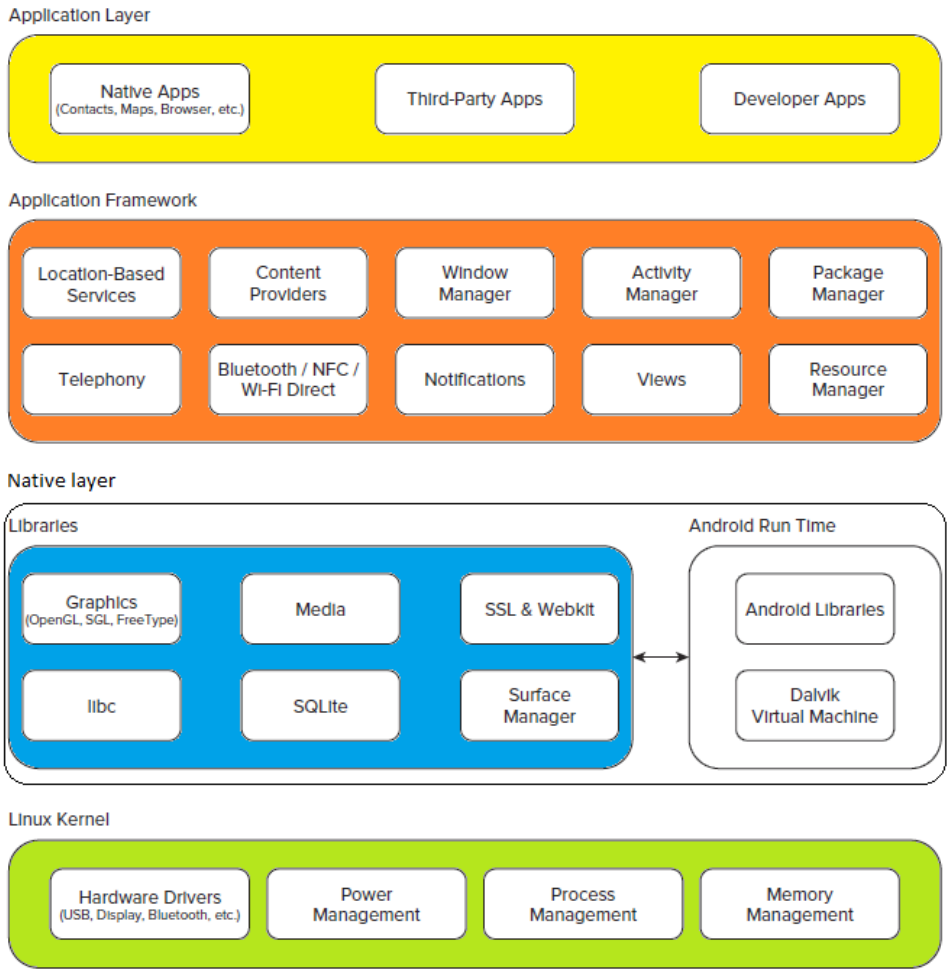


Figure 4.1: Android architecture [37]

such it is not a trivial task to replace it with a kernel from kernel.org because some major changes are required to enable the rest of the Android stack to run correctly. For example, the Binder, which is extensively used for system services in Android, is an Android specific component that resides inside the kernel. Without it, these system services do not run.

4.2.2 Native Layer

The native layer maintains core libraries that are designed to handle different types of data. Amongst other things, the SQLite library resides here, this is a library that contains a relational database engine and allows efficient persistence of data. This layer also holds the Dalvik Virtual Machine [63], which is the software that runs apps on Android.

4.2.3 Application Framework

The application framework exposes the functionalities supplied by the Android OS to the app developer. It also supplies an abstraction of hardware access, which enables the developer to use one API to control different hardware components.

4.2.4 Application layer

This layer contains all apps, both those that come pre-installed on the device and those that are developed for the device.

4.3 Software components

This Section introduces Services, Intents, Activities, Fragments and SQLite, which are five key software components in Android that are extensively used in this thesis.

4.3.1 Services

Services are ways for processes to work in the background. More specifically, they linger in the background and supply any app with the right privileges, access to their functionality. This is how it is possible to start a pdf viewer app to display a pdf document from within another app. This is made possible through Androids Binder driver, which resides in the kernel. Because of this, it is able to forward messages to the service, which can execute processes based on these messages and return results. This seems to the caller as a synchronous call to a local function, because the thread is blocked in the binder, waiting for the service to return the result (thread migration). This is illustrated in Figure4.2.

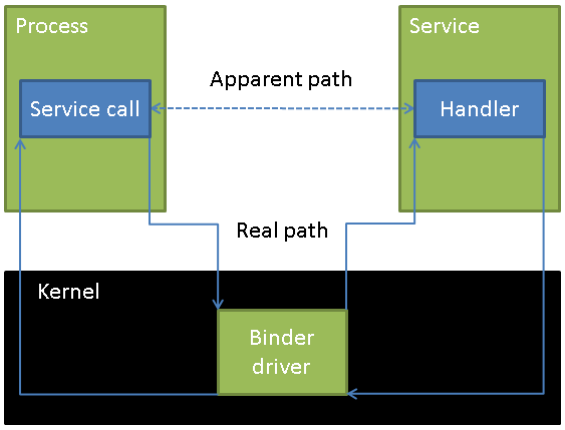


Figure 4.2: Binder driver

4.3.2 Intent

One of the most important components of Android is Intents. Intents allow apps to perform *inter process communication* (IPC) amongst each other. It is also what is used for apps to use Services. The developer can define in the Intent filter of an app's manifest file, that the app is interested in Intents from a given Service. By doing this, the app is able to receive Intents from the Service. The Intent filter for listening for the tag technology discovered Intent is supplied in Listing 4.1.

Listing 4.1: Intent filter

```
1 <intent-filter>
2     <action android:name="android.nfc.action.TECH_DISCOVERED"
          />"
3 </intent-filter>
```

When a service wants to broadcast some event to registered apps, Android investigates the Intent filter of all installed apps, the filters that includes the Intent being broadcasted, receives the Intent. Therefore, the example in Listing 4.1 enables the app to receive *android.nfc.action.TECH_DISCOVERED* Intents which are broadcasted when NFC tags are discovered.

4.3.3 Activity

Android applications consist of one or more Activities [5]. These Activities can inflate Android layout files and thereby create their own UIs. An app can consist of multiple Activities that can be produced on demand and implements their own user interaction logic. The Activities are designed to fill one screen with components, and they can therefore not reside inside each other.

All Activities extend the android.app.Activity class (or derived versions of this). By extending this class, they fall under the control of the Activity lifecycle, which is illustrated in Figure 4.3. When the app is launched,

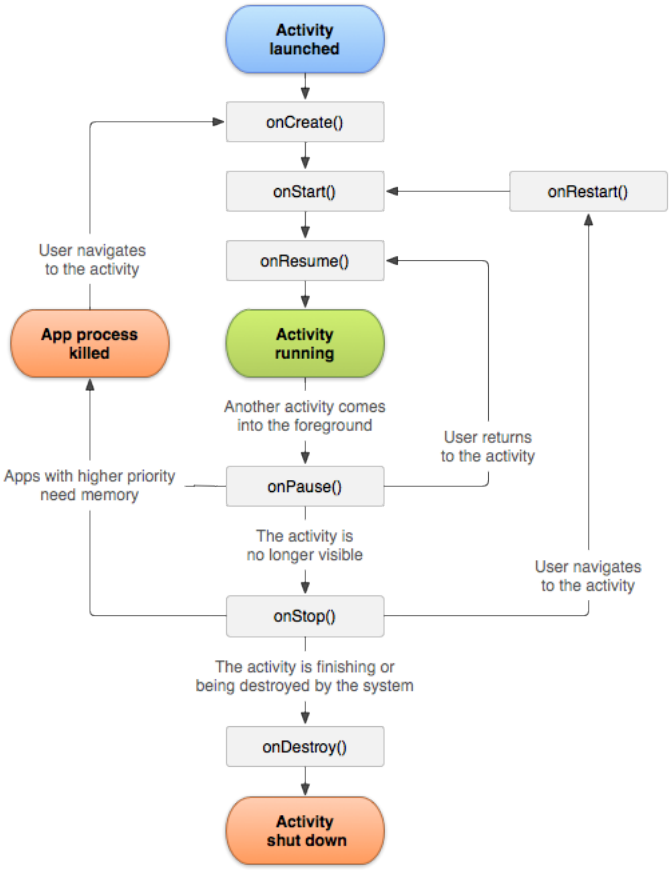


Figure 4.3: Activity lifecycle [13]

it initiates the *onCreate()* function, which allows the Activity to inflate a layout file. The layout file contains xml description used by Android to display widgets and other UI components. Once this is finished, it progresses to the *onStart()* function. Once this function is called, the UI components are already available and any changes to these, such as programmatically adding new UI components, need to be done post this stage. Right before the app comes to the foreground, it calls on the *onResume()* function.

The Activity has three states. When it is in the foreground and running, it is in the *running* state, this enables user interaction. When it has lost focus but is still visible, it is in the *paused* state. And lastly, when the Activity is hidden from view by another app or Activity, it is in the *stopped* state. Transcending between these states causes the *onDestroy()*/*onCreate()*, *onStop()*/*onStart()* and *onPause()*/*onResume()* functions to be called.

4.3.4 Fragment

The lacking ability of reusing Activities inside other Activities has evolved the need for Fragments. Fragments are basically UI components that can

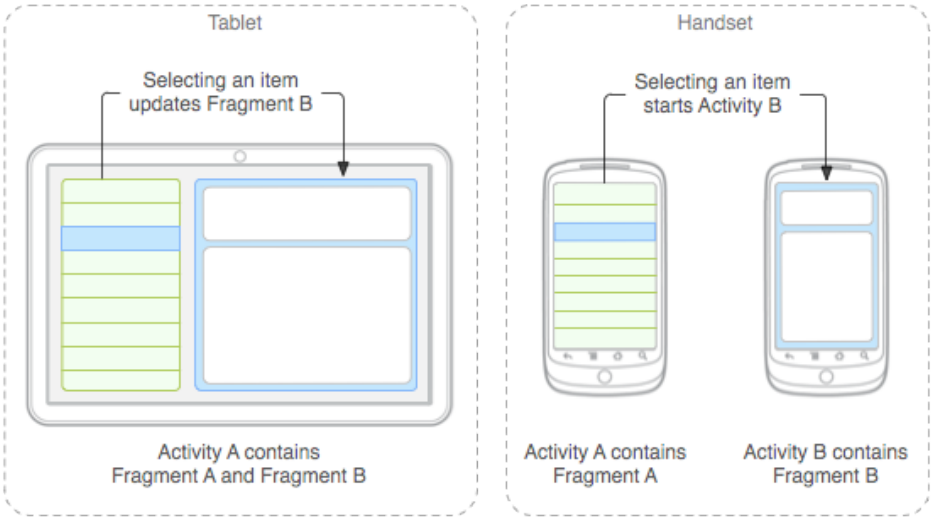


Figure 4.4: Fragment [13]

be integrated into other Activities or Fragments as illustrated in Figure 4.4. The advantage of Fragments is that it allows reusable UI components to be created. The Fragments have their own life cycles, which strongly resembles that of the Activity with an addition of some extra function calls that allow the developer to customize the Fragment prior to and after being attached to the Activity.

4.3.5 SQLite

SQLite[1] is an engine that provides persistence of data in a relational database. This component can be used for efficient storing, accessing and porting of data, if it is to be used for post processing on a more powerful device such as a PC, or if it should be available for processing by other apps. Several softwares exist that can access data available in SQLite databases, such as SQLite Manager [35], which enables the use of SQL language to query and extract information from the database.

4.4 App development with Android

Androids official web page for developers [48] offers two *integrated development environments* (IDE's), the Eclipse[60] IDE with the *android development tools* [37] (ADT) plugin and the Android Studio IDE. Eclipse is a well-established IDE for developing java applications and is an open source solution for automated building, running and debugging of source code. It provides tools for refactoring and the ability to extend its functionality through plugins. One such is the ADT plugin. ADT extends Eclipse with an integrated environment where Android apps can be built, debugged, tested and exported to signed .apk (Android application package) files. Such files are ready to be distributed to smart phones

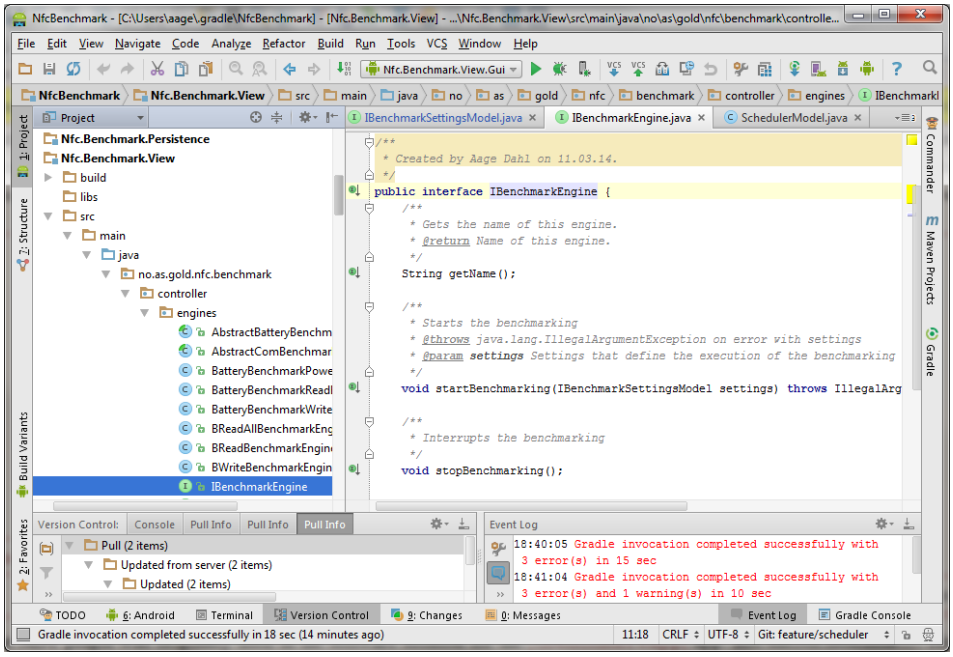


Figure 4.5: Android Studio IDE

running the Android operating system [47]. Furthermore, it provides an Android Layout file editor with preview functionality that enables interactive user interface (UI) development. It also provides a range of other abilities that are not further discussed in this work, but for the interested party, this information can be found at androids official developer site [25].

Android Studio is another IDE that is built on top of the InteliJ IDEA, which provides much of the functionality that Eclipse does with the difference of a slightly more extensive refactoring ability. Additionally, Android Studio uses Gradle for building solutions and project management. This framework provides an efficient way of handling multi project solutions. Solutions can be divided into modules where each module can contain any number of projects. In larger projects, where libraries are built to support the application functionality, it can be a handy feature to collect these libraries in modules that link them to the main solution. This means that these library projects can be debugged and extended simultaneously as working on the main solution. Additionally, in case of flexible libraries, these projects can be included into other solutions as well. The GUI part of this IDE is displayed in Figure 4.5.

Eclipse with the ADT plugin (depicted in Figure 4.6) was originally used in the software solution for the NFC benchmark app 6.4. However, as the development progressed and it was identified that a second app with a lot of common features were needed, it became clear that the Android Studio IDE with the Gradle build system was an advantage due to its modular design pattern. This allows modules to be shared between the two apps and thereby reduce development effort and improve consistency between

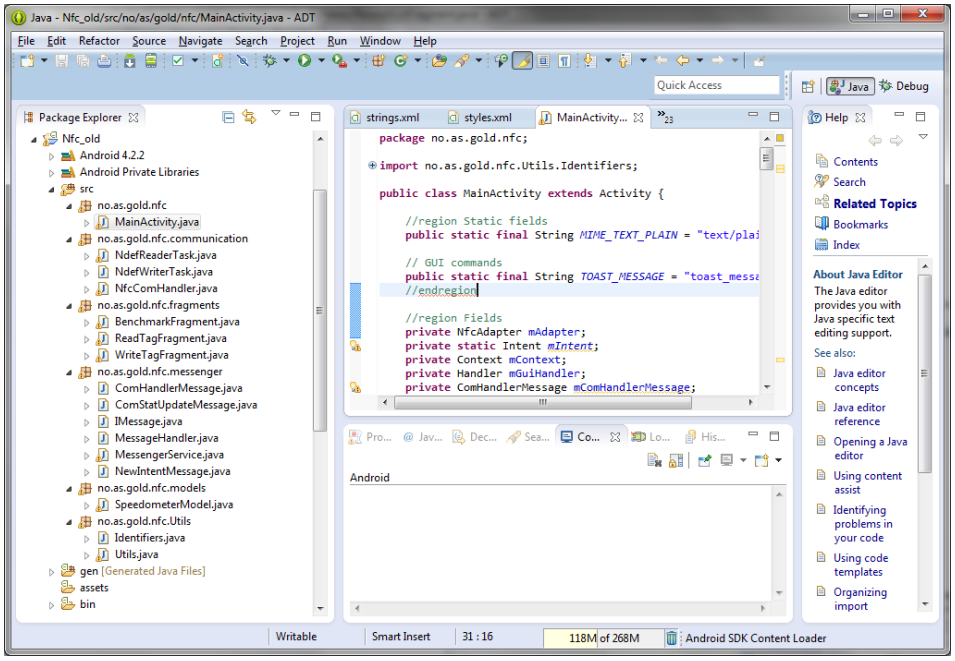


Figure 4.6: Eclipse IDE

the apps. It is also recognized that Android Studio is an IDE that is specifically tailored to develop Android apps. A challenge that is identified with this IDE is that it is in a pre-beta stage, and a range of difficulties were encountered as the project progressed. The main difficulty being the lack of information available to assist in resolving bugs that existed in the IDE. For example, the Android Studio 0.4.2 supposedly has support for building and deploying apps with java libraries. However, as two of the libraries in this solution are of this type, it was quickly identified that although the solution compiles and deploys, the app encounters an exception when accessing parts of the functionality that uses the java library modules. This is a bug that is introduced in one of the version between 0.4.0 and 0.4.2, and it therefore works correctly with Android Studio 0.4.0. The result is that only versions existing before 0.4.0 can be used. Other issues are related to importing of projects that required some manual handling.

Chapter 5

Design patterns

To retain a structure in the developed software and to decouple the view from the implementation, the *Model-View-Controller* (MVC) [22] pattern is used in the software developed for this work. This helps to maintain the code as the solution grows and more projects are added. Alternatives to this pattern exist, such as *model-view-presenter* (MVP) where all data between the view and the model goes through the presenter. This pattern puts the presentation logic into the presenter, which also handles user interaction logic. The MVC pattern is chosen because it clearly separates the user interface from the business logic.

The singleton design [22] is used in certain utility tools to ensure that all objects refer to the same instance of the tool. The factory pattern is used to have a single source of mapping objects to interfaces so that the construction of objects is handled in a single factory.

5.1 MVC pattern

The MVC pattern is used to decouple the user interface from the underlying business logic. This allows the interface to be easily changed without requiring modifications to the business logic. The pattern consists of three components: the Model, the View and the Controller. The controller handles user requests such as when the user presses a button. The model models the data that the user is working with and ensures data integrity, *i.e.* makes sure that changes to the data follow given rules. The view merely presents the data through a *graphical user interface* (GUI).

When the user requests some data, this request is handled by the controller. The controller calls on the correct functions in the model, which in turn executes the action and updates the view with the results. This is illustrated in figure 5.1

5.2 Singleton pattern

The singleton pattern ensures that only one instance of a class is in use at any given point in time in an application. This is useful when it is important

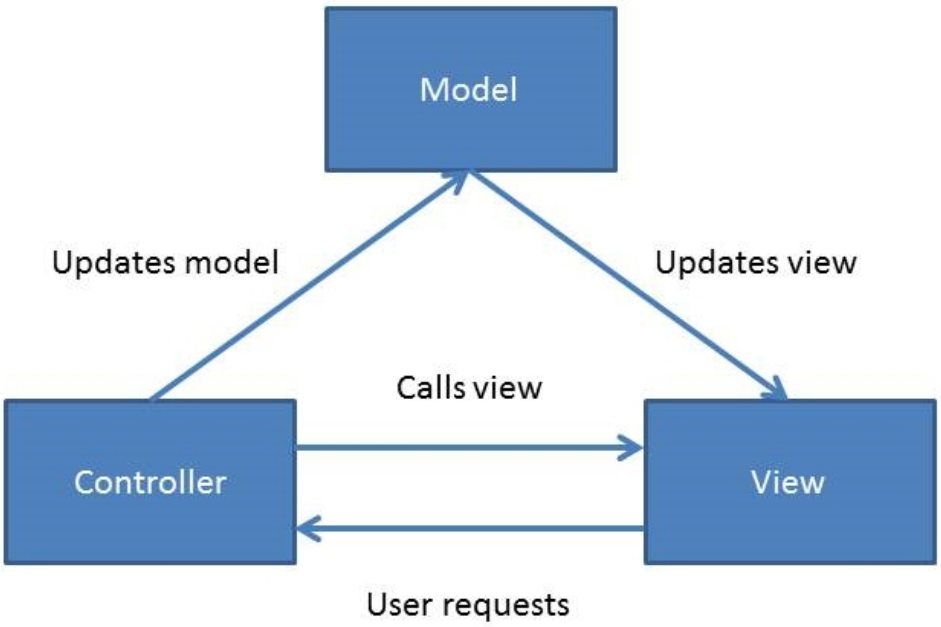


Figure 5.1: MVC pattern

that the same instance is used throughout the application.

5.3 Factory pattern

The factory pattern defines a factory that is used to create other objects. The objects created by the factory can be represented by an interface and therefore allows the actual object class to be changed without having to alter the classes that are using these objects.

5.4 Wrapper pattern

The wrapper pattern is used to allow one interface to be used from another. It is used to map functions existing in one class to a desired interface without needing to change the source code in the original class.

Part II

Design and implementation

Chapter 6

App design

This chapter describes the design of the software developed for this thesis. It starts with introducing the requirements for the apps and continues with decomposing these into objectives. At the end, it presents the architecture of the software.

6.1 Requirements

The software developed in this work is required to communicate with a range of different tags, extract data related to the tags communication performance and test proprietary tag protocols. These requirements can be summarized in the following three higher level objectives:

- to develop a standalone solution that provides a single interface to communicate with a range of tags, and
- to develop software for benchmarking tags, and
- to develop software for testing proprietary tag communication protocols that are based on one of the standards described in Chapter 3.

The standalone solution is required to provide communication access to tags. It must be able to provide developers of any apps with this access.

The software for benchmarking tags must provide a user interface that enables the user to monitor and collect data about communication performance of tags. The data collected must show how reading and writing operations affect the battery and what throughputs are accomplished with a given tag. The throughput is defined as the number of bytes successfully transferred, and the throughput is therefore reduced when corrupted bytes are transferred. The resulting data must be presented to the user.

The communication protocol testing software must provide the user with an interface for transmitting arrays of bytes to the tag and monitoring the response. It must also allow the user to read arrays of bytes from specific locations in a tags memory and writing arrays of bytes to a tags memory.

6.2 Requirements decomposition

This section describes how the requirements are decomposed into two apps and a library, as well as the basic design decisions that are taken. The first objective requires a standalone solution that other solutions can build upon and therefore needs to be considered as a component with no dependencies to solutions that use it.

The second and third objectives simply demands two features that can be provided through one or two apps. The two apps solution separates the features into two apps and thereby obtains a clear distinction between the two. Alternatively it can be provided through a single app, which initially enables the user to choose which features of the app to use. This solution requires more functionality to be added to a single app, inherently making it more complex. However, it also provides the user with a single access point for both features, which makes them more readily accessible. I have chosen to use the solution with two apps. The main reasoning behind this decision is that both apps are using the standalone solution to provide communication access to the tags, which ensures that the standalone solution is flexible enough to be used for multiple purposes.

6.2.1 Tag communication

The first objective requires a standalone solution for communicating with tags. Two alternatives to accomplish this objective in an Android environment is to develop a library that enables this communication and resides inside the apps address space, or to develop an Android Service that is registered in the Binder of the Android Kernel. A Service has very loose coupling and allows any app with the correct privileges to use it. It also has control over the number of apps that are using its services. These are desirable features, which makes the component more user friendly than the library. But it also has a downside, information transmitted between them need to be serializable, which requires extra implementation effort.

The library version resides in the apps address space and is therefore directly accessible through the library's API. In contrast to the Service, objects transmitted between the library and other components in the app, do not have to be serializable. On the downside, the library does not know if any other instances of itself are currently running. This can create issues where more than one instance tries to communicate with the tag simultaneously, which might result in unexpected behavior.

For the current project, I have chosen to work with the library solution because of the reduced implementation effort and because of the unlikelihood of more than one instance of the library running at the same time. It is recognized that the library solution has been designed so that it easily can be migrated to a Service solution. This is discussed in more detail in the Chapter 11.

6.2.2 Tag benchmarking

The second objective aims at benchmarking the tags, and this feature is provided in the NFC benchmark app. The app monitors and collects data about communication performance and energy consumption related to reading from and writing to a tags memory. It also estimates the round trip time for data communication and the energy consumed while powering a tag without performing any I/O operations.

The collection of data can be done through one single procedure that monitors and collects all the desired data. This makes it simple to handle the procedure's lifecycle, as there is only one benchmark to handle. However, it also adds some complexity to the benchmark and therefore makes it more difficult to extend with monitoring of new data. Alternatively, one benchmarking procedure can be created for each communication feature to be monitored. If the benchmarking procedures follow a certain pattern, they can be controlled in a generic way, and the solution can be designed to expand with an undefined number of benchmarks. I have chosen to follow the second suggestion and create a design where one benchmark exists for every communication feature.

Interaction with benchmarking procedures can be accomplished in several ways. A procedure can be accessed in a single view, where the GUI is tailored to the needs of the benchmark and controls the benchmark lifecycle. Alternatively, all benchmarks can be selected from a single view, where the GUI is generic enough to provide configuration of all benchmarks. The first alternative provides more flexibility to the given benchmark and the possibility of a more tailored GUI, but it also requires that a single benchmark is completed before another is initiated. The second alternative enables automation of the run of multiple consecutive benchmark procedures. This increases the complexity of the app, which in its turn increases the implementation effort. However, the user friendliness is drastically improved as each benchmark can take from minutes to hours depending on tag type and settings. Therefore, the opportunity to run them in concession without requiring users to monitor for the completion of each individual benchmark, is a time-saver. Additionally, if the tests are extensive, it might be beneficial to allow the app to run throughout the night when the user is unavailable. I have chosen the second alternative, where the user can select benchmarks from a single view to run consecutively. The choice is made because the user can then collect more data with less interaction.

Data created by monitoring the communication must be accessible by the user. This can be provided either in the GUI or in the form of a file. The GUI renders the information available for instant evaluation by the user. However it makes it more difficult for the user to do any further analysis. Additionally, the data that is collected in this app can consist of millions of samples, and it is therefore difficult to present this data in a meaningful way. By exporting the data to a file, it is possible to run post processing routines to analyze the information. I choose to export the data to a file, because it makes the data readily available for post processing.

Two ways of storing the data to a file has been assessed in this work, through the use of a SQLite database or by storing it in a XML file. The XML file is simple to create, but requires solutions to be developed for extracting and analyzing the data. The SQLite database is a piece of software that provides a SQL database engine. It provides simple and effective extraction, insertion and analysis of information from the database with the use of SQL queries. I choose to use the SQLite database solution for persisting the data, because of the extra functionality it provides, and because it makes the data more readily accessible for other applications and users.

6.2.3 Tag protocol testing

The third objective aims at testing protocols, and this feature is provided in the NFC protocol tester app. The app enables users to transmit collections of bytes to a tag and observe the tags response. It also allows the user to write byte sequences to and read byte sequences from the tags memory. The user is able to modify the bytes transmitted to the tag so that it can resemble any given proprietary communication protocol as long as the tag is based on one of the standards described in Chapter 3.

6.3 Overview

This section briefly introduces the apps that are developed in this work. The introduction for each app describes how users interact with them and how the apps respond.

The NFC benchmark app allows the user to run a set of benchmark tasks that are designed to collect performance metrics from the tag. These performance metrics measure the communication and battery performance related to different tag operations and stores the results in a database. The settings supplied with the testing enables the user to customize what benchmarking tasks to run and how they will run, *e.g.* how granular the tests will be. The user interaction with this app is illustrated in Figure 6.1. As it shows, the user configures the settings for the app, and when the tests are run, data is collected and transmitted to a database called *BenchmarkResults.db*. This database is of the type SQLite[1] and is be further described in Section 6.4.

The NFC protocol tester app allows the user to perform three basic operations on the tag: reading, writing and transmitting custom commands. These commands produce some response from the tag, which is presented to the user. The operations of this app are illustrated in Figure 6.2.

6.4 NFC benchmark app functionality and UI

[1]SQLite is a software that implements a SQL database engine

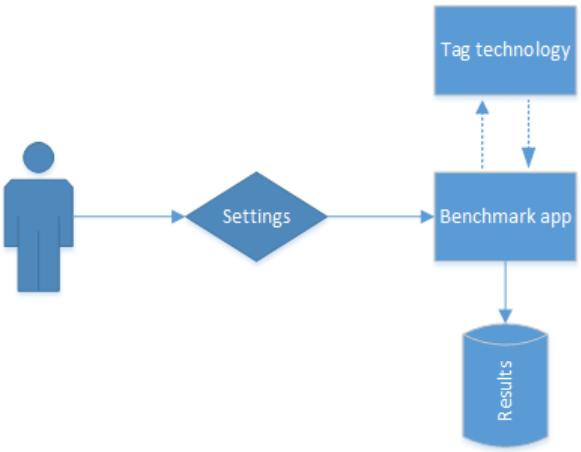


Figure 6.1: Basic NFC benchmark app architecture

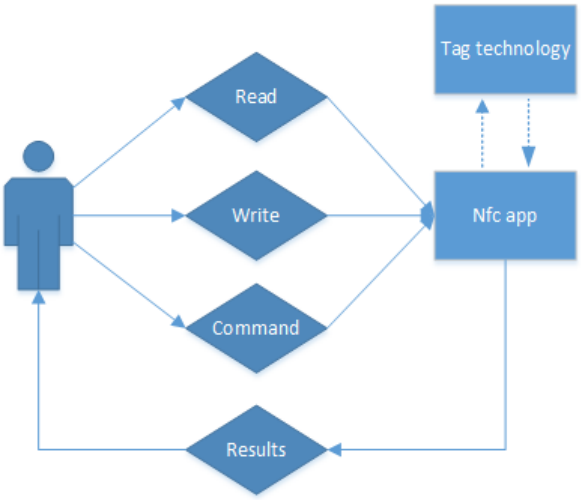


Figure 6.2: Basic NFC protocol tester app architecture

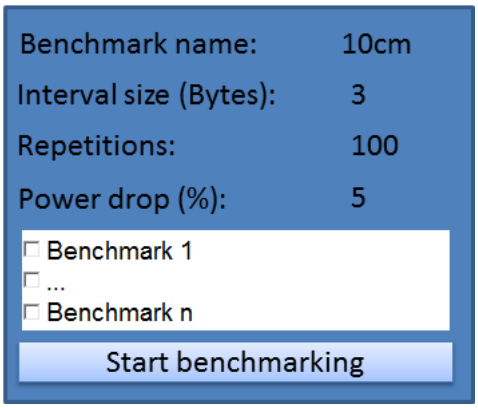


Figure 6.3: Benchmark Fragment mockup

The NFC benchmark app hides the details relating to the tags memory structure and protocol selection, giving the user an abstracted view of the tag. The different benchmarks available are described in Table 6.1. To add some flexibility to the benchmarking procedures, the user can configure some settings that change the behavior of the benchmarks. Configuration of the settings are available through the same view that allows the user to start the benchmarking procedures. Figure 6.3 shows a mockup of this view, illustrating input fields for benchmark selection and parameter configuration.

Four parameters are supplied with the settings: the *Benchmark name*, the *Interval size*, *Repetitions* and *Power drop*. The *Benchmark name* allows the user to tag the collected data with a description of the run. The *Interval size* and the *Repetitions* configurations are used for benchmarking procedures related to the communication performance, and the *Power drop* parameter is used for benchmarking procedures related to the battery performance.

6.4.1 Benchmark tests

Table 6.1 summarizes the seven benchmark tests that the user is able to run with the NFC benchmark app. These are described in the coming paragraphs. Additionally, this app offers a selection of other benchmark tests that can be started and run through separate views. These tests include communication testing for NDEF messages, and results from these are displayed on screen instead of being persisted in a database.

The *Write binary* and *Read binary* benchmarks generate and transmit random data stored in packets to and from the tag. The size of these packets start at 1 byte and expands with an interval of *Interval size* until reaching the full span of the tags memory. For each packet size, the transfer is repeated *Repetitions* times, and for each repetition, a sample of the communication performance is created.

The *ReadAll binary* benchmark reads the full memory of the tag and measures the performance as a sample that is stored in a database. This procedure is repeated *Repetitions* times.

The *RTT* benchmark transfers a minimal command to a tag and measures how quickly it responds. The measurement is stored as a sample in a database. This procedure is repeated *Repetitions* times.

Name	Description	Input	Output
Battery read	Measures battery consumption for reading tag.	n	Time taken until battery level is reduced by n%
Battery write	Measures battery consumption for writing to tag.	n	Time taken until battery level is reduced by n%
Battery powering	Measures battery consumption for powering tag.	n	Time taken until battery level is reduced by n%
RTT	Calculates round trip time.	Repetitions r	r RTT samples
Write binary	Calculates write communication performance.	Interval size i, repetitions r	Status, size and throughput for each sample
Read binary	Calculates read communication performance.	Interval size i, repetitions r	Status, size and throughput for each sample
ReadAll binary	Calculates readAll communication performance.	Repetitions r	Duration and size for each sample

Table 6.1: Benchmarking tests overview

The *Battery read*, *Battery write* and *Battery power* benchmarks performs read, write and power operations respectively until the energy level of the battery is dropped with *Power drop%*. The power operation consists of inductively powering the tag without transferring or receiving any commands from it.

6.4.2 Persistence

Once the information from the benchmarking has been collected, it is necessary to store this information in a structured way. This is accomplished through a database so that the data is easily accessible for other applications for further analysis. Each entry in the database stores information about when the data is recorded, the user defined name, tag type etc. The database consists of seven tables, one for each benchmark described in Table 6.1. The tables, including attributes are listed in Listing 6.1. Each attribute is described in Table 6.2. The database is stored at the following location in the phones filesystem: *storage/sdcardo/Nfc.Benchmark/BenchmarkResults.db*.

Listing 6.1: Creation of tables for benchmark database

```
CREATE TABLE ReadBatteryPerformance(id INTEGER PRIMARY
    KEY,time TEXT,name TEXT,technology TEXT,startLevel
    FLOAT,endLevel FLOAT,elapsedTime FLOAT,bytesTransferred
```

Attribute	Description
id	Unique id for Table entry
time	Time the sample is recorded
technology	Technology the sample is associated with
startLevel	Battery level at start of sample
endLevel	Battery level at end of sample
elapsedTime	Time elapsed
bytesTransferred	Number of bytes transferred during sample
RTT	Round trip time
size	Size of packet transferred
duration	Time taken for transferring packet
status	Status of transfer; *Succeed*, *Fail* or *CommunicationError*

Table 6.2: Attributes description

```
    INTEGER);
CREATE TABLE WriteBatteryPerformance(id INTEGER PRIMARY
    KEY,time TEXT,name TEXT,technology TEXT,startLevel
    FLOAT,endLevel FLOAT,elapsedTime FLOAT,bytesTransferred
    INTEGER);
CREATE TABLE PowerBatteryPerformance(id INTEGER PRIMARY
    KEY,time TEXT,name TEXT,technology TEXT,startLevel
    FLOAT,endLevel FLOAT,elapsedTime FLOAT,bytesTransferred
    INTEGER);
CREATE TABLE RTT(id INTEGER PRIMARY KEY,time TEXT,name
    TEXT,technology TEXT,RTT FLOAT);
CREATE TABLE WriteCommunicationPerformance(id INTEGER
    PRIMARY KEY,time TEXT,name TEXT,technology TEXT,size
    INTEGER,duration FLOAT,status TEXT);
CREATE TABLE ReadCommunicationPerformance(id INTEGER
    PRIMARY KEY,time TEXT,name TEXT,technology TEXT,size
    INTEGER,duration FLOAT,status TEXT);
CREATE TABLE ReadAllCommunicationPerformance(id INTEGER
    PRIMARY KEY,time TEXT,name TEXT,technology TEXT,size
    INTEGER,duration FLOAT,status TEXT);
```

6.5 NFC protocol tester app functionality and UI

The NFC protocol tester app is required to supply three functionalities, which can be provided in a single view. Advantages of this solution are that it is simple to make and does not provide any effort with maintaining multiple views. The disadvantages are that a lot of UI components need to be fitted into a small area, which can make it more difficult to use and which makes the code more difficult to manage. An alternative is to

have one Activity to handle each functionality and a start-up screen that allows the desired Activity to be chosen. This separates the functionalities making them easier to manage. However, it does not provide a simple way of switching between each functionality. Another alternative is to use a single Activity with multiple Fragments, where the Fragments provide the views. These can be handled by a ViewPager[2] that allows the user to use the "swipe" gesture to navigate between the views. I choose to use the ViewPager alternative with a set of swipe views due to its simple and intuitive navigation system.

The NFC protocol tester app consists of a set of swipe views, where each view enables the user to configure and perform one of its operations. The read operations allow the user to select a memory address to read from and the number of bytes to read. Assuming that the command is successfully executed, the app presents the results to the user. A mockup of this view is presented in Figure 6.4a.

The write operation allows the user to select a memory address and an array of bytes to write to this memory address on the tag. The app responds with information about the operations success or failure. A mockup of this view is shown in Figure 6.4b.

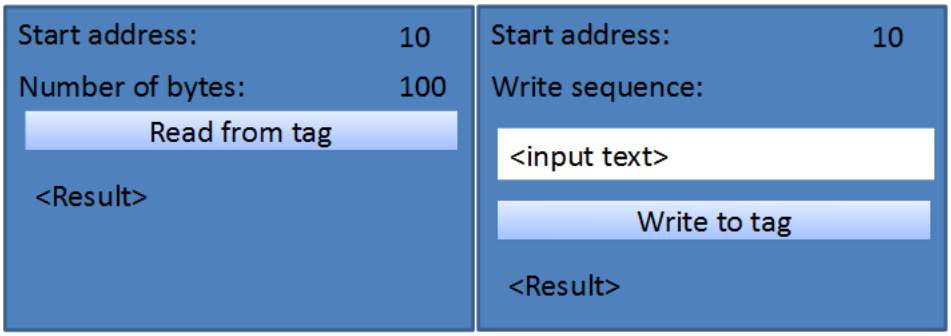
Finally, the operation for transmitting a command to the tag enables the user to produce an array of bytes to transmit to the tag. This array is wrapped with standard SOD[3] and EOD [4] data in accordance with the protocol related to the given tag type. The app presents the user with the tags response to the command. A mockup of this view is presented in Figure 6.4c.

6.6 Architecture

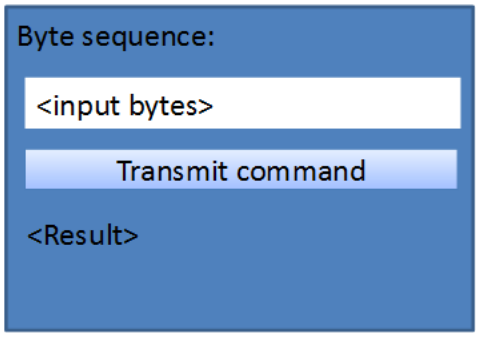
The two apps developed in this work have some similar functionality requirements such as those that require them to communicate with tags. These functionalities can either be shared between the apps or they can each implement them independently of each other. To share the functionality requires some overhead, because the functionality needs to be extracted to a library that is generic enough to cover the functional needs of both apps. The functional requirements are rarely identical between apps, and it is therefore often more effective to develop two solutions that are tailored to the specific needs of the given app. However, this requires up to twice the implementation effort. Additionally, functionality that is shared is more rigorously tested when used by two apps. Because of this, I choose to extracted some functionality into libraries that are shared between the apps developed.

Android Studio offers three levels of organizing the source code: in packages, in projects and in modules. Packages generally structure classes within a project, and to use these classes, the whole project needs to be

[2]Belongs to the android.support.v4.view package
[3]Start Of Data is a byte specifying the length of the command
[4]End Of Data is the CRC calculated on the bytes in the command



(a) Read Fragment mockup (b) Write Fragment mockup



(c) Command Fragment mockup

Figure 6.4: NFC protocol tester mockups

referenced. All the components of an app can be included inside a single project and be structured into packages. This makes it easy to navigate between classes, but it makes it less efficient to share functionality between solutions, because the whole project needs to be referenced as a library. Alternatively, blocks of functionality can be extracted to library projects. The advantage of this is that the functionality supplied by the project can easily be shared between projects and solutions. Finally, modules can be used, these group projects that offer some functionality to solutions. I have chosen to use all these organizational methods in my solution. Packages are used for structuring classes inside projects, and modules are used to group projects into large functional blocks. This is done to maintain a rigid structure in the source code and to separate blocks of code so that they can efficiently be reused.

The NFC benchmark and NFC protocol tester apps consist of six modules each. These modules have one or more stand-alone projects with dataflow as depicted by the arrows in Figure 6.5. Two of the modules are the same for both apps and contain code for handling overlapping functionality such as communicating with the different tag types. Figure 6.5 illustrates the module structure for the two apps.

Subsections 6.6.4 and 6.6.3 introduce and discuss the modules that are exclusive for the NFC protocol tester app NFC benchmark app respectively. Subsections 6.6.1 and 6.6.2 do the same for the *SimpleMessenger* and

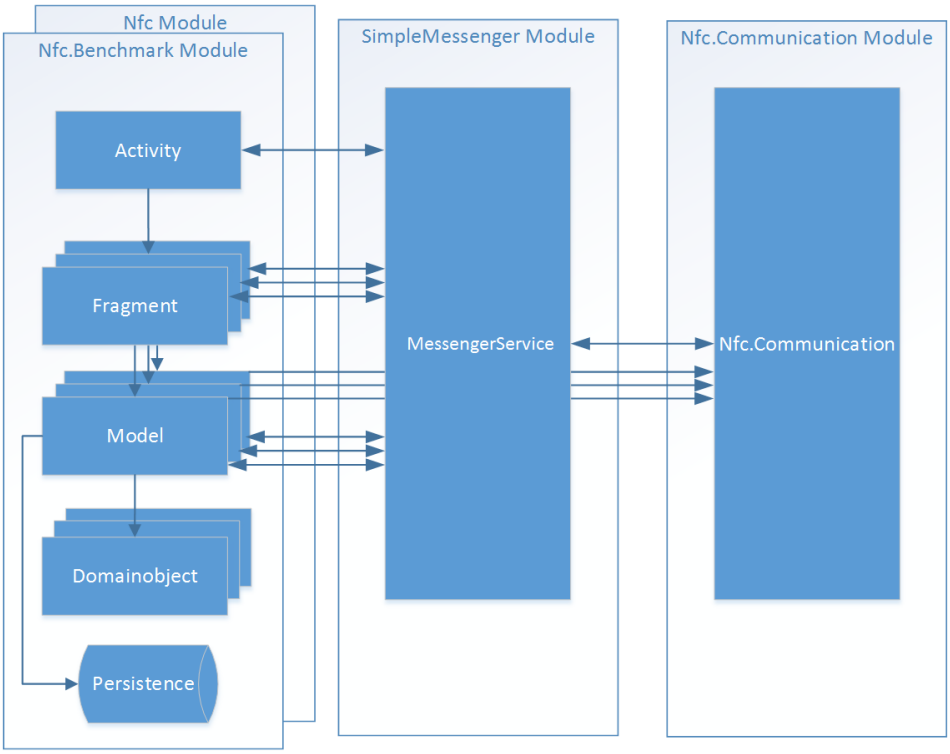


Figure 6.5: Modules

Nfc.Communication modules.

6.6.1 Messaging system module

Android components, such as Fragments and Activities, are very decoupled, which makes sharing of information a challenge. In Androids official developer guide, it is advised to have the Fragments implementing interfaces that allow the Activity to respond to events generated by the Fragments [49]. This has the disadvantage that Fragments require an Activity to relay the information if Fragments need to communicate. It also means that the Activity needs to implement code to handle events from all of its Fragments. In this work, there are several Fragments associated with one single Activity and implementing handlers for all the events generated by the Fragments makes the implementation very complex.

An alternative is to use Android's built in messaging system with Intents to enable communication. This removes dependencies between the sender and the receiver of the Intents. However it also requires that the messages are serializable. Sometimes objects are transmitted between senders and receivers, and the serialization requirement adds extra implementation effort.

Another alternative is to develop an event driven messaging system where Fragments and other objects pass messages to the system, and the system makes sure that the messages are transmitted to the designated receivers. In this system, senders are unaware of who the receivers are, which makes them loosely coupled. Any object wanting to receive messages need to register in this system. The advantage of this solution is that multiple receivers can be registered for handling a message without needing to make any changes to the sender.

I choose to follow the third solution, because it reduces dependencies between senders and receivers, and because it does not require serialization. The event driven messaging solution that is designed and implemented is depicted in Figure 6.6. This is a framework that enables any object to transmit a message that implements the *IMessage*[5] interface when an event occurs. Other objects can register a callback function that is executed with the message as a parameter, whenever such a message is transmitted. By utilizing this functionality, an object can send a message when it has information to share and any object can receive and handle it.

Interaction with the UI is in Android handled by the UI thread. Blocking on this thread renders the UI unresponsive and hinders a good user experience. Even more, if the UI thread is blocked for more than a few seconds, it responds with an "application not responding" dialogue [50]. It is therefore important that heavy workloads and blocking processes are delegated to worker threads and that the UI thread is simply used for updating the UI. In the two apps developed for this work, the SimpleMessenger framework is used to enable worker threads to execute

[5]Interface defined in the *MessengerService* module, belongs to the *no.as.gold.utils.simplemessenger* package

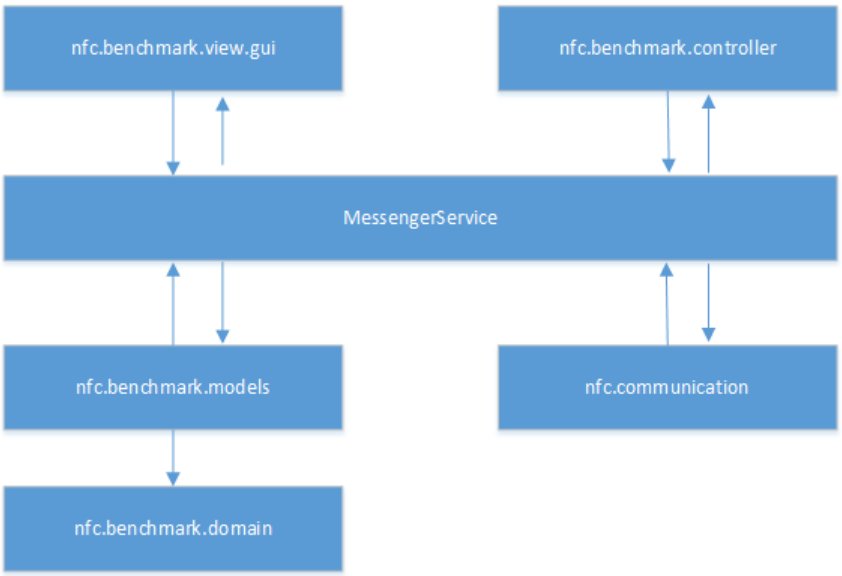


Figure 6.6: MessengerService

processes in the background and then inform the UI thread of the result when they are finished.

6.6.2 NFC communication library module

When communicating with the NFC device, the caller thread is blocked until the call returns with a response form the Android operating system. It is therefore imperative to create a multi-threaded system, where time consuming tasks are performed asynchronously by worker threads. The NfcComHandler library therefore offers functions that allow the object to communicate with the tags asynchronously. The function calls are non-blocking and start background threads that perform the communication. The results are published to the *MessengerService* framework by sending an instance of the *Nfc.Communication.ComHandlerMessage* class. This object contains basic information about what is sent, received and if there was some problems with the communication. The information contained in this message can be received by any object that chooses to do so. By registering a handler in MessengerService that is paired with the ComHandlerMessage class, the message is received and the handler is executed. This multi-threaded environment can result in multiple threads accessing the *Nfc.Communication* classes, thus, to distinguish between the responses, a UUID object can be paired with the message on creation. When a UUID is given to the message being sent, the Nfc.Communication framework sets the UUID in the responding ComHandlerMessage to the given UUID, and the receiver can thereby filter for this message. Figure 6.7 gives an example of an object using the NfcComHandler class with the MessengerService framework to communicate with a tag.

It is important for the *NFC benchmark* app that the behavior of the

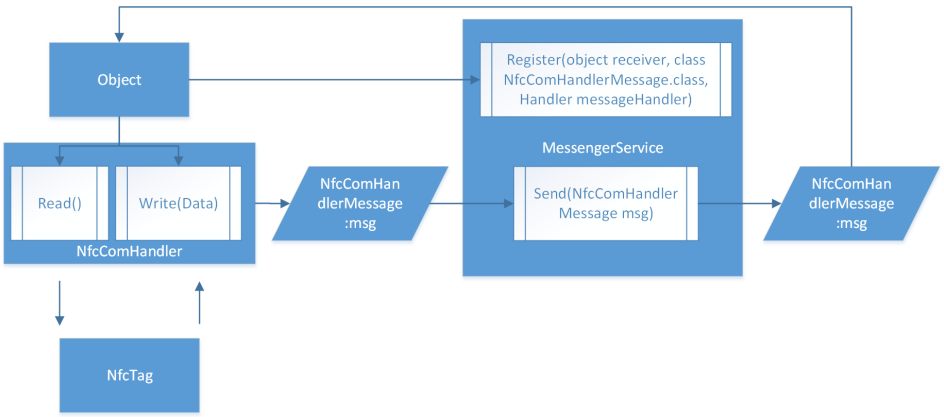


Figure 6.7: NFC communication

communication can be monitored. This is accomplished by having an object with a worker thread that continuously initiates communication with the tag and waits for a response before repeating the process with possibly a changed parameter. Simultaneously, another object can register interest for the same message type and simply record statistics related to the communication performance.

6.6.3 NFC benchmark app modules

This subsection introduce the modules that are exclusive for the NFC benchmark app.

6.6.3.1 Activity module

The *Nfc.Benchmark.View.Gui* is the project that contains the main Activity of the app. The project has two responsibilities: to render the different benchmarking operations for the user and to display messages to the user. To accomplish this, the project is dependent on two other projects, *Nfc.Benchmark.View* and *SimpleMessenger*. *Nfc.Benchmark.View* contains the Fragments to be rendered, and *SimpleMessenger* is the channel through which Fragments send messages that should be rendered for the user.

6.6.3.2 Fragments module

The structure in this module follows the MVC pattern, where the classes in the *Controller* package and *Model* package and the Layouts associated with each class in the *Controller* package, has the role of the Controller, Model and View respectively.

The *Nfc.Benchmark.View* module contains a set of Fragments for performing benchmarking operations and for configuration purposes in addition to one benchmark Fragment for running sets of consecutive benchmarks. This module has five packages that are described in the following paragraphs.

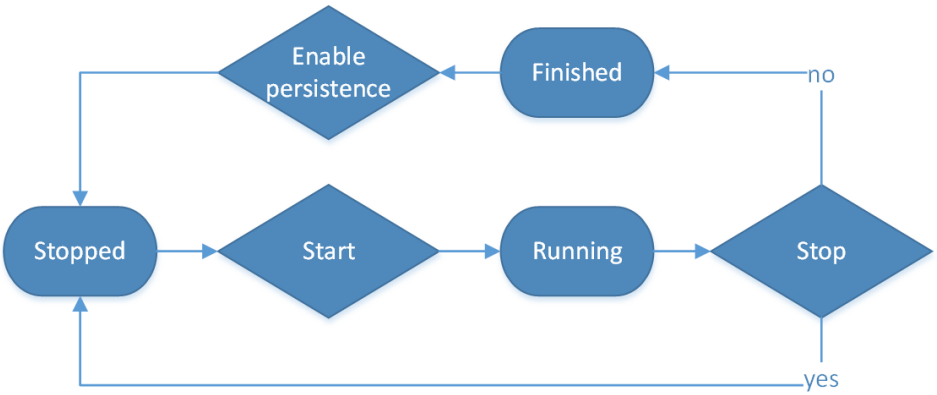


Figure 6.8: Engine lifecycle

Controller classes reside in the *controller*[6] package and extend the *Fragment*[7] class. These attach to the user interface and handle user interaction with UI components. Most of these Fragments handle the lifetime of a benchmarking procedure and therefore have one or more *IBenchmarkEngines*[8]. These Fragments fulfill the Controller part of the MVC pattern.

Engine classes reside in the *engines*[9] package. These implement the *IBenchmarkEngine*[10] interface, which controls the lifecycle of the benchmark procedures and enables starting, stopping and persisting the benchmarking. The lifecycle is illustrated in Figure 4.3. The engine is initially stopped. When the engine is started, it proceeds to the running state where benchmarking activities are performed. At any time during running, the engine can be stopped, at which point the engine goes to the stopped state. If the engine is not stopped and completes its run, it enables persistence of the data and proceeds to the stopped state.

Message classes reside in the *messages*[11] package. These classes implement the IMessage interface and are used with the SimpleMessenger framework to communicate between Fragments in the *Nfc.Benchmark* module.

Model classes reside in the *models*[12] package. Each model is linked to one or more Fragments and will maintain user data and make sure that

[6] Full package name: *no.as.gold.nfc.benchmark.controller*
[7] Belongs to the *android.app* package
[8] Interface for a benchmark engine, residing in the no.as.gold.nfc.benchmark.controller.engines package
[9] Full package name: *no.as.gold.nfc.benchmark.controller.engine*
[10] Resides in the *no.as.gold.nfc.benchmark.controller.engines* package
[11] Full package name: *no.as.gold.nfc.benchmark.controller.messages*
[12] Full package name: *no.as.gold.nfc.benchmark.models*

Fragments are updated with changes. These classes fulfill the task of the model in the MVC pattern.

6.6.3.3 Domain module

The *Nfc.Benchmark.Domain* module has objects designed for storing the results of the benchmarking activities. These objects are persisted in a database.

6.6.3.4 Persistence module

The *DatabaseHandler* class resides in the *Nfc.Benchmark.Persistence* module. This class handles creation, updating and maintaining of a SQLite database that persists samples from benchmarking engines.

6.6.4 NFC protocol tester modules

This subsection introduces the modules that are exclusive for the NFC protocol tester app.

6.6.4.1 Activity module

The *Nfc.View.Gui* is the project that contains the main Activity of the app. The project has two objectives: to render the GUI and to display messages to the user. To accomplish this, the project has dependencies to two other projects, *Nfc.View* and *SimpleMessenger*. *Nfc.View* contains the Fragments to be rendered and *SimpleMessenger* is the channel through which Fragments send messages to the UI thread.

6.6.4.2 Fragments module

The *Nfc.View* module contains the majority of the app's functionality. It consists of a collection of Fragments that enable the user to interact with the tag. This project is dependent on *Nfc.Communication* to interact with the tag and is dependent on *Nfc.Domain* to maintain objects produced by the interaction.

The module is comprised of two packages: the *Controller*[13] and the *Models*[14]. The *Controller* package contains code for handling the Fragment inflation and mapping user inputs to model classes in the *Models* package. It contains three Fragments that each accomplishes the objectives described in Subsection 6.5: reading from the tag, writing to the tag and sending user defined commands to the tag.

The *Models* package contains the information about the user inputs and updates the Fragment when new information arrives. One model exists for each Fragment in the *Controller* package.

[13]Full package name: no.as.gold.nfc.controller
[14]Full package name: no.as.gold.nfc.models

6.6.4.3 Domain module

The *Nfc.Domain* module contains the domain objects for the NFC protocol tester app.

6.6.4.4 Persistence module

The *Nfc.Persistence* module contains classes that handle persistence of domain objects. Since there are no objects to persist in the NFC protocol tester app, this module does not contain any functionality and is only created to maintain a consistent structure of the app. If data collected through this app should be persisted in future works, it should be done through this module.

Chapter 7

App implementation

This chapter presents the implementation of the two apps that are developed for benchmarking of NFC tags and for testing proprietary NFC communication protocols. The apps are comprised of four key components each and two shared components, which are discussed in more detail throughout the coming sections. Only the most important components and classes are discussed. There exist other utility tools and convenience classes that are not touched as these are assumed to be self-explanatory and no deeper knowledge is required to understand their use.

The source code for the software described in this chapter, can be located in the following repositories:

- Messaging system [8]
- NFC communication library [9]
- NFC protocol tester app [10]
- NFC benchmark app [11]

7.1 Messaging system

MessengerService is a singleton class that resides within the SimpleMessenger module and allows processes to broadcast messages to any interested object without having a direct reference to them. This is done by messages being divided into a set of classes and allowing objects to register callback functions to handle a given message type. Registration is done with the function presented in Listing 7.1.

Listing 7.1: Register callback

```
1 /**
2  * Registers interest for a certain type of messages
3  * @param key: Identification of the handler
4  * @param messageHandler: Callback function when receiving a message
       of the given class type
5  * @param messageClass: Type of message to listen for
```

```
 */
public synchronized void Register(Object key,
                                  Class<?> messageClass,
                                      MessageHandler messageHandler)
                                      {

    // Initiate new hashmap for new recipient
    if(!mHandlers.containsKey(key)) {
        ConcurrentHashMap<Class<?>, ArrayList<MessageHandler>>
            hashMap = new ConcurrentHashMap<>();
        mHandlers.put(key, hashMap);
    }

    ConcurrentHashMap<Class<?>, ArrayList<MessageHandler>>
        hashMap = mHandlers.get(key);
    if(!hashMap.containsKey(messageClass)) {
        ArrayList<MessageHandler> handlers = new ArrayList<>();
        hashMap.put(messageClass, handlers);
    }

    mHandlers.get(key).get(messageClass).add(messageHandler);
}
```

Each registration requires a key, which exists so that the handler later can be removed if desired. The callback function implements the interface in Listing 6.6. A limitation of this implementation is that only one callback function can be registered for the combination of one key and one message class.

Listing 7.2: MessageHandler interface

```
/**
 * This is the interface of messages that can be sent by the {@link
      no.as.gold.simplemessenger.MessengerService}
 * Created by Aage Dahl on 06.12.13.
 **/
public interface IMessage {
    public String getMessage();
}
```

When a process sends a message, MessengerService executes the callback functions that are registered for this message type as indicated in Listing 7.3. This component has a close resemblance to the way Intents work in Android, only the communication is synchronous, and it cannot communicate between apps. The advantage of using MessengerService is that, since it runs in the address space of the app itself, objects can be referenced directly. Intents require objects to be serializable to allow them to travel via the kernel. Since the handler is executed from inside the apps address space, the overhead of including the kernel is

redundant and serializing and deserializing simply creates extra processing and implementation efforts.

Listing 7.3: Sending of messages

```
1  /**
2   * Sends message to all callback functions that registered interest for this
        message type
3   * @param msg: message to be transmitted
4   */
5  public synchronized <T extends IMessage> void send(T msg) {
6      for(Object receiver : mHandlers.keySet()) {
7          ConcurrentHashMap<Class<?>, ArrayList<MessageHandler>>
               recepientHandlers = mHandlers.get(receiver);
8          // If this receiver does not have any handlers - go to next receiver
9          if(recepientHandlers == null)
10             continue;
11
12         for(Class<?> key : recepientHandlers.keySet())
13             if(key.isInstance(msg))
14                 for(MessageHandler handler :
                       recepientHandlers.get(key))
15                     handler.handle(msg);
16     }
17 }
```

7.2 NFC communication library

NfcComHandler is the core class in the Nfc.Communication module. It has two static classes: NdefComHandler and ByteComHandler. These handle transfers of Ndef messages and raw bytes respectively. To the end user, this creates a simplified view of tags as a simple storage media.

Two of the base stubs in NdefComHandler are displayed in Listing 7.4. These show that the tag is passed with a UUID object to the function. These are non-blocking functions that start a separate thread to perform the communication asynchronously. When the communication is finished, an instance of a ComHandlerMessage[1] is sent to the MessengerService framework together with the supplied UUID object. Therefore, it is required that anyone wanting to know the result of the transfer, needs to register a handler for the ComHandlerMessage class in MessengerService. The UUID object can be used to uniquely identifying the results from a transfer.

Listing 7.4: Sending and receiving NDEF messages

```
1  /**
```

[1]Belongs to the *no.as.gold.nfc.communication.messages* package

```
2    * Reads the data available on a tag and marks it for sending to a
         designated receiver. This task is done asynchronously and will result
         in a ComHandlerMessage being published to MessengerService.
3    * @param tag Tag to read from
4    * @param initiatorUUID Universally unique sender ID to associate with
         this read operation
5    */
6   public static void Read(Tag tag, UUID initiatorUUID);
7
8   /**
9    * Writes bytes to a tag. This task is done asynchronously and will result in
         a ComHandlerMessage being published to MessengerService.
10   * @param tag Bytes to write
11   * @param initiatorUUID Universally unique identifier of object initiating
         this write operation
12   * @param data Bytes to write
13   */
14  public static void Write(Tag tag, UUID initiatorUUID, byte[] data);
```

The transfers of binary data greatly resemble that of NDEF messages with the difference that it also requires the offset into the tags memory where the data should be inserted or extracted. The stubs for these functions are given in Listing 7.5.

Listing 7.5: Sending and receiving binary data

```
1   /**
2    * Writes bytes to a tag. This task is done asynchronously and will result in
         a ComHandlerMessage being published to MessengerService.
3    * @param tag Tag to write to
4    * @param selectedTech Technology to use when transferring
5    * @param writeUUID Unique ID to associate this write operation with
6    * @param payload Bytes to write
7    * @param offset Offset into the memory of the tag where data is to be
         inserted.
8    */
9   public static void Write(Tag tag, String selectedTech, UUID writeUUID,
        byte[] payload, int offset);
10
11  /**
12   * Writes data to a tag. This task is done asynchronously and will result in
         a ComHandlerMessage being broadcasted to MessengerService.
13   * @param tag Tag to read from
14   * @param selectedTech Technology to use when transferring
15   * @param mReadUUID Unique ID to associate this read operation with
16   * @param start Byte position to start reading from
17   * @param length Number of bytes to read
18   */
19  public static void Read(Tag tag, String selectedTech, UUID mReadUUID,
```

```
    int start, int length);
20
21  /**
22   * This function transfers a command consisting of raw bytes directly to
         the tag. No SOD or EOD must be added because it will be added
         automatically.
23   * When the transfer is completed, this function will post the results as a
         {@link
         no.as.gold.nfc.communication.messages.ComHandlerMessage} to
         {@link no.as.gold.simplemessenger.MessengerService}.
24   * @param tag Tag to send the command to
25   * @param uuid UUID to mark the message with
26   * @param command raw byte command to be transferred to the tag. (do
         not include SOD or EOD)
27   */
28  public static void TransferCommand(Tag tag, UUID uuid, byte[]
      command);
```

In addition to reading and writing, ByteComHandler provides a TransferCommand function. The objective of this function is to transfer a command consisting of raw bytes to the given tag. This is done by simply transferring the raw bytes to the tag and posting the response in a ComHandlerMessage to the MessengerService framework together with the supplied UUID object. This allows the sender to identify the response by registering a callback function to handle ComHandlerMessages and filtering for messages with the given UUID object.

In some situations, (such as when, for example, the response time needs to be minimized) it is desirable to perform blocking read and write operations. This is implemented for binary communication with the tags through a set of wrapper classes that wrap a given technology to present a single interface for performing IO operations with the tag. The WrapperFactory[2] is a helper class that helps with selecting the correct wrapper for tags. The method stubs for this class is given in Listing 7.6. As can be seen, it only requires the tag. The factory identifies the technologies supported by the tag and checks if any of these are supported by the Nfc.Communication library, if so it returns the wrapper that handles this technology. If multiple technologies are supported, it selects the wrapper that handles the topmost abstraction level. For example, NFC Forum Type 2 Tag supports both NfcA technology and MifareClassic technology. The MifareClassic technology is a higher level protocol for communicating with the tag and the selected wrapper is therefore the one that wraps around the MifareClassic technology.

Listing 7.6: Wrapper factory

```
1  /**
2   * Creates a wrapper from the given tag.
3   * @param tag Tag to create wrapper from
```

[2] Belongs to the *no.as.gold.nfc.communication.technologies* package

```
4   * @return Wrapper for communication with tag
5   */
6  public static ITagTechnologyWrapper getWrapper(Tag tag) throws
       UnsupportedOperationException;
```

7.3 NFC benchmark app

The Nfc.Benchmark module contains four projects: Nfc.Benchmark.Domain, Nfc.Benchmark.Persistence, Nfc.Benchmark.View and Nfc.Benchmark.View.Gui that handles domain objects, persistence of data, Fragments and Activities respectively.

7.3.1 Domain

The Nfc.Benchmark.Domain module contains three interfaces that define the three sample types that are collected when benchmarking the tags: IRttSample, IComBenchmarkSample and IBatteryBenchmarSample. Additionally, it contains three classes that implement these interfaces and a convenience class for handling collections of IComBenchmarkSample samples.

7.3.2 Persistence

Persistence of domain objects is done through an SQLite database, and particularly with the help of the SQLiteOpenHelper [3]. The DatabaseHandler class contained in this module, extends the abstract SQLiteOpenHelper class, which requires two functions to be implemented. Parts of this class with stubs of its core functions are given in Listing 7.7. This listing has left out the creation of the seven tables that store samples from the benchmarks. For each of these tables, there exist an add method that adds a sample to the designated database table.

The listing also contains two stubs that are required when extending the abstract SQLiteOpenHelper class, the *onCreate()* and *onUpgrade()* methods. The *onCreate()* method is the method that creates the tables for the database and is only executed when the database that is being referenced either has an old version number or does not exist. The *onUpgrade()* method makes it possible to perform operations on the database prior to running the *onCreate()* method. This is an ideal place to make table updates if the tables have changed and we want to keep the data stored in the old tables. As a side-note, this function was used to introduce a new table for the RTT measuring, because this metric was introduced after the first measurements in the experiment (see Section 8.1) were recorded.

Listing 7.7: The DatabaseHandler class

[3] Abstract helper class to manage SQLite databases, resides in android.database.sqlite.SQLiteOpenHelper package

```
1  /**
2   * This class maps objects to a database. The class only handles putting
        objects to the database.
3   * Created by Aage Dahl on 18.03.14.
4   */
5  public class DatabaseHandler extends SQLiteOpenHelper {
6
7      public DatabaseHandler(Context context) {
8          super(context, Environment.getExternalStorageDirectory() +
               "/Nfc.Benchmark/" + DATABASE_NAME, null,
               DATABASE_VERSION);
9      }
10
11     //region SQLiteOpenHelper Overrides
12     // Creating Tables
13     @Override
14     public void onCreate(SQLiteDatabase db){...};
15
16     // Upgrading database
17     @Override
18     public void onUpgrade(SQLiteDatabase db, int oldVersion, int
           newVersion){...};
19     //endregion SQLiteOpenHelper Overrides
20
21     //region Public methods
22
23     /**
24      * This function removes all tables in the database
25      */
26     public void clearDatabase(){...};
27
28     /**
29      * Adds a sample to the binary read benchmark Table
30      * @param sample Sample to be added
31      */
32     public void addReadCommunicationSample(IComBenchmarkSample
           sample) throws Exception{...};
33
34     /**
35      * Adds a sample to the binary read benchmark Table
36      * @param sample Sample to be added
37      */
38     public void addWriteCommunicationSample(IComBenchmarkSample
           sample) throws Exception{...};
39
40     /**
41      * Adds a sample to the binary read all benchmark Table
42      * @param sample Sample to be added
```

```
43          */
44          public void
                addReadAllCommunicationSample(IComBenchmarkSample
                sample) throws Exception{...};
45
46          /**
47           * Adds a sample to the rtt Table
48           * @param sample Sample to be added
49           */
50          public void addRTTSample(IRttSample sample) throws
                Exception{...};
51
52          /**
53           * Adds a sample to the battery read benchmark Table
54           * @param sample Sample to be added
55           */
56          public void addReadBatterySample(IBatteryBenchmarkSample
                sample) throws Exception{...};
57
58          /**
59           * Adds a sample to the battery write benchmark Table
60           * @param sample Sample to be added
61           */
62          public void addWriteBatterySample(IBatteryBenchmarkSample
                sample) throws Exception{...};
63
64          /**
65           * Adds a sample to the battery power benchmark Table
66           * @param sample Sample to be added
67           */
68          public void addPowerBatterySample(IBatteryBenchmarkSample
                sample) throws Exception {...};
69          //endregion Public methods
70
71          //region private methods
72          ...
73      }
```

7.3.3 Fragments

The Nfc.Benchmark.View module is composed of three main components: Fragment layouts, models and controllers. The Fragment layouts define how UI elements are displayed to the user, the controllers map UI components to functions and the models contain the data created through user events. Some Fragments have little functionality and therefore have no models, which is due to the fact that to create a full model for handling a simple list of objects is seen as too much overhead and adds unnecessary

complexity to the solution.

Most of the Fragments handle the lifetime of a single benchmark operation. This is simplified by delegating the benchmarking procedure to an engine that implements the interface presented in Listing 7.8.

The interface defines *startBenchmarking()* and *stopBenchmarking()*, which control the two states the benchmarks can be in: running and stopped. When in the running state, the benchmark is performing the desired operation until completed and then progresses to the stopped state. Calling stopBenchmarking() when the state of the benchmark is running changes its state to stopped and stops the benchmarking. Additionally, the interface defines a *persist()* function. This stores all the data collected during the benchmarking to the supplied database.

The BenchmarkSchedulerFragment differs from the rest, because it is used to start a collection of benchmarks. It uses the interface presented in Listing 7.8 to run a set of selected benchmark engines in sequence.

Listing 7.8: Benchmark engine interface

```
1  /**
2   * Created by Aage Dahl on 11.03.14.
3   */
4  public interface IBenchmarkEngine {
5      /**
6       * Gets the name of this engine.
7       * @return Name of this engine.
8       */
9      String getName();
10
11     /**
12      * Starts the benchmarking
13      * @throws java.lang.IllegalArgumentException on error with settings
14      * @param settings Settings that define the execution of the
                benchmarking
15      */
16     void startBenchmarking(IBenchmarkSettingsModel settings) throws
            IllegalArgumentException, NullPointerException, IOException;
17
18     /**
19      * Interrupts the benchmarking
20      */
21     void stopBenchmarking();
22
23     /**
24      * Persists the samples contained in this engine to the given database.
25      * @param dbh database handler
26      */
27     void persist(DatabaseHandler dbh) throws Exception;
28 }
```

7.3.4 Activity

The MainActivity class resides inside the Nfc.Benchmark.View.Gui module. This class handles and displays all Fragments available in the Nfc.Benchmark.View module as separate tab views. Additionally, it displays a log output window at the bottom of the screen. Whenever a Fragment experiences a runtime error, it transmits an instance of the *ErrorMessage*[4] class with the error message description, to MessengerService. The log output window is used to display these messages. This is done by the MainActivity instance registering a handler in MessengerService for handling ErrorMessage messages. The handler extracts the error description from the message and displays it in the output window.

The NFC benchmark app has an intent filter in its manifest file to receive Intents when a new NFC technology is detected. The issue with this is that Android tries to start a new Activity every time this happens. If the app is already running, it is important that the currently active Activity receives the Intent, and it is not desired that a new Activity is started. Therefore, the MainActivity class calls on the functions as shown in Listing 7.9. When the *setupForegroundDispatched()* function is called, it redirects any Intents from NFC tags to the currently running Activity. This enables the running instance of the NFC benchmark app to have full control of the NFC communication as it is in a running state. When exiting the running state, it releases the control of the NFC communication by calling the *stopForegroundDispatch()* function. This enables other apps to receive Intents when tags are discovered.

Listing 7.9: Setup and stopping of foreground dispatch

```
1  @Override
2  protected void onResume() {
3      super.onResume();
4      setupForegroundDispatch(this, mAdapter);
5  }
6
7  @Override
8  protected void onPause() {
9      super.onPause();
10     stopForegroundDispatch(this, mAdapter);
11 }
```

When an Intent arrives to the MainActivity, and it is identified that this is an Intent containing a new tag (*e.g.* a new NFC tag has been detected), the process wraps this tag inside a NewIntentMessage and broadcasts it to the MessengerService framework. This allows other objects to receive information about the discovery of new tags.

[4]Belongs to the *no.as.gold.simplemessenger.messages* package.

7.4 NFC protocol tester app

The Nfc module has an identical structure to the Nfc.Benchmark module, it also contains four projects: Nfc.Domain, Nfc.Persistence, Nfc.View and Nfc.View.Gui.

7.4.1 Domain

The Nfc.Domain module contains domain objects used by the NFC protocol tester . Currently there are no domain objects, and this project is only added to stay true to the architectural design.

7.4.2 Persistence

The Nfc.Persistence module is prepared for persistence of data. The current version does not persist any data, and therefore this module does not supply any functionality.

7.4.3 Fragments

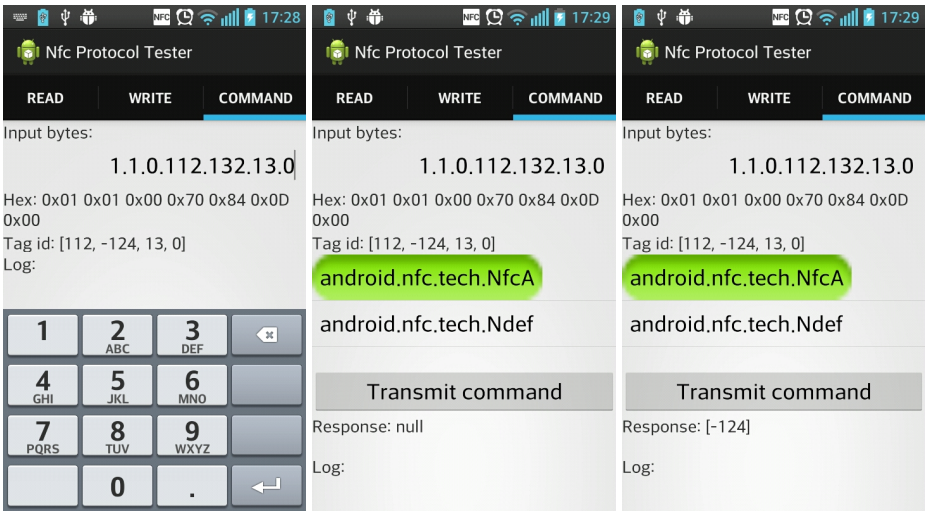
The Nfc.View module is composed of three views, which each has a Fragment layout, a model and a controller. The Fragment layouts define how UI elements are displayed to the user, the controllers map UI components to functions and the models contain the data created through user events. The three views allow the user to perform reading of a tags memory, writing to a tags memory and transmitting user-defined commands to a tag. The response from the tag is presented to the user.

7.4.4 Activities

The MainActivity class resides in the Nfc.View.Gui module. This renders the Fragments that enable UI components to be presented to the user. It also handles user navigation between the three available views.

7.4.5 Evaluation

This subsection is supposed to introduce the results from testing Zaher's tag. However, the tag is still in the process of being manufactured. Due to this, I have decided to pursue testing of Tag1 to show that the software is able to send commands in byte format to the tag and observe the response. Three screen-shots of the app are presented in Figure 7.1. The first screen-shot shows how the byte array is inputted. The input 1.1.0.112.132.13.0 is deciphered to the following hexadecimal values: $0x01$ $0x01$ $0x00$ $0x00$ $0x70$ $0x84$ $0x0D$. The input is deciphered by extracting the values between the dots and interpreting them as byte values. The given input is a command for Tag1 to read the byte value at address 0 in the tags memory. This command is constructed in accordance with the protocol description in [17].



(a) Inputting command (b) Selecting technology (c) Command transferred

Figure 7.1: NFC protocol tester illustrations

When the tag is detected, the app displays the tags id and the tags available technologies. In the second screen-shot, the *android.tech.NfcA* technology is selected. This results in the underlying NFC communication library selecting this technology when transmitting the command to the tag.

In the third screen-shot, the command has been transferred to the tag. The tag responds with the byte located at memory address 0, and this value is displayed just below the "Transmit command" button. This value is -124, which is not surprising as the first four bytes in the Tag1 memory contain the tag ID. As can be seen from the screen-shot, this tags ID is 112, -124, 13, 0, and the byte value at memory address 1 should therefore be -124.

Chapter 8

The experiment

This chapter starts by describing the design of the experiment. It follows with the description of the implementation of the experiment and data collection. The chapter that follows introduces and describes the results of the experiment.

8.1 The experiment

In this section, the design of an experiment is introduced. The goal of this experiment is to establish which tag standard is most suitable for Zaher's tag, and how the communication protocol should be designed. Zaher's tag is planned to reside inside human tissue, therefore it is important to know how the tag standards are affected by this environment.

The communication protocol of Zaher's tag is not defined, and one of the design decisions to make is how to bundle information transferred from the tag to the reader. Information traveling between the tag and the reader can be bundled in order to improve communication efficiency, and it is therefore important to know the effect of doing this, both on battery and on throughput. To identify how this tag is going to be used, it is important to know how it performs at different distances between the reader and the tag, as this influences how the user interacts with it.

The experiment shows how the NFC benchmark app is used to extract information about the performance of five tags based on five different standards. The tag standards are described in Chapter 3. The results of the analysis are used to deduce which of these standards are better suited for Zaher's tag.

The objective of the experiment is therefore to identify communication characteristics and battery performance of tag-standards and show how these are influenced by:

- displacement between the tag and the reader, and
- size of data transferred, and
- placing the tag in a saline solution.

Description	Quantity
LG P700 mobile phone	1
Rail	1
Rail-clamp	1
Clamp	1
NFC forum Tag1 (Topaz 512)	1
NFC forum Tag2 (NTAG203)	1
NFC forum Tag3 (FeliCa Lite-S)	1
NFC forum Tag4 (Mifare Desfire)	1
MifaceClassic tag (MF1S50)	1
Saline solution (0.9%NaCl)	150ml
Plastic card	5

Table 8.1: Materials for experiment

The experiment is designed to extract a lot of data from multiple areas of interest.

The experiment must be repeatable so that any significant findings can be verified through further experiments. It must be possible to adjust the displacement between the reader and the tag in a controlled and accurate way so that it produces representative measures for all tags. It is also important that the number of external factors is limited and that variables are changed in a controlled manner. This ensures that it is a controlled experiment. Too many uncontrolled external factors can pollute the results and render them unusable. The experiment also has to be realistic for Zaher, so that he is able to base his decisions on the results.

To achieve repeatability, the experiment design is explained in full through the remainder of this chapter. This includes materials used and construction of the testbed. This also introduces how the other requirements are fulfilled.

The results are analyzed and presented in Chapter 9.

8.1.1 Design

The materials used in this experiment are summarized in Table 8.1.

Figure 8.1 shows the planned set-up of the experiment. The rails allow the displacement between the phone and the tag to be changed, but still remain accurate. The clamp is attached to the phone and assures that the phone resides in the same position during the experiment and thereby reduces the impact of external variables. Each tag is attached to a plastic card, and the plastic card is fastened to the rail-clamp.

For each tag and displacement, the seven benchmarks summarized in Table 6.1 are executed, and the collected data is stored in a database called *BenchmarkResults.db*. The benchmarks collect data related to tag types, size of the data being transmitted, timestamp for the measurement, whether communication is achieved or not and displacement between the tag and the reader.

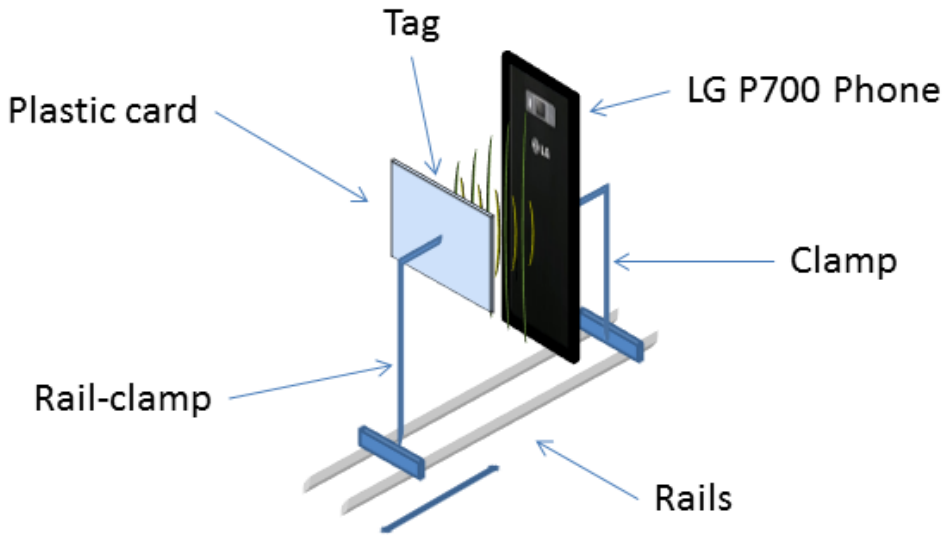


Figure 8.1: Illustration of testbed

In the first part of the experiment, the tag and the reader is separated by air while data is collected. In the second part of the experiment, the data collection is repeated with the tags covered with approximately seven millimeter saline solution on all sides. This is done to replicate the permittivity the tag experiences while covered with human tissue[21, 46]. To cover the tags in a saline solution, the tags are inserted into small plastic bags that in turn are inserted into larger bags containing a saline solution. The bags prevent the saline solution from damaging the tags. The thickness of the saline solution is controlled by hand prior to all experiments.

The saline solution variable was not planned in the original experiment, but is added towards the end of this work. Due to limited time, an optimal solution is not available and any results must therefore be treated with causion. At best, the results from the saline solution part of the experiment can be used as an indication towards further investigation. More realistic constructions exist, such as discussed here [45].

8.1.2 Phone settings

It is important to prevent the external interferences from influencing the results of the sampling. It is possible to acquire this by deriving a new version of the Android operating system, where all but the most basic operations are removed and where full priority is given to the NFC benchmark app. This gives very accurate results regarding the power consumption and communication efficiency. However, the effort required to do this exceeds what is feasible in this thesis. Alternatively, running apps can be turned off, and the phone can be set in "Flight mode", which turns off all wireless technology. This mode requires that NFC is manually enabled. I choose to use the second method because the first requires more time than I have available.

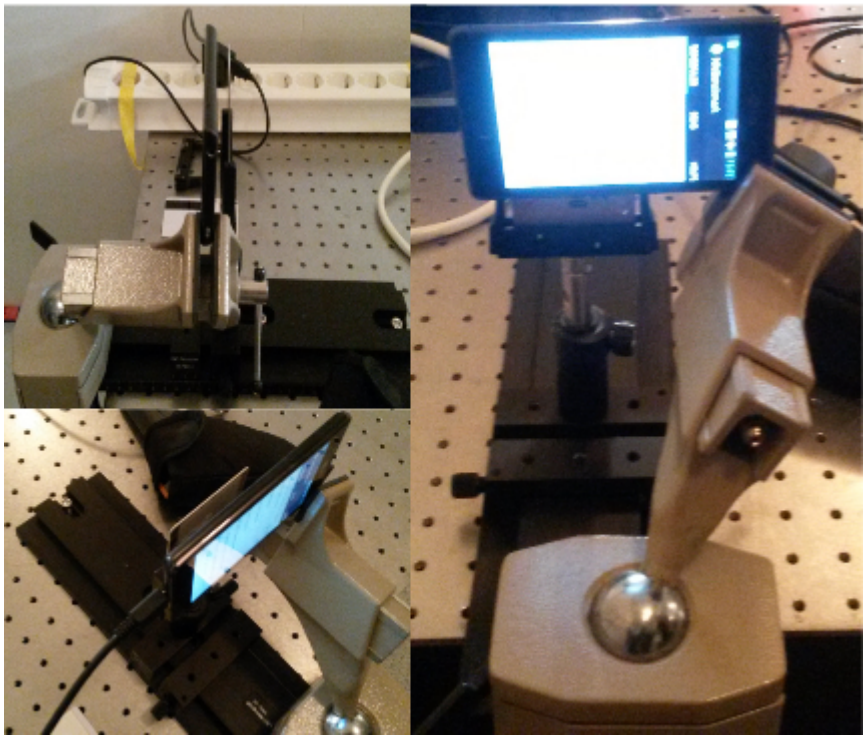


Figure 8.2: Test-bench construction

8.2 Experiment implementation

This section explains how the experiment is implemented.

8.2.1 Composition

The test-bench is constructed based on the ideas in the preceding chapter with some minor modifications. The phone is placed horizontally as this fitted with the clamp that was acquired. The construction of the testbed is shown in Figure 8.2. The construction is kept for the duration of the experiment without being taken apart. The rail and the clamp are fixed to the table, the phone is fixed to the clamp, and the rail clamp is attached to the rail. Each of the five tags is glued to the upper part of a plastic card, and the plastic cards are in turn fastened to the rail clamp.

8.2.2 Experience

The preliminary battery tests show signs of great variation between each conducted benchmark, and, after repeated trials with the same tag and identical distance from the tag, a pattern starts to emerge. The result is greatly dependent on the battery level at the beginning of the benchmarking. Hence, when the tests are conducted with a battery level of 40%, the drop of 3% takes about 450 seconds. However, when the same experiment is conducted with a battery level of 100%, the same drop takes about 350 seconds. To avoid this variance, the battery is fully recharged

before performing battery tests for reading from, writing to and powering of the tag.

8.2.3 Unexpected behavior

The benchmarks without the saline solution are conducted as planned, and in this setting, the tag is placed at the topmost part of the plastic card. For the benchmarks with the saline solution, to fully submerge the tag and at the same time prevent the saline solution from pouring into the plastic bag, the tag is glued to the lower part of the plastic card. This has the effect that the phone needs to be tilted about 30° to detect the tag. I did consider the idea of redoing the experiments. However, it takes a few weeks to collect the samples, and the objective limitations of the study, prevent me from repeating the whole experiment once again.

8.3 Data collection

This section summarizes how data is collected and extracted.

8.3.1 Persistence

All results are stored in a SQLite database and extracted to a laptop for further analysis. Whilst on the laptop, the data is exported to a MySQL database for further processing. MySQL has an extended set of functions available for analysis of data compared to SQLite. This is extensively used to extract statistical information on the collected data.

8.3.2 Extraction

The *Name* of each sample follows the syntax *n*mm, where *n* identifies the displacement in millimeters between the tag and the reader. All samples collected with the saline solution have "_*s*" appended to the end of the *Name*. In this way, it is possible to identify which samples belonged to which tests. The *Technology* identifies the tag technology that the sample is based on.

For each average calculated, the standard deviation is also calculated to indicate how much the results vary from the estimated average.

Three queries are used to extract the data from the SQLite database. The query presented in Listing 8.1 extracts the average round trip time measured for each technology and displacement. RTT gives the expected latency of the tags and defines the expected delay for a response. The acquired value tells something about the minimum expected delay from information is sent until the response is received. This value indicates what the expected response time is and how long the reader needs to power the tag to receive a response.

The results consist of averaging and measuring the standard deviation on the 50 samples collected for each setting, and are grouped by technology

and name. The average throughput is given by:

$$RTT_{average} = \frac{\sum_{i=1}^{n} rtt_i}{n}, \; where \left| \begin{array}{lcl} n & = & sample\,count \\ rtt & = & round\,trip\,time \end{array} \right. \tag{8.1}$$

The standard deviation is given by:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}, \; where\, \mu = \frac{1}{N}\sum_{i=1}^{N} x \tag{8.2}$$

Listing 8.1: RTT SQL query

```
-- Calculate average RTT for each technology and name
SELECT
        technology as Technology,
        name as Name,
        avg(RTT) as Average,
        STDDEV_POP(RTT) as StandarDeviation
FROM
        benchmarkresults.rtt
GROUP BY
        technology, name
```

The query presented in Listing 8.2 extracts the average duration and communication throughput grouped by the name, technology and size of the sample. This is used to extract communication throughput for reading and writing data packets of different sizes to tags at different displacements between the tag and the reader. The different throughputs are important to measure, because they describe how the tags perform in relation to sizes of packets transmitted. As stated in the introduction of this experiment, the communication protocol is not yet defined, and the packet sized performance can influence how data should be bundled to acquire optimal performance. In addition to the throughput, the standard deviation is measured to indicate how the measured values vary from the given average.

In this query, < *table* > is one of the three tables that relate to communication performance: *ReadAllCommunicationPerformance*, *ReadCommunicationPerformance* and *WriteCommunicationPerformance*. The standard deviation is calculated based on Equation 8.2 and the following formula presents the averaging performed by this query:

$$T_x = \frac{x}{\frac{\sum_{i=1}^{n} t_i}{n}}, \; where \left| \begin{array}{l} T = Avg.\,throughput \\ x = size \\ n = sample\,count \\ t = elapsed\,time \end{array} \right. \tag{8.3}$$

Listing 8.2: Communication performance SQL query

```
-- Calculates average throughput for each size, technology and name
```

```
SELECT
        name as Name,
        technology as Technology,
        size as SizeInBytes,
        avg(duration) AS AverageDuration,
        avg(size/duration) AS AverageSpeed,
        STDDEV_POP(size/duration) as StandardDeviation
FROM
        benchmarkresults.<table>
WHERE
        status = "Succeed"
GROUP BY
        name, technology, size
```

The last query is presented in Listing 8.3 and extracts the average energy consumption for each second elapsed and for each byte communicated for the reading, writing and powering benchmarks related to estimating energy consumption. These are key values to measure as they describe how the energy efficiencies of the different standard. High energy efficiency is desired, because it improves usability of the system. The measurement of energy consumed per second is used to describe how reading, writing and powering operations affect the battery can help design a protocol that is more energy friendly.

$< table >$ relates in this context to one of the three table types: *ReadBatteryPerformance*, *WriteBatteryPerformance* and *PowerBatteryPerformance*. The following formula presents the calculation of energy per second:

$$E_{second} = \frac{e_{start} - e_{end}}{t}, where \left| \begin{array}{lcl} e_{start} & = & Start\,energy\,level \\ e_{end} & = & End\,energy\,level \\ t & = & Elapsed\,time \end{array} \right. \tag{8.4}$$

The following formula calculates the energy consumption per second:

$$E_{byte} = \frac{e_{start} - e_{end}}{b}, where \left| \begin{array}{lcl} e_{start} & = & Start\,energy\,level \\ e_{end} & = & End\,energy\,level \\ b & = & Bytes\,transferred \end{array} \right. \tag{8.5}$$

Listing 8.3: Battery performance SQL query

```
-- Calculate average energy drop per byte and per second for each
    technology and name
SELECT
        name AS Name,
        technology AS Technology,
        (startLevel – endLevel) AS EnergyDrop,
        elapsedTime AS ElapsedTime,
        (startLevel – endLevel)/ElapsedTime AS EnergyDropPerSecond,
        (startLevel – endLevel)/bytesTransferred AS EnergyDropPerByte,
```

```
FROM
        BenchmarkResults.<table>
WHERE
        startlevel <> endLevel and elapsedTime > 0
GROUP BY
        name, technology, EnergyDrop
```

Chapter 9

Data presentation

In total, more than one million table entries are collected and analyzed, and approximately three GB of data is transmitted between the NFC reader and the five tag types. The result of this data collection is presented in this chapter and visualized in a set of figures that aim at comparing the performance of each tag type. One section is dedicated to each benchmark, where the data collected for that benchmark is described and discussed.

As a general note, all communication performance is based on the actual transmission of data, which does not include time for connecting to and disconnecting from the tag. The measurements of throughput are based on the amount of successfully transferred bytes per second. Unsuccessful byte transfers are those that contain corrupted data, which result in reduced throughputs.

In the presented results, the displacement varies from 0mm to 30mm. 30mm is defined from the approximately average detection range of the five tags. 0mm is defined as the situation when the tag is touching the phone. Not all tags have values for all displacements, which is due to the fact that not all benchmarks were successful. The unsuccessful benchmarks did not communicate with the tag throughout the run and are therefore left out from the results.

The aim is to extracting one sample with the saline solution for each tag, however, more were collected, and I decided to use these values as well to have a stronger basis for analysis.

All values given in tables and referenced in the text are correct to 3s.f.

9.1 Round trip time

Round trip time (RTT) is the time taken for a signal to be transmitted from the sender to the recipient and back again. This value is estimated in computer networks by transmitting minimal packets from the sender to the receiver and back again.

The RTT values are based on the average response time for transmitting the commands listed in Table 9.1 and is calculated using Formula 8.1. None of the tags have a specific command for estimating the RTT. Therefore, the given set of commands are used to base this value on. The number of bytes

transferred are defined by the respective commands and therefore vary for each tag type.

The timer for estimating the RTT is started at the point when the command is sent with a *transcieve*[48] function. This function allows apps to transmit a command to a library, which then transmits it to the tag. When the operation is completed and the function returns with the result, the timer is stopped and the RTT is calculated.

Figure 9.1 presents the RTT values. The delay is displayed along the vertical axis, and the tag types and displacements are shown along the horizontal axis. The samples with and without a saline solution are shown with red and blue columns respectively. The values are calculated by averaging 50 samples collected for each displacement. One standard deviation from the calculated average is indicated by black error bars on top of each column.

At 0mm displacement, Tag1 has a RTT of 8.52ms (3s.f.). This varies greatly from MifareClassic, which has a RTT of 31.0ms (3s.f.). When comparing these values and considering the standard deviation, it is apparent that MifareClassic shows a significantly higher RTT and that this is statistically significant. The results indicate a high overhead for transferring data with MifareClassic tags.

Tag3 shows results for the saline solution that indicates that the saline solution improves the RTT. However, the standard deviation shows that this is not a statistically significant difference.

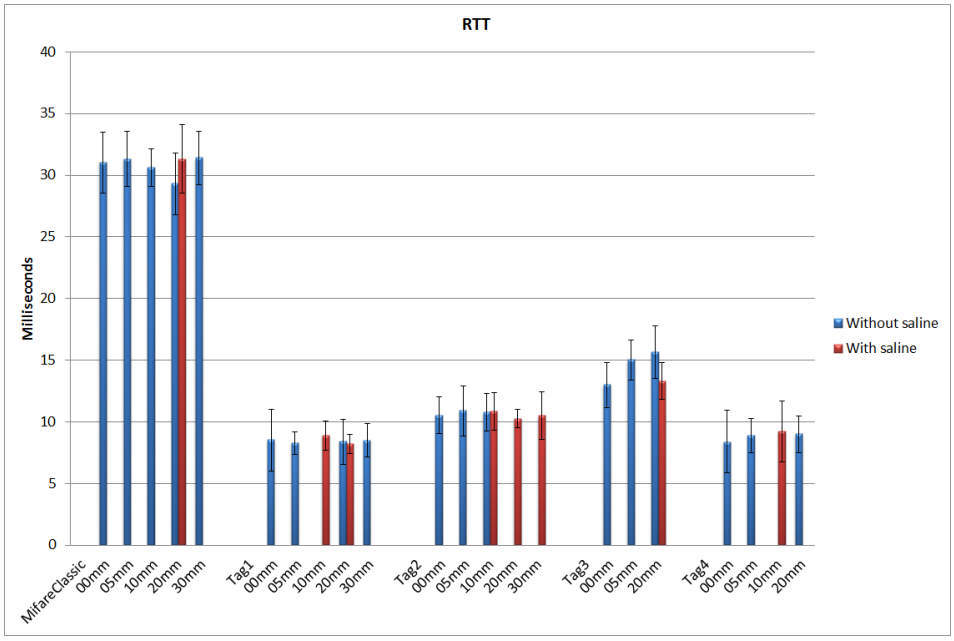


Figure 9.1: Round trip time

Tag	Command	Bytes sent	Bytes received
MifareClassic	Authenticate sector	13	3
Tag1	Read	9	3
Tag2	Read	4	18
Tag3	Read	8	18
Tag4	$0x00$[1]	3	4

Table 9.1: Commands used to estimate RTT

9.2 Read communication performance

The read communication performance is based on the reading of $1 \rightarrow n$ bytes of data from the tag, where n is the full size of the tags memory. The step size between each sample is one byte. Reading is repeated 50 times for each displacement, tag type and size, and then the average of these 50 samples represent the read throughput acquired for the given tag, size and displacement. The graphs in Figure 9.2 and 9.3 show these read throughputs. The data in the figures is organized with the throughput given along the vertical axis, the size given along the horizontal axis and the tag type and displacement along the depth axis. The throughput is calculated with the equation given in Formula 8.3. A slice of throughput values with standard deviation for 64 bytes packages are provided in Table 9.2 for the results without a saline solution and in Table 9.3 with a saline solution.

The first Subsection describes the results acquired without a saline solution, whereas the second describes the results with the saline solution.

Reading of data is done with the *read* command of the respective tags. For tags that have multiple commands for reading, the one that best fits the remaining bytes to read, is used. Hence, if a given tag has commands to read 4 and 16 bytes, the command for reading 16 bytes is used until four or less bytes remain.

9.2.1 Without saline solution

The graphs presented in Figure 9.2 shows the read throughput when the tag is not covered with a saline solution. The throughput varies greatly with the tag type and packet size, and it is clear that some tags are better suited for certain packet sizes. The graphs also show that there is no noticeable difference in throughput when the displacement between the tag and the reader is increased. The tag with the most noticeable reduction in throughput with displacement is Tag3, which has a throughput of 3080 (3.s.f) BPS at 64 byte packets and 0mm displacement, compared to 2650 (3s.f.) BPS at 20mm displacement. With a standard deviation of 120 and 144 bytes respectively, the results do show some statistical significance. The other differences are much smaller and do not show any great correlation between throughput and displacement.

[1]Not a defined command, responds with error code $0x67$ $0x00$ (Wrong length)

Tag	0mm		5mm		10mm		20mm		30mm	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
MifareClassic	812	60.4	811	61.4	-	-	-	-	811	84
Tag1	108	3.10	111	4.06	-	-	110	3.67	110	4.56
Tag2	1520	106	1430	151	1450	144	-	-	-	-
Tag3	3090	120	2580	175	-	-	2650	144	-	-
Tag4	1320	64.2	1190	145	-	-	1220	141	-	-

Table 9.2: Throughput and S.D. in BPS for reading 64 bytes packets without saline solution

A noticeable feature of the graphs is that they have throughput values for packet sizes up to the size of the tags memory. This is because the tag cannot receive read commands that expand beyond the tags memory. It is easy to see from this that Tag1 has the smallest and Tag4 has the largest memory.

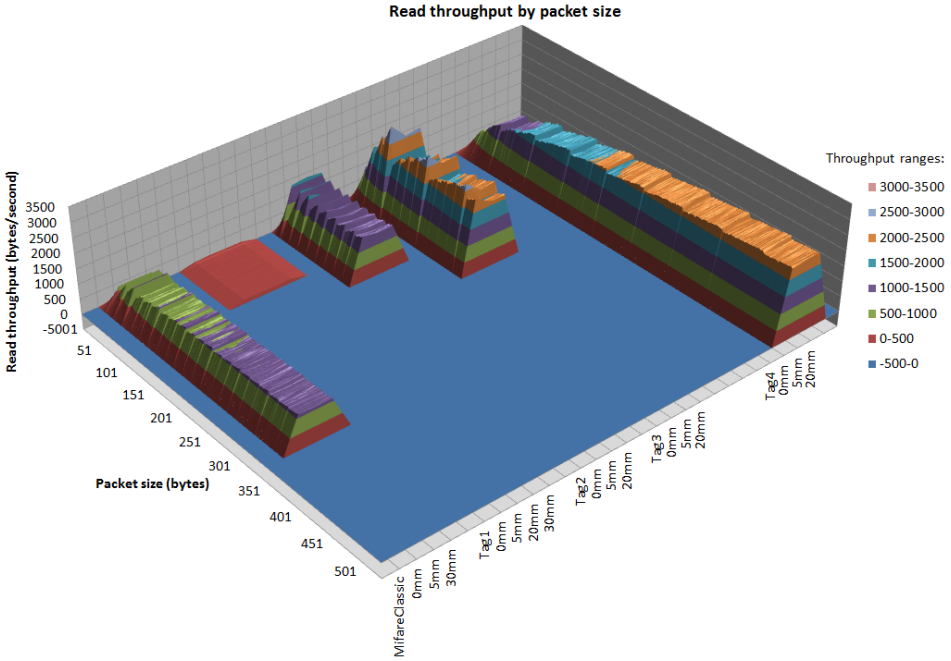


Figure 9.2: Read communication performance without saline solution

9.2.2 With saline solution

The graphs presented in Figure 9.3 show the read throughput when the tag is covered with a saline solution. As without the saline solution, the throughput varies greatly with the tag type and packet size, and it distinctly follows the same pattern indicating that the saline solution does not greatly degrade the read throughput. The saline solution physically separates the tag from the reader, so it is naturally not possible to test the read

Tag	10mm		20mm		30mm	
	μ	σ	μ	σ	μ	σ
MifareClassic	-	-	820	62.5	-	-
Tag1	110	4.24	110	4.41	-	-
Tag2	1490	165	1470	149	1460	142
Tag3	-	-	2990	144	-	-
Tag4	1220	143	-	-	-	-

Table 9.3: Throughput and S.D. in BPS for reading 64 bytes packets with saline solution

throughput at 0mm and 5mm displacements.

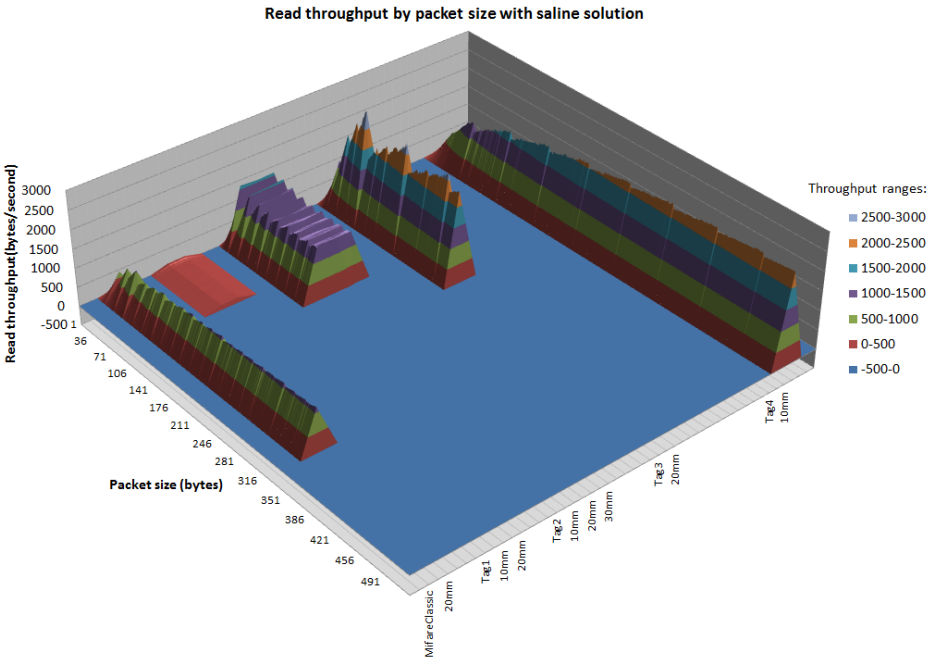


Figure 9.3: Read communication performance with saline solution

9.2.3 Summary

Some patterns are observable from the figures introduced in the two preceding subsections. There are distinct peaks at regular intervals for all except Tag1. Tag4 and MifareClassic displays a rounder pattern where the peaks are less prominent compared to Tag2 and Tag3. The displacement between the tag and the reader does also not seem to influence the read throughput within the tested range.

Tag	Size (bytes)
MifareClassic	333
Tag1	122
Tag2	143
Tag3	208
Tag4	512

Table 9.4: Memory sizes read with the ReadAll function

9.3 Read all communication performance

The graphs in Figure 9.4 show the results for reading all the available memory in the tags. Tags are aligned along the horizontal axis of the figure, and the read throughput is aligned with the vertical axis. The samples with and without a saline solution are shown with red and blue columns respectively. One standard deviation from the calculated average is indicated by black error bars on top of each column. The memory sizes read are summarized in Table 9.4, and the equation used for calculating the results is given in Formula 8.3. The results are based on the average and standard deviation calculated from 50 samples collected at each displacement between the tag and the reader. The figure compares the communication throughput acquired for all the five tags at different displacements, and with and without a saline solution.

It is clear from the figure that the saline solution and displacement has no noticeable effect on the throughput when reading the whole of the tag's memory. None of the results show any difference that greatly exceeds a standard deviation from the mean. However, when comparing the tag types, the difference becomes significant. The largest difference can be observed with Tag1 at 0mm displacement with a throughput of 3440 (3s.f.) BPS and MifareClassic with a throughput of 1090 (3s.f.) BPS at the same displacement. This is significant, since the standard deviation is 127(3s.f.) BPS and 47.6 (3s.f.) BPS on each tag respectively. It is also clear that Tag1 read throughput is much higher than the other acquired read throughputs, both with and without the saline solution and at any displacement.

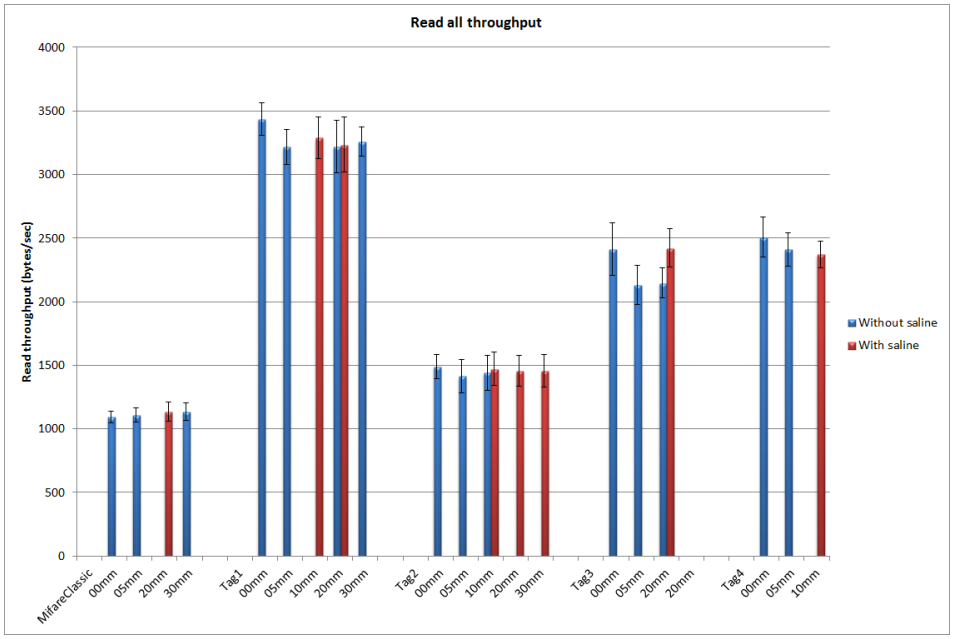


Figure 9.4: Read all communication performance

9.4 Write communication performance

The write communication performance is based on the writing of $1 \rightarrow n$ bytes of data to the tag, where n is the full size of the tags memory. The step size between each sample is one byte. Writing is repeated 50 times for each displacement, tag type and size and then the average of these 50 samples represents the write throughput acquired for the given tag, size and displacement. The graphs in Figure 9.5 and 9.6 show these write throughputs. The data in the figure is organized with the throughput given along the vertical axis, the size given along the horizontal axis and the tag type and displacement along the depth axis. The throughput is calculated with the equation given in Formula 8.3. A slice of throughput values with standard deviation for 64 bytes packages are provided in Table 9.5 for the values without a saline solution and in Table 9.6 with saline solution.

The first subsection describes the results acquired without a saline solution, and the second describes the results with the saline solution.

Writing of data is done with the *write* command of the respective tags. For tags that have multiple commands for writing, the one that best fits the remaining bytes to write, is used. Hence, if a given tag has commands to write 4 or 16 bytes, the command for writing 16 bytes is used until four or less bytes remain.

9.4.1 Without saline solution

The graphs presented in Figure 9.5 show the write throughput when the tag is not covered with a saline solution. Similar to the read throughputs, the write throughputs vary greatly with the tag type and packet size, and it is

clear that some tags are better suited for certain packet sizes. The graphs also show that within the tested range, there is no noticeable difference in throughput when the displacement between the tag and the reader is increased. The only tag that seems to have some reduced effect with displacement is again Tag3.

A noticeable feature of these results is that the acquired throughputs for writing are significantly lower than those acquired for reading. This is most striking when comparing the read throughput of Tag2 and Tag3 with the write throughputs of Tag2 and Tag3. If considering the throughput at 64 bytes and 0mm displacement for these two tags, the relationship between read and write throughput is $\frac{3090}{763} \approx 4.05$(3s.f.) times for Tag3 and $\frac{1520}{303} \approx 5.02$(3.s.f) times for Tag2. The standard deviation of the write throughput for Tag2 signifies that the mean for reading is $\frac{1520}{22.1} = 68.8\sigma$ away from the mean for writing, which can hardly be said to be a random effect. MifareClassic, Tag1 and Tag4 seem to have a much smaller difference between its writing and reading throughputs. Performing the same calculation with these reveals that the same relationship is $\frac{812}{674} \approx 1.20$(3s.f.) for MifareClassic, $\frac{108}{68.1} \approx 1.59$(3s.f.) for Tag1 and $\frac{1320}{1060} \approx 1.25$(3s.f.) for Tag4.

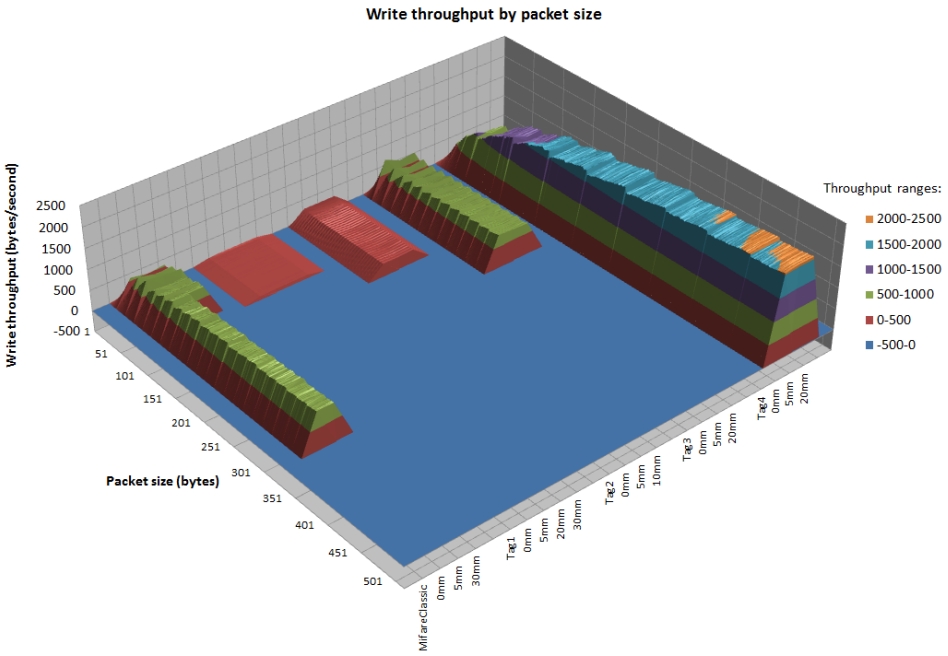


Figure 9.5: Write communication performance without saline solution

9.4.2 With saline solution

The graphs presented in Figure 9.6 show the write throughput when the tag is covered with a saline solution. As without the saline solution, the throughput varies greatly with the tag type and packet size, and it distinctly follows the same pattern indicating that the saline solution does not greatly

Tag	0mm		5mm		10mm		20mm		30mm	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
MifareClassic	674	53.6	638	38.9	-	-	-	-	643	32.5
Tag1	68.1	0.65	68.3	1.31	-	-	68.1	1.16	68.0	1.13
Tag2	303	22.1	287	34.0	301	21.2	-	-	-	-
Tag3	763	28.0	708	17.9	-	-	706	32.6	-	-
Tag4	1060	94.4	977	55.8	-	-	977	40.4	-	-

Table 9.5: Throughput and S.D. in BPS for writing 64 bytes packets without saline solution

degrade the write throughput. The saline solution physically separates the tag from the reader, which makes it impossible to test the write throughput at 0mm and 5mm displacements.

The data in Table 9.5 and Table 9.6 confirm that the throughput values do not vary greatly for package sizes of 64 bytes.

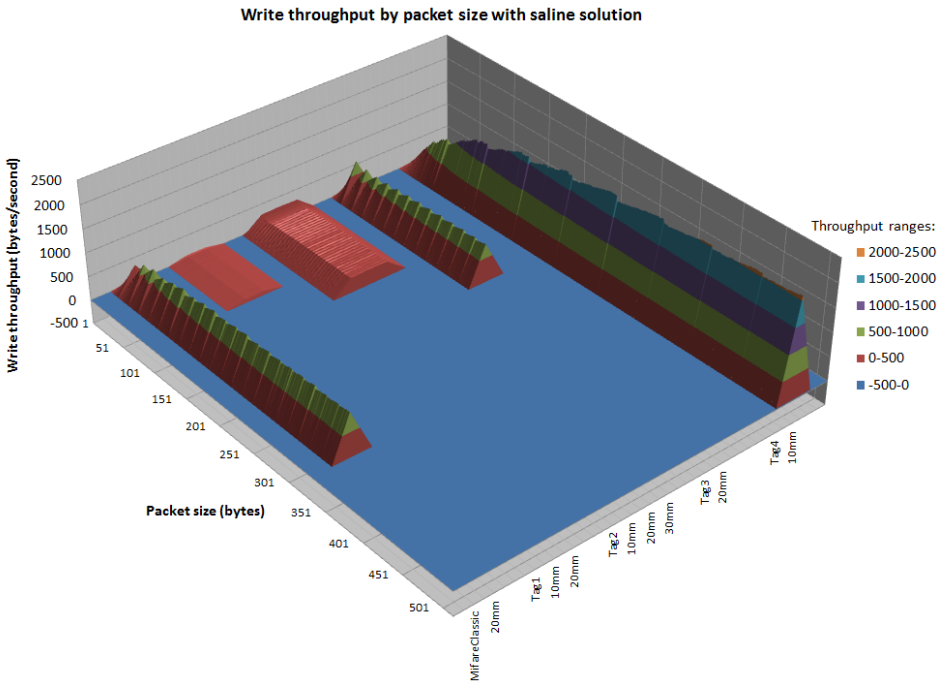


Figure 9.6: Write communication performance with saline solution

9.4.3 Summary

Similar to the results from the read throughputs, the write throughputs presented in the two preceding subsection also show some distinct peaks at regular intervals for all tags except Tag1. A difference is that Tag2 and Tag3 have much more reduced write throughputs compared to read throughputs than MifareClassic, Tag1 and Tag4. Tag4 seems to have little difference between read and write throughputs overall. Again the

Tag	10mm		20mm		30mm	
	μ	σ	μ	σ	μ	σ
MifareClassic	-	-	656	31.7	-	-
Tag1	68.3	1.06	69.1	1.15	-	-
Tag2	300	22.6	298	14.8	302	20.4
Tag3	-	-	740	36.1	-	-
Tag4	991	45.1	-	-	-	-

Table 9.6: Throughput and S.D. in BPS for writing 64 bytes packets with saline solution

displacement between the reader and the writer, and the introduction of the saline solution does not seem to influence the write throughputs significantly.

9.5 Battery performance while reading

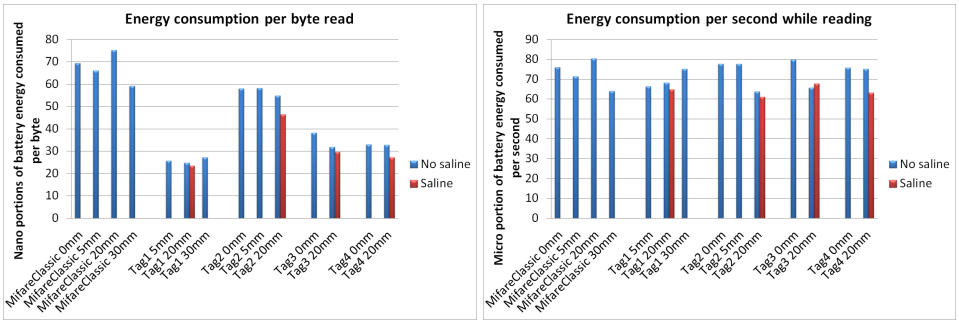
Figure 9.7 compares the energy consumption registered when continuously reading from the tags at varying displacements, both with and without a saline solution covering the tags. The benchmark reads the full memory of the tag using the *ReadAll* function, and the packet size read for each tag is summarized in Table 9.4. The results are presented in two figures, where the first shows the battery consumption per second, and the second shows the energy consumption per byte transferred. The energy consumption per second is based on Formula 8.4, and the energy consumption per second is based on Formula 8.5. In these figures, the tags are presented with displacements along the horizontal axis, and the relative energy consumption is presented along the vertical axis. The samples with and without a saline solution are shown with red and blue columns respectively.

Figure 9.7b shows that there is not much difference between the battery consumption per second regardless of what tag, displacement or whether or not the tag is covered by a saline solution. It does however show that the registered values vary with ±10%.

Figure 9.7b shows that the saline solution and displacement does not seem to influence the battery consumption much (if anything the energy consumption seems to have declined slightly). Comparing the general consumption for the tag types reveals a striking difference. Tag1, with its 25.7(3s.f.) nano portions consumed per byte, is almost three times more energy efficient per byte transferred compared to MifareClassic, with its 69.5(3s.f.) nano units consumed per byte, and outperform all the other tags. Tag3 and Tag4 seem to be close to, but not as efficient as Tag1. Tag2 is in the middle and consumes more than two times the energy per byte compared to Tag1.

Tag	Bytes
MifareClassic	333
Tag1	95
Tag2	143
Tag3	208
Tag4	512

Table 9.7: Packet sizes written for each tag



(a) Energy consumption per byte (b) Energy consumption per second

Figure 9.7: Energy consumption for reading operations

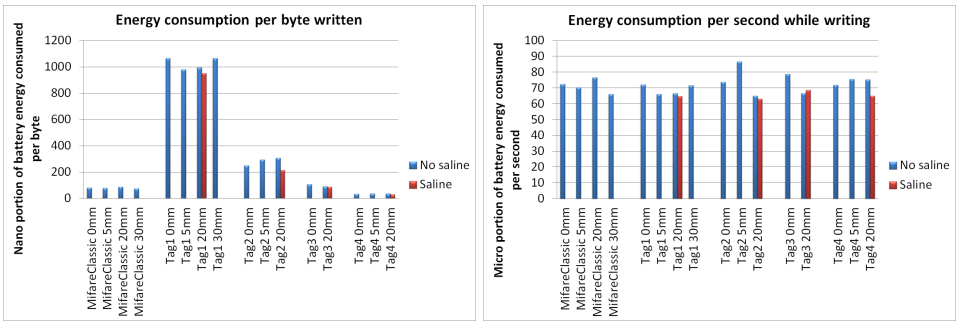
9.6 Battery performance while writing

Figure 9.8 compares the energy consumption registered when continuously writing to the tags at varying displacements, both with and without a saline solution covering the tags. The benchmark writes packets with the sizes summarized in Table 9.7, using the *write* command. The results are presented in two figures, where Figure 9.8a shows the battery consumption per second ,and Figure 9.8b shows the energy consumption per byte transferred. The energy consumption per second is based on Formula 8.4, and the energy consumption per byte is based on Formula 8.5. In these figures, the tags are presented with displacements along the horizontal axis, and the relative energy consumption along the vertical axis. The samples with and without a saline solution are shown with red and blue columns respectively.

Figure 9.7a shows that the saline solution and displacement does not seem to influence the battery consumption much. Also here there is an indication that the energy consumption declines slightly with the introduction of the saline solution. Comparing the general consumption for the tag types reveals a striking difference that also is striking when compared to reading. Tag1, with its 1070(3s.f.) nano portions consumed per byte, is vastly less energy efficient compared to the other tags, whereas Tag4, with its 35.8(3s.f.) nano portions consumed per byte, seems to be the most energy efficient.

Figure 9.7b shows, just as with the reading operation, that there is not

much difference between the battery consumption per second regardless of what tag, displacement or whether or not the tag is covered by a saline solution. Also here the registered values vary with about ±10%.



(a) Energy consumption per byte (b) Energy consumption per second

Figure 9.8: Energy consumption for writing operations

9.7 Battery performance while powering

Figure 9.9 compares the energy consumption registered per second when continuously powering the tags at varying displacements, both with and without a saline solution covering the tags. In this figure, the tags are presented with displacements along the horizontal axis, and the relative energy consumption along the vertical axis. The energy consumption per second is based on Formula 8.4. The samples with and without a saline solution are shown with red and blue columns respectively. As shown by the figure, the energy consumption per second varies much between each individual sample. When looking at the results with the saline solution, the energy consumption seems to have been slightly reduced.

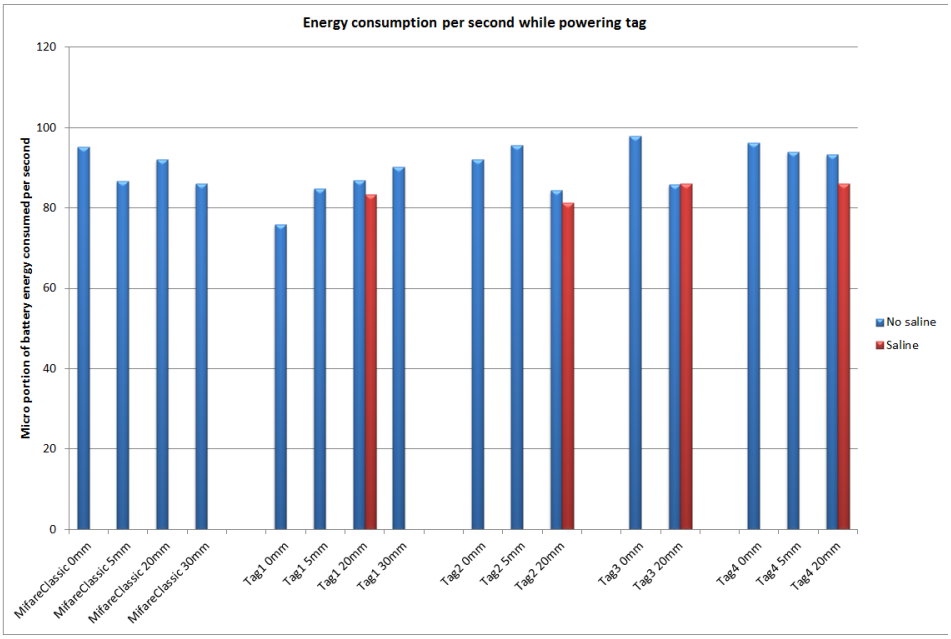


Figure 9.9: Energy consumption while powering tag

Part III

Conclusion

Chapter 10

Analysis

This chapter discusses the features described in Chapter 9. It aims at explaining why these features exist by deducing what created them. The sections in this chapter progress by describing the features identified for the three controlled variables: displacement, saline solution and tag type. Each section discuss the effect on communication performance and energy consumption. The term throughput is used in this context and refers to the amount of bytes that are successfully transferred from the sender to the receiver, and, therefore, transfers that fail reduce the overall throughput.

10.1 Displacement

The displacement between the tag and the reader is controlled in the experiment and defines the distance between the phone and the tag.

10.1.1 Communication performance

It is striking that changing the displacement does not seem to have any effect on the throughput. The only tag that seems to consistently acquire a higher throughput with a shorter distance is Tag3.

Two factors that are expected to affect the throughput are propagation time and noise on the communication channel. The propagation time is the time it takes for a signal to travel from the sender to the receiver. The signal speed of radio-waves in air is close to the speed of light, and the propagation time is therefore defined by the formula:

$$T_{Propagation} = \frac{d}{c}\{sec\},\ where \left| \begin{array}{l} d = disntance \\ c \approx 3 \times 10^8 ms^{-1} \end{array} \right. \tag{10.1}$$

Hence, the theoretical propagation time for changing the displacement by 20mm is: $\frac{0.03}{c} \approx 1 \times 10^{-10}$ seconds. This is such a small number that it is not expected to have any noticeable influence on the throughput.

The effect of noise is a factor that is expected to increase along with the displacement between the tag and the reader. As a result, the power experienced by the tag is reduced, and the noise therefore becomes more

Displacement (mm)	Signal strength reduction (dB)	Difference (dB)
5	−50.93	-
10	−44.91	6.02
20	−38.89	6.02
30	−35.37	3.52

Table 10.1: Signal strength reduction

prominent. The graph presented in Figure 10.1 is a fictional graph that illustrates how the noise affects the signal. The noise in the illustrated examples is the same. However, the power of the signal is reduced. As can be seen, when the power is reduced to one third of full power, the signal is already showing significant signs of influencing the results.

The *free-space path loss*[56] (FSPL) formula defines how a signal degrades with distance in free air and is summarized in the following formula:

$$FSPL = \left(\frac{4\pi d}{\lambda}\right)^2, where \left| \begin{array}{l} d = disntance \\ \lambda = wavelength \end{array} \right. \tag{10.2}$$

The FSPL in decibel is calculated with the formula[56]:

$$FSPL(dB) = 10log_{10}\left(\left(\frac{4\pi d}{\lambda}\right)^2\right) = 20log_{10}\left(\frac{4\pi d}{\lambda}\right) \tag{10.3}$$

The wavelength is defined as:

$$\lambda = \frac{c}{f}, where \left| \begin{array}{l} c \approx 3 \times 10^8 ms^{-1} \\ f = frequency \end{array} \right. \tag{10.4}$$

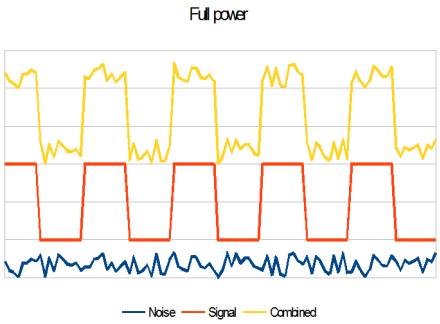
Therefore, NFC, with a frequency of $13,56MHz$, has a wavelength of $\frac{c}{13,56MHz} \approx 22,12$meters. Table 10.1 shows the reduction in signal strength with the displacements given for this experiment. As can be seen, the displacement does not incur a great loss in signal strength, which concurs with the observations from the experiment. Another notion to make is that for every doubling of the distance, the value drops with about $6dB$.

10.1.2 Energy consumption

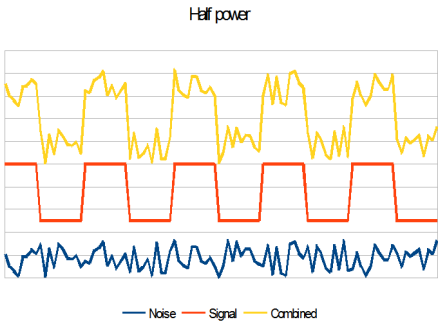
The overall energy consumption does not seem to be much affected by displacement. These observations closely match those for throughput, because the reduced need for retransmission leads to more stable energy consumption.

10.2 Saline solution

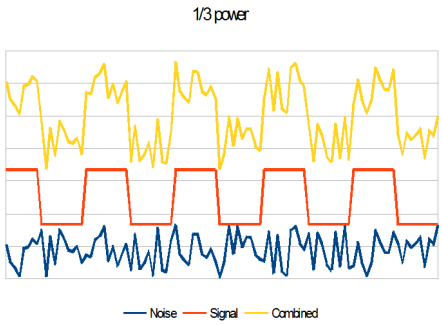
The saline solution introduces another media for the electromagnetic field to pass through. This field is expected to reduce the power of the field



(a) Full power



(b) Half power



(c) One third power

Figure 10.1: Power to noise ratio

and, thereby, increase energy consumption and reduce throughput. These expectations and the observations from the experiment are discussed for communication performance and energy consumption in the following subsections.

10.2.1 Communication performance

Surprisingly, the observed throughputs do not seem to be affected by the introduction of a seven millimeter thick saline solution.

The tags are submerged in a saline solution as described in Chapter 8, and the phone is tilted about 30°. This can result in the antenna on the reader being better aligned with the antenna of the tag. If the tag antenna is aligned to cover a larger surface area of the reader antenna, it results in a higher power experienced by the tag[15]. The higher power seems to cancel out the reduction in power due to the introduction of the saline solution. Further analysis should be conducted to confirm this hypothesis.

10.2.2 Energy consumption

It is expected that the saline solution significantly reduces power with distance. However, the results indicate otherwise. The assumed reason behind these observations is the same as for the throughput. The alignment with the antenna of the reader and the tag is assumed to produce a stronger electromagnetic field in the experiment conducted with the saline solution.

10.3 Tag type

This section compares the performance of the five tag types and discusses the most striking features of the tags.

10.3.1 Communication performance

The graph displayed in Figure 9.1 indicates that the RTT is not greatly dependent on the displacement from the tag. This seems reasonable as radio-waves travel with a speed close to the speed of light, and increasing the distance by a few centimeters does, therefore, not have a noticeable effect. The RTT packets are designed to be as small as possible, merely consisting of the packet overhead. The small size results in the noise on the channel having less impact as retransmission is less costly. However, the RTT shows the impact of the overhead of the packet and the computation time. When comparing the five tags, it can be seen that the NFC forum type tags have very similar RTTs of about 10ms. This differs greatly from the MifareClassic tag, which has an RTT of approximately 30ms. Compared to Tag1, with a RTT of about 8ms, the MifareClassic tag is almost four times slower. This indicates a higher cost for transmission of small packets for the MifareClassic tag compared to Tag1. This also means that for transmission of small packets, Tag1 seems to be the better option.

Tag	Bytes sent	Bytes received	Total bytes	Transfers
Tag1	9	3	12	1
Tag2	4	8	22	1
Tag3	18	31	49	1
Tag4	19	12	31	3
MifareClassic	18	21	39	2

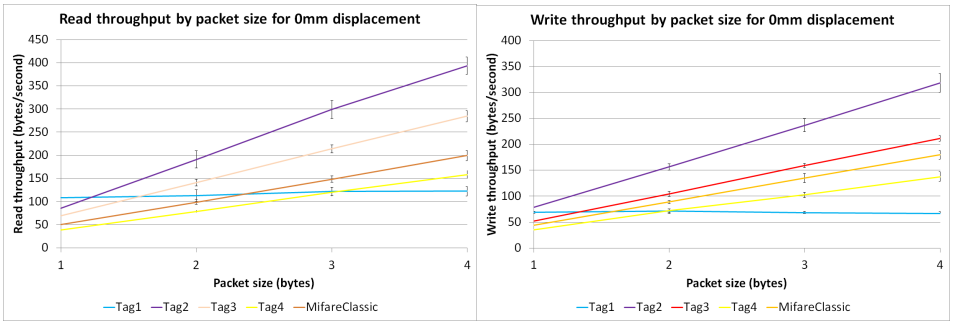
Table 10.2: Overhead for reading a byte

When comparing the overhead required to transfer packets of data, it becomes clear why Tag1 is more efficient. This relates well to the overhead of reading and writing small packets. Table 10.2 summarizes the minimum number of bytes transferred for reading a single byte of data from the memory of the five tags. The data in the table is extracted from observing data communication between the tag and the reader as the reading is performed. As can be seen, Tag1 has the smallest overhead for reading a single byte.

The results indicating that Tag1 is more suitable for smaller packets, is also confirmed when zooming in on the read and write communication performance for Tag1, as shown in Figure 10.2. One standard deviation from the calculated average is indicated by black error bars for each throughput value. The zoomed version only shows the throughput for 0mm displacement between the reader and the tag and without the saline solution. This is to make the results more readable. According to the figure, Tag1 is either better or close to the other tag types for reading and writing of one byte. The figure also shows that Tag1 rapidly drops in transmission efficiency compared to the other tags when the size increases. This is consistent for Tag1 for all packet sizes with the *read* command.

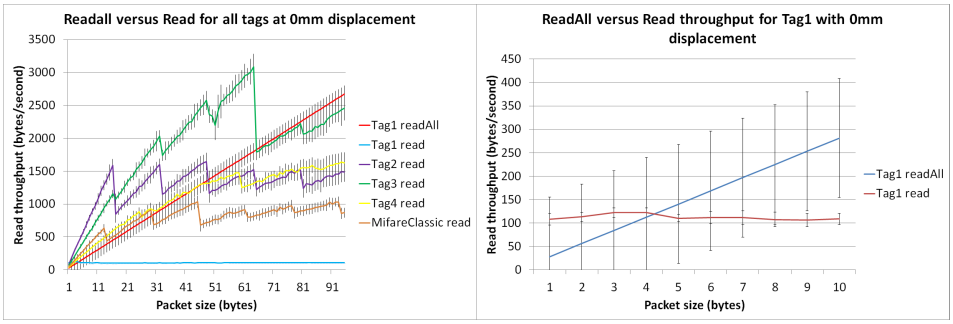
When comparing the throughputs with the overhead for packets of one byte it is clear that these show a strong relation. The only outlier is Tag3, which has a higher throughput compared to Tag4 and MifareClassic even though it also has a higher overhead. To understand this relation, it is important to consider the number of transfers. With MifareClassic, it is necessary to verify a sector on the tag before performing any read operations, which therefore result in two transfers. Reading from Tag4 requires three transfers: one for selecting the application on the tag, one for selecting the file on the tag and one for sending the actual command. It is clear from this comparison that multiple transfers incur a cost and reduce throughput.

Tag1 has a special command for reading all the tags memory, which has very little overhead and is called the *readall* command. The other tag types have general *read* commands that scale well as the size of the packet increases. However, they do not acquire the same throughput as Tag1 for reading the full memory. Figure 10.3a compares the communication performance with the *readall* command of Tag1 with the *read* commands of the other tags at a displacement of 0mm from the tag. The standard deviation is indicated with vertical black bars for each measurement.



(a) Read (b) Write

Figure 10.2: Read and write throughput for small packets at 0mm displacement



(a) Tag1 *readall* versus *read*. (b) Tag1 *read* vs Tag1 *readall*

Figure 10.3: Tag1 *readall* command performance

The graph indicates that the *readall* command of Tag1 outperforms the other tags when reading packets with a size greater than 64 bytes and smaller than 96 bytes. However, the results are not entirely conclusive when considering the standard deviation of the throughputs. Figure 10.3b compares the *read* and *readall* commands for Tag1 and shows that when reading more than four bytes, it is more efficient to use the *readall* compared to the *read* command.

The graphs in Figure 10.4 show the read and write throughputs without the saline solution and at a displacement of 0mm for all tags. These graphs distinctly show the peaks observed in Chapter 9.

The read throughputs for Tag2, Tag3, Tag4 and MifareClassic have peaks at certain packet sizes, which is not so strange when considering the design of the read command in the respective tags. The read command for Tag2 reads pages from the memory consisting of 16 bytes, and therefore, the pattern that emerges is that the peak performance is at intervals of 16 bytes read from memory. The exact same pattern is visible with the MifareClassic tag, only at a lower throughput. This can be explained by the fact that the MifareClassic tag is required to authenticate sectors before reading from them.

The disadvantage with page sizes of certain sizes is that the throughput

is dependent on the location in the memory to read from. With a page size of 16 bytes, reading 16 bytes from address zero in the tags memory, can be done in one read. However, to read 16 bytes from memory address one requires two reading operations of 16 bytes each, which reduce the throughput.

The peaks found with Tag4 repeat themselves at less regular intervals, and this fits well with its command design for reading. The command specifies that once a read command has been transmitted to the tag, the tag splits the response into transfers containing 59 bytes each. Hence, a transfer of 60 bytes requires two transfers compared to the single transfer required for 59 bytes. An advantage that Tag4 gives is that memory is not mapped to pages of any specific size, and therefore, starting at an offset into the memory should not incur a penalty on throughput.

Tag3 has a different pattern compared to the others as it has some distinct peaks inside its peaks. These can be explained by the tag offering an adaptable read command. This command can be modified to read from one to four blocks. Each block consists of 16 bytes of data and the read throughput should therefore have minor peaks at an interval of 16 bytes and major peaks at an interval of 64 bytes. These peaks are identified in the graph.

The write throughput acquired is very different from the read throughput for all but Tag1 and Tag4. The *write* commands for these tags are very similar to the *read* command. Writing to Tag1 is accomplished one byte at a time. Therefore, it has little change in throughput as the packet size increases. Write command for Tag4 has a limit of 52 bytes on the first transfer, but allows consecutive transfers to be of up to 59 bytes each.

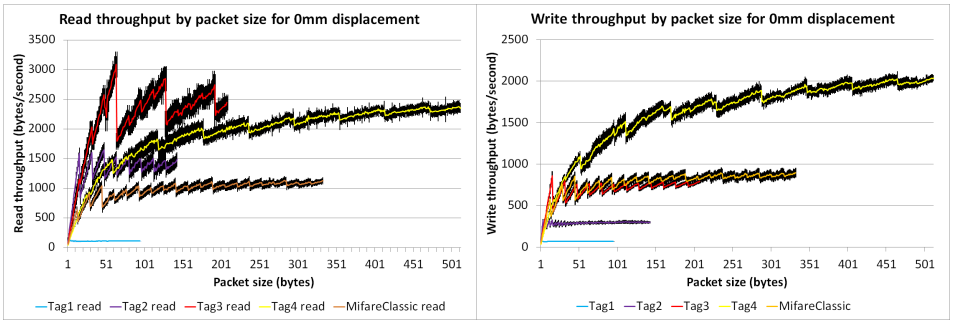
As is seen from the graphs, the write throughput for Tag3 and MifareClassic is rather similar. This fits with their write command structure as well, they both write with a size of 16 bytes. Therefore, they have performance peaks at intervals of 16 bytes.

Tag2 also have small peaks, although at much higher frequency. This can be explained by the *write* command enabling writing of 4 bytes at a time.

Another explanation to the overall reduction in throughput for reading operations compared to writing operations, is that writing to flash memory (the tags memory) is many times more time-consuming than reading form it.

10.3.2 Energy consumption

The energy consumption per byte for read and write operations varies greatly between each tag. However, the energy consumption per second seems to be rather consistent. This fits well with the throughputs measured for the respective tags. Tag1, with its *readall* command has a measured average throughput of 3440 bytes per second. Tag4 has a measured average throughput for reading its total memory of 2510 bytes per second. These values correlate well to the average battery consumption per byte transferred. If the energy consumption is unchanged and the throughput



(a) Read throughputs (b) Write throughputs

Figure 10.4: Communication performance at 0mm displacement without saline

Tag	0mm displacement			
	Read throughput (BPS)		Write throughput (BPS)	
	Estimated	Measured	Estimated	Measured
MifareClassic	1090	1090	880	895
Tag1	2590	3440	67.7	68.2
Tag2	1340	1490	295	298
Tag3	2090	2410	732	792
Tag4	2300	2510	2009	2034

Table 10.3: Estimated throughput from energy consumption and measured throughput

increases, the energy consumption per byte decreases. The formula below pressents the relationship between between energy consumption per byte, throughput and the energy consumption per second:

$$T_e = \frac{E_t}{E_b}, \; where \left| \begin{array}{lcl} E_t & = & Energy\,consumption\,per\,second \\ E_b & = & Energy\,consumption\,per\,byte \\ T_e & = & Estimated\,throughput \end{array} \right. \tag{10.5}$$

This formula is used in Table 10.3 to estimate the throughput based on the energy consumption per second and the energy consumption per byte. The calculated values are then compared with the actual measured throughput. The results reveal a striking correlation between the measured and estimated throughput. This is a strong indication that the RF activity consumes equal amount of energy, regardless of whether read or write operations are performed.

In Figure 10.5, the energy consumption per second is compared for both read, write and power operations. An interesting result is that the values indicate that energy consumption for merely powering is more expensive than performing I/O operations on the tag. When comparing the writing and powering operations of Tag4 at 0mm displacement, the powering is $1 - \frac{96.2}{71.9} = 33.7\%$ more expensive. Two theories are deduced from this, either the phone detects tags with a maximum power and then adjusts the power

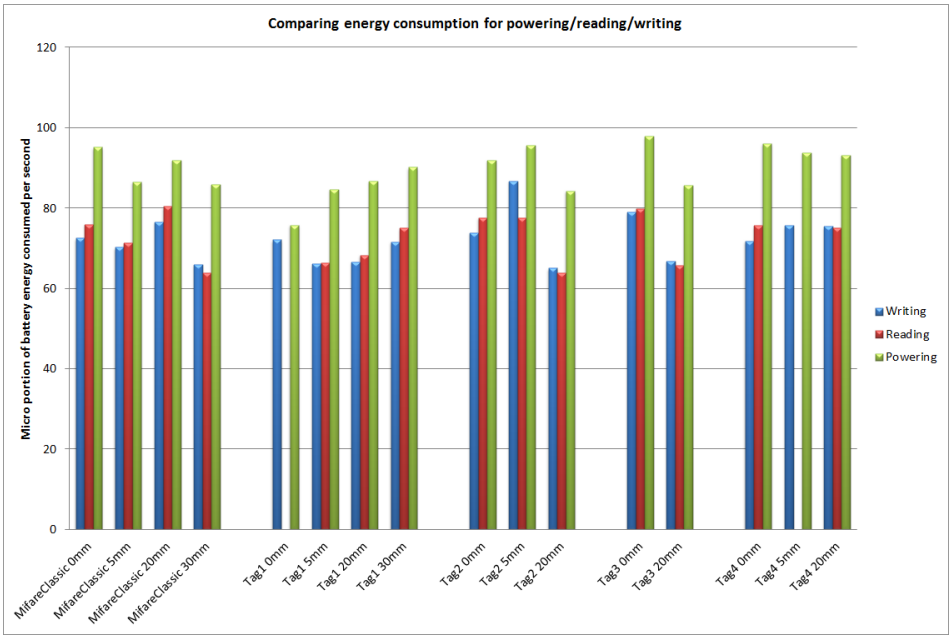


Figure 10.5: Comparing energy consumption for reading, writing and powering

to what is required for communication when the tag is detected, or the modulation of the RF field results in a reduction in energy consumption. When the reader transfers information to the tag, it modulates the RF field, this can be done by reducing the power transmitted to the RF antenna. If so, this results in a reduced energy consumption, which may be the effect detected in this experiment. It is also possible that the values in the graph represent a purely random effect, and to verify that this is not the situation, further benchmarking needs to be conducted.

10.4 Experiment conclusion

One of key features that this experiment tries to define is how the energy consumption can be characterized. The energy consumption influences the user a great deal. The user is expected to use a phone as a reader and a phone has a limited energy source. If the energy source depletes to quickly, the user experience degrades.

The results from the experiment indicate that the energy consumption is not dependent on the tag type or standard. However, further experiments are required to confirm this. If it is correct, the energy efficiency is more related to the throughput than the actual energy consumption per second. It is therefore important to consider this aspect in the design of the data communication, so that maximum throughput can be gained, and thereby the highest energy efficiency can be acquired.

Zaher's tag provides power to a number of sensors, and in turn, they provide it with information that the tag transfers to the reader. Therefore,

most of the time, the tag is transferring information to the reader. If Zaher is able to bundle the information transferred from the tag to the reader in such a way that it utilizes the full capabilities of Tag1, it is advised to base his tag on the NFC Forum Tag 1 Type. This is a simple tag to establish communication with, and it performs well when data is bundled correctly. Furthermore, it has a simple memory structure that can be extended up to 1KB, which enables a significant storage capacity.

NFC Forum Tag1 Type occupies a small range of commands and there is nothing preventing Zaher from expanding this command set with his own propfrietary commands. These could be used to establish reading of information directly from sensors or to change internal settings of the tag. If the tag collects information gained from the sensors over time in a buffer, custom commands can be created that read and clear these buffers. If the number of commands exceeds what can be represented by a byte, a variable-width code encoding can be used to extend the command set (such as with UTF-8). In this way, the only limit seems to be the available throughput.

Although the tag standard defines a maximum memory range, it is possible for the tag to have its own extended memory. This memory could be accessed through the proprietary command set and thereby allow the tag to store more data.

As the experiment showed, there are no conclusive results that indicate that energy can be saved by using the reader to power the tag and only transfer the information at given intervals. Rather this seems to have the opposite effect. I therefore advice Zaher to await further test results before designing the tag in this way. With the given results, it seems to be a better solution to enable the reader to use the tag to forward sensor values directly.

The saline solution is used as a substitute to human tissue and the results from the experiment does not indicate the expected results. However, limitations in the experiment design are identified, discussed and suggestions are proposed to improve the reliability of the results in Section 10.2.

Chapter 11

Accomplishments and future works

This chapter describes the accomplishments and suggests topics for further work related to the apps and the library developed in this thesis. One section is dedicated to each app and library.

11.1 NFC protocol tester app evaluation

The NFC protocol tester app is designed to assist tag developers in testing their proprietary commands. These commands are entered as byte values separated by a dot. The user enters the command and places the phone close enough to the tag that it is detected. The screen then displays the tags supported technologies. In the end, the user pushes the "Transmit command" button, and the tag's response to the command is displayed. This process is depicted in the pictures in Figure 11.1.

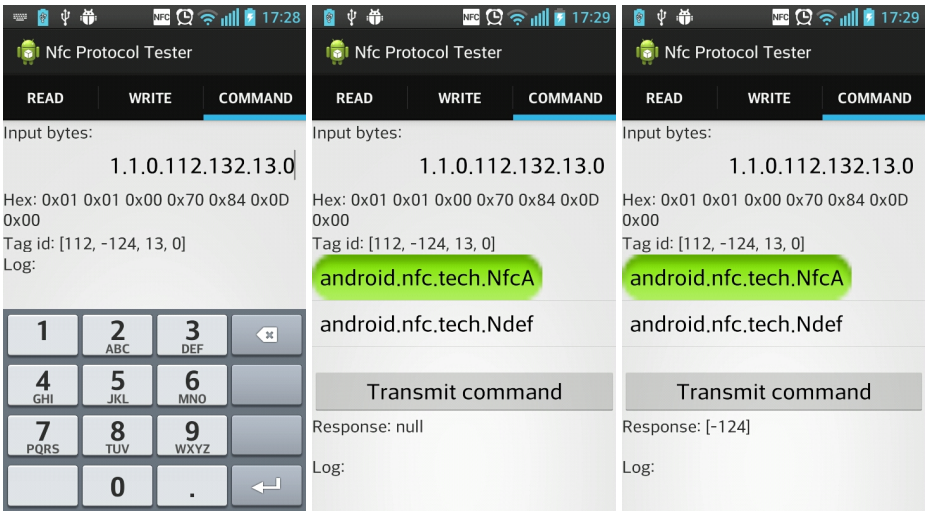
This section evaluates the accomplishments of this app and discusses what future work should be done to improve its usefulness.

11.1.1 Accomplishments

This app has accomplished transmission of user defined commands to a tag. It has also accomplished reading from and writing to the tags memory. The app enables transfers of commands to any tag type as long as it is supported by the NFC communication library. The app is designed to be used in the development of Zaher's tag to test its command set. This tag is not yet produced and thus the functionality has been tested with a NFC Forum Tag 1 Type, which is able to execute the command set defined by this standard and responds as expected.

11.1.2 Future improvements

Some features are desirable, such as inputting the commands in hexadecimal instead of dot delimited byte values. Additionally, it could be desirable to enable users to store the created commands so that it is not needed



(a) Command input (b) Selecting technology (c) Command transferred

Figure 11.1: NFC protocol tester app

to construct them multiple times. Another feature could be to enable the user to pair commands with expected results, to allow the app to execute sets of these commands in sequence and observe if the responses match, and report the ones that do not match. This could be used to enable automatic testing of the produced tags to ensure that newly added changes do not affect the behavior of already tested commands. This enables a leaner development approach to designing the command set of tags.

In the development phase of a tag, it is not always desirable to implement the full CRC check before testing the tags command set. This is because implementation of calculating the CRC on a IC chip is an extensive piece of work. Therefore, it could be desirable to add a feature that allows the app to read the response even though the CRC fails. In the development of Zaher's tag, the initial tests do not require CRC checks. However, this feature requires extensive work to be done with the NFC Communication library as this unit then needs to use its own native libraries to communicate with the NFC hardware and enable the forwarding of data even though it does not pass the CRC.

11.2 NFC benchmark app evaluation

The NFC benchmark app enables automatic benchmarking of all tag types that are supported by the NFC communication library developed in this work.

11.2.1 Accomplishments

This app enables automatic benchmarking of tags and this is firmly tested through the experiment described in Chapter 8. Seven benchmarking pro-

cedures are automated through the app, and an additional two benchmarking procedures are available for testing by being run manually. This app also enables storing of benchmarking results to a database for further analysis. It has through this work been used as a tool for supporting Zaher in developing his tag, which was one of the key objectives.

11.2.2 Future improvements

The results gained from observing the power consumption display great variation, which indicates inaccurate measurements. This seems to be related to the method of monitoring the battery consumption. There are systems that give more accurate results by observing built-in battery voltage sensors such as described here [64]. These systems could significantly improve the accuracy of the estimated energy consumption of tag operations.

11.3 NFC communication library

The NFC communication library is designed to enable simplified communication with NFC tags. In this section, the accomplishments related to this library and future works are described.

11.3.1 Accomplishments

The library is extensively used by two apps to provide communication with five NFC tags based on five different standards. The library allows apps to perform read and write operations, both in the form of NDEF messages and binary data, to the tags memory. It also allows apps to send their own constructed commands to the tag and receive a response.

11.3.2 Future improvements

Migration of this library towards a service solution could enable other apps to share its functionality such as described in Chapter 4. The library is currently tailored to work with the tags introduced in Chapter 3. However other tags exist and further work should be aimed at enabling communication with these as well.

Bibliography

[1] Grant Allen and Mike Owens. *The Definitive Guide to SQLite*. Apress, 2010.

[2] *An introduction to video content analysis industry guide*. BSIA - british security industry association. 2009.

[3] Jose Bravo et al. 'Enabling NFC Technology for Supporting Chronic Diseases: A Proposal for Alzheimer Caregivers'. In: *must find journal* (2008), p. 17.

[4] Gregor Broll et al. 'Improving the Accessibility of NFC/RFID-based Mobile Interaction through Learnability and Guidance'. In: *must find journal* (2009), p. 10.

[5] Barry Burd. *Android Application Development All-in-One For Dummies*. Hoboken, NJ: John Wiley & Sons, 2011.

[6] Ed Burnette. *Hello, Android: Introducing Google's Mobile Development Platform*. 2nd. Pragmatic Bookshelf, 2009. ISBN: 1934356492, 9781934356494.

[7] Hames Cadle and Donald Yeates. *Project Management for Information Systems*. fifth edition. Pearson, 2008.

[8] Aage Dahl. Version git checkout bddf7bd. URL: https://bitbucket.org/aahdahl/no.as.gold.utils.simplemessenger.

[9] Aage Dahl. Version git checkout 4004a7c. URL: https://bitbucket.org/aahdahl/no.as.gold.nfc.communication.

[10] Aage Dahl. Version git checkout 51bfabe. URL: https://bitbucket.org/aahdahl/no.as.gold.nfc.

[11] Aage Dahl. Version git checkout aeaf02a. URL: https://bitbucket.org/aahdahl/no.as.gold.nfc.benchmark.

[12] Gauthier Van Damme and Karel Wouters. 'Practical Experiences with NFC Security on mobile Phones'. In: (2009).

[13] Android Developers. *Fragments*. Last accessed: 29.03.14. Google. URL: http://developer.android.com/guide/components/fragments.html.

[14] David Dressen. 'Considerations for RFID Technology Selection'. In: *Atmel Applications Journal* (2004), pp. 45–47.

[15] Klaus Finkenzeller. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification*. 2nd ed. New York, NY, USA: John Wiley & Sons, Inc., 2003. ISBN: 0470844027.

[16] NFC Forum. *Bluetooth Secure Simple Pairing Using NFC*. 18.10.2011. URL: http://www.nfc-forum.org/resources/AppDocs/NFCForum_AD_BTSSP_1_0.pdf.

[17] NFC Forum. *Type 1 Tag Operation Specification*. NFC Forum. 13th Apr. 2011.

[18] NFC Forum. *Type 2 Tag Operation Specification*. NFC Forum. 31st May 2011.

[19] NFC Forum. *Type 3 Tag Operation Specification*. NFC Forum. 28th June 2011.

[20] NFC Forum. *Type 4 Tag Operation Specification*. NFC Forum. 28th June 2011.

[21] K. R. Foster and H. P. Schwan. 'Dielectric Permittivity and Electrical Conductivity of biological Materials'. In: *CRC Handbook of biological Effects of Electromagnetic Fields*. Ed. by C. Polk and E. Postow. 1986, pp. 25–96.

[22] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[23] Eric Freudenthal et al. 'Suitability of NFC for Medical Device Communication and Power Delivery'. In: *must find journal* (2007), p. 4.

[24] Stefano Levialdi Ghìron et al. 'NFC Ticketing: A Prototype and Usability Test of an NFC-Based Virtual Ticketing Application'. In: *First International Workshop on Near Field Communication* (2009), pp. 45–50.

[25] Google. *Android*. Last accessed 20.05.2013. URL: http://developer.android.com/index.html.

[26] Javier Gozalvez. 'First Google's Android Phone Launched'. In: *IEEE VEHICULAR TECHNOLOGY MAGAZINE* (2008), p. 8.

[27] Jeffrey Hightower and Gaetano Borriello. 'A Survey and Taxonomy of Location Systems for Ubiquitous Computing'. In: (2001).

[28] Nadav Hochman and Raz Schwartz. 'Visualizing Instagram: Tracing Cultural Visual Rhythms'. In: *AAAI Technical Report WS-12-03 Social Media Visualization* (2012), pp. 6–9.

[29] *Identification cards - Contactless integrated circuit(s) cards - Proximity cards*. Norm. 2000.

[30] *Identification cards - Contactless integrated circuit(s) cards - Proximity cards - Part4: Transmission protocol*. Norm. 2001.

[31] Antonio J. Jara et al. 'Evaluation of the security capabilities on NFC-powered devices'. In: *must find journal* (2010), p. 9.

[32] *JIS X 6319-4 - Specification of implementation for integrated circuit(s) cards - Part 4: High Speed proximity cards*. Japanese Standards Association, 20th July 2005.

[33] CoryD Kidd et al. 'The Aware Home: A Living Laboratory for Ubiquitous Computing Research'. In: *Cooperative Buildings. Integrating Information, Organizations and Architecture*. Lecture Notes in Computer Science 1670 (1999). Ed. by NorbertA Streitz et al., pp. 191–198. DOI: 10.1007/10705432_17. URL: http://dx.doi.org/10.1007/10705432%5C_17.

[34] Josef Langer, Christian Saminger and Stefan Grunberger. 'A COMPREHENSIVE CONCEPT AND SYSTEM FOR MEASUREMENT AND TESTING NEAR FIELD COMMUNICATION DEVICES'. In: *must find journal* (2009), p. 6.

[35] Lazierthanthou. *SQLite Manager*. Last accessed: 15.04.14. Mozilla. URL: https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/.

[36] Richard G. Mair. 'Protocol-Independent Detection of Passive Transponders for Near-Field Communication Systems'. In: *must find journal* (2010), p. 6.

[37] R. Meier. *Professional Android 4 Application Development*. Wrox Professional Guides. Wiley, 2012. ISBN: 9781118102275. URL: http://books.google.no/books?id=bmJIl%5C_wPgQsC.

[38] *MF1 IC S50 - Function specification*. Product data sheet. NXP Semiconductors, 15th Jan. 2007.

[39] Thomas B. Moeslund et al., eds. *Visual Analysis of Humans - Looking at people*. Springer, 2011.

[40] Juergen Morak et al. 'Feasibility of mHealth and Near Field Communication Technology based Medication Adherence Monitoring'. In: *must find journal* (2012), p. 4.

[41] Nokia. *Introduction to NFC*. Nokia, 19th Apr. 2011.

[42] *Open Handset Alliance*. Last accessed: 30.03.14. URL: http://www.openhandsetalliance.com/.

[43] Charl A. Opperman and Gerhard P. Hancke. 'Using NFC-enabled Phones for Remote Data Acquisition and Digital Control'. In: *must find journal* (2011), p. 6.

[44] Marc Pasquet. 'Secure payment with NFC mobile phone in the SmartTouch project'. In: *Collaborative Technologies and Systems* (2008), pp. 121–126.

[45] David Michael Peterson. 'Tissue equivalent phantom development for biomedical applications'. In: (2009).

[46] R. Pethig. 'Dielectric properties of body tissues'. In: *Clin. Phys. Physiol. Meas.* 8 (1987), pp. 5–12.

[47] Android Open Source project. *Android Developer Tools*. Last accessed: 17.04.14. Google. URL: http://developer.android.com/tools/help/adt.html.

[48] Android Open Source project. *Android Development official web page*. Last accessed: 02.04.14. Google. URL: http://developer.android.com/.

[49] Android Open Source project. *Communicating with Other Fragments*. Last accessed: 17.04.14. Google. URL: http://developer.android.com/training/basics/fragments/communicating.html.

[50] Android Open Source project. *Processes and Threads*. Last accessed: 17.04.14. Google. URL: http://developer.android.com/guide/components/processes-and-threads.html.

[51] Jukka Riekki, Ivan Sanchez and Mikko Pyykkonen. 'NFC-Based User Interfaces'. In: *must find journal* (2012), p. 7.

[52] Hiroshi Sakai and Akira Arutaki. 'Protocol Enhancement for Near Field Communication (NFC): Future Direction and Cross-Layer Approach'. In: *must find journal* (2011), p. 6.

[53] NXP Semiconductors. *NTAG203 - NFC Forum Type 2 Tag compliant IC with 144 bytes user memory*. NXP Semiconductors. 12th Dec. 2011.

[54] Philips Semiconductors. *mifare DESFire Contactless Multi-Application IC with DES and 3DES Security MF3 IC D40*. Norm. Philips Semiconductors. Philips Semiconductors, Apr. 2005.

[55] James C. Sheusi. *Android application development for Java programmers*. Course Technology PTR, 2013.

[56] Mario Marques da Silva. *Multimedia Communications and Networking*. CRC Press, 2012.

[57] Ryan Smith. *Using Mifare Classic Tags*. Skyetek Inc, Apr. 2005.

[58] Esko Strömmer et al. 'Application of Near Field Communication for Health Monitoring in Daily Life'. In: *must find journal* (2006), p. 4.

[59] Busra Ozdenizci Vedat Coskun Kerem Ok. *NEAR FIELD COMMUNICATION: FROM THEORY TO PRACTICE*. John Wiley & Sons, Inc., 2012. ISBN: 978-1-119-97109-2.

[60] Deepak Vohra. *Java EE development with Eclipse*. Packt Publishing, 2012.

[61] A Voss et al. 'Influence of Low Sampling Rate on Heart Rate Variability Analysis Based on Non-linear Dynamics'. In: *Computers in Cardiology* (1995), pp. 689–692.

[62] Mark Weiser. 'The computer for the twenty-first century.' In: *Scientific American* (1991), pp. 94–104.

[63] Karim Yaghmour. *Embedded Android*. Sebastopol, CA: O'Reilly Media, 2013.

[64] Lide Zhang et al. ‘Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones’. In: (2010).