St. Pölten University of Applied Sciences

/medien & digitale technologien

/fh/// st.pölten

UNITA - Ultrasonic Network for IoT Applications

Smart Ultrasonic Beacons for the Communication of IoT Devices

Diploma Thesis

For attainment of the academic degree of

Dipl.-ing.für technisch-wissenschaftliche Berufe

in the Masters Course Digital Media Technology at St. Pölten University of Applied Sciences, **specialized area Mobiles Internet**

Submitted by:

Florian Taurer, BSc

dm171557

Advisor and First Assessor: FH-Prof. Dipl.-Ing. Mag. Dr. Matthias Zeppelzauer

Second Assessor: Alexis Ringot, master

Zeillern, 15.01.2020

Declaration

- The attached research paper is my own, original work undertaken in partial fulfillment of my degree.

- I have made no use of sources, materials or assistance other than those which habe been openly and fully acknowledged in the text. If any part of another person's work has been quoted, this either appears in inverted commas or (if beyond a few lines) is indented.

- Any direct quotation or source of ideas has been identified in the text by author, date, and page number(s) immediately after such an item, and full details are provided in a reference list at the end of the text.

- I understand that any breach of the fair practice regulations may result in a mark of zero for this research paper and that it could also involve other repercussions.

Date: _____ Signature: _____

Acknowledgement

I would first like to thank my thesis advisor FH-Prof. Dipl.-Ing. Mag. Dr. Matthias Zeppelzauer of the Institute of Creative\Media/Technologies at University of Applied Sciences St. Pölten. The door to Prof. Zeppelzauer's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it.

I would also like to acknowledge Alexis Ringot master at the University of Applied Sciences St. Pölten of the Institute of Creative\Media/Technologies as the second reader of this thesis, and I am gratefully indebted to his very valuable comments on this thesis.

Further, I would like to thank Netidee for supporting me with a scholarship, which helped me a lot to finance the hardware of my master thesis.

I would also like to thank the experts who were involved in the hardware research, assemble and tests for this research project, as well as the experts who were involved in the user study design: Armin Kirchknopf BA MA BSc, Dipl.-Ing. Christoph Braun BSc, Dr. Victor Adriel de Jesus Oliveira MSc and Dipl.-Ing. Stefanie Größbacher BSc. Without their passionate participation and input, the validation survey could not have been successfully conducted.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Abstract

Internet of Things (IoT) got an important field of research and development in the last years. IoT devices often use Bluetooth or Wi-Fi for communicating with the Internet or other devices. In recent years, ultrasound has become a new additional way of communicating with devices. Currently, there are no open source solutions for building beacons using ultrasound as an alternative communication channel. The master thesis UNITA provides a hardware and software solution for building devices and a whole infrastructure using ultrasonic communication. UNITA includes the creation of the hardware with a construction manual for the ultrasonic communication device, called UNITA beacon. Further, a SDK and a server basis for creating applications on those beacons got implemented. UNITA beacons represent a novel communication endpoint for many use cases. One of those use cases is the ultrasonic blackboard SocialWall, which was developed with the UNITA SDK as a proof-of-concept application. Additionally, the evaluation of SocialWall was part of the thesis. The result of UNITA was a hardware concept with a first prototype. On the software side, a SDK was created and published, as well as server basis. In addition, user studies were fulfilled. They showed that ultrasound is an interesting and innovative way of communicating with devices, which is well accepted and also trusted by the users, though security still plays an important role for them. UNITA is open source and can be further developed by anybody, as well as used for developing application like SocialWall.Further, UNITA could be an important new area for the industry and developed more into that direction.

Kurzfassung

Internet of Things (IoT) wurde über die letzten Jahre hinweg zu einem wichtigen Feld für Forschung und Entwicklung. IoT Geräte nutzen of Wi-Fi oder Bluetooth zur Kommunikation mit dem Internet oder anderen Geräten. In den letzten Jahren entwickelte sich Ultraschall zu einem neuen zusätzlichen Weg mit anderen Geräten zu kommunizieren. Momentan gibt es keine Open Source Hardware- und Software-Lösungen für das Erstellen von eigenen Beacons, die Ultraschall als alternativen Kommunikationskanal nutzen. Die Diplomarbeit UNITA stellt nun eine Lösung für sowohl Hardware, wie auch Software, zur Verfügung. Diese Lösung soll es möglich machen, eigene Geräte und die Infrastruktur für Ultraschallkommunikation zu bauen. UNITA umfasst die Erstellung der Hardware mit einer Bauanleitung für die Ultraschallkommunikationsgeräte, genannt UNITA Beacon. Weiters, wurde ein SDK wie auch ein Server entwickelt, um eigene für die Beacons zu programmieren. UNITA Beacons stellen einen neuartigen Kommunikationsendpunkt für viele Bereiche dar. Einer dieser Einsatzbereiche ist eine Art Schwarzes Brett mit Ultraschall mit dem Namen "SocialWall", welches als Machbarkeitsstudie umgesetzt wurde. Zusätzlich war die Evaluierung von SocialWall Teil der Diplomarbeit. Das Ergebenis von UNITA sind ein Hardwarekonzept für Ultraschallbeacons, wie auch ein Prototyp eines Beacons. Weiters, wurde ein SDK implementiert und zusätzlich ein Server erstellt. Die Evaluierung mit Nutzern zeigte, dass Ultraschall einen interessanten und innovativen Weg der Kommunikation darstellt. Diese Kommunikationsmethode wurde bei den Nutzern gut angenommen und weiters hat sich auch gezeigt, dass sie Ultraschall Vertrauen. Trotzdem ist die Sicherheit der Nutzerdaten ein wichtiger Anknüfungspunkt für weitere Schritte. UNITA ist Open Source Software und kann von jedem weiterentwickelt werden, wie auch von jedem für die Entwicklung von neuen Applikation verwendet werden. UNITA stellt auch einen wichtigen und interessanten Bereich für die Industrie dar.

Contents

1 Introduction

The area of Internet of Things (IoT) gained increasing importance in recent years. More and more everyday devices become smart and connected. Most of today's IoT devices communicate via Wireless LAN or Bluetooth, which do not provide a seamless experience. This thesis focuses on the development and construction of an Ultrasonic Network for IoT Applications (UNITA). That is, an SDK for developing applications and a construction manual for ultrasonic beacons. With these beacons, it is possible to create a network which can be used to seamlessly connect smartphones, smartwatches and other IoT devices with each other and the Internet.

Ultrasonic beacons represent a novel communication endpoint for the Internet and would especially enable low-threshold devices (as often used in IoT) to connect to the Internet. Communication via sound is thereby very convenient as, in contrast to Bluetooth, no device pairing is required and, in contrast to radio-based NFC, no expensive sensors are needed. This infrastructure provides an alternative way of communicating for IoT devices, e.g. to seamlessly send and receive data over sound, pair devices automatically and connect with the Internet for data exchange. Possible applications include location-based services, mobile payments, secure authentication, home automation, and device synchronization.

In general, ultrasonic communication can be considered a novel additional communication channel to existing ones like Wi-Fi, Bluetooth and NFC which is cheap and easy to use. In future, a combination of different channels could increase the quality of various services, such as the accuracy of location tracking (Legendre, 2015) for navigation and the improved security of adding sound as an additional channel for multi-factor authentication.

Currently, there are no open source SDKs for implementing IoT applications using ultrasonic communication, as well as no open source beacon specialized on ultrasonic communication. The master thesis should answer the question of how an ultrasonic beacon could be developed with simple hardware, how well such a beacon works and to which degree users accept and trust the beacon and its communication. Further, the overall goal of UNITA is to build a first open hardware ultrasonic beacon to send and receive data via inaudible signals and to create an open source software basis using ultrasonic communication for developers.

This thesis addresses the following major goals for the realization of a first open source ultrasonic beacon:

- Easily reproducible hardware should be used.
- The costs and energy consumption should be kept at a minimum.

- Building ad-hoc ultrasound networks, should be possible.
- The implementation of a software infrastructure (server and SDK) to enable information sharing should be achieved.
- The correct operability of a proof-of-concept mobile and a ultrasonic beacon application should be validated.
- The acceptance and trustfulness of ultrasonic communication via a user study should be evaluated.

The development process behind the work in this thesis was follows:

- Literature and hardware research, leading to a set of suitable components
- Creation of sketches of the SDK
- Interface to the open source ultrasonic communication protocol SoniTalk
- Design and Elaboration of Use cases
- Finish of a first version of the SDK
- Implementation and testing of one use case as a proof-of-concept application
- Hardware tests, evaluation, improvement and retests
- Realization and evaluation via a user study

Through working on the open source project SoniTalk, much prior knowledge about ultrasonic communication could be gathered. The goal in SoniTalk was the development of a protocol for data-over-sound and a proof-of-concept demo application. UNITA uses this protocol as a base for sending and receiving ultrasonic signals.

The thesis is structured as follows. Chapter 2 gives an overview of ultrasound, existing technologies, and related projects. It begins with a brief overview of ultrasound characteristics, its modulation forms and usage as a network information carrier. Next, several available frameworks and ultrasonic applications in different areas are proposed, leading to and highlighting security aspects of ultrasonic communication. Further, a brief overview of other communication technologies as well as the OSI-model is described. Chapter 3 begins with a general overview and then focuses on the hardware implementation process, as well as the software development. Each part gives information about the requirements, challenges, problems, the process itself and results of the UNITA component. Furthermore, the general areas of use and the other use cases created for developing the SDK are described in chapter 4. This leads to the implementation of the proof-of-concept demonstrator application "SocialWall", further described in chapter 4. The evaluation of the software, as well as the hardware, becomes the main topic of chapter 5, which further leads to a discussion about the whole UNITA project results, including possible extensions in chapter 6, and ending with the conclusion in chapter 7.

2 Background, Theory and Related Works

2.1 Ultrasound

Smartphones are able to listen to (almost) inaudible sounds. Most microphones and loudspeakers are capable to receive and send ultrasound up to 20kHz, whereas humans perceive sound only up to 18kHz (Wang et al., 2016). Most humans over 18 years already cannot hear frequencies above 17Khz (Deshotels, 2014). Different speakers have different levels of frequency selectivity. Therefore, some speakers are more suitable for specific frequencies than others. The same is true for the microphones, which are rather simple because of the form factor of smartphones. In general, speakers and microphones show reduced sensitivity at higher frequencies (Wang et al., 2016).

By analyzing different sounds, it shows that the characteristics of time and frequency depend on the type of audio. Therefore, these characteristics play an important role when decoding embedded ultrasonic information. Connected with that, the ambient noise can cause significant interferences. Wang et al. (2016) measured different environments and found out that most ambient noise frequencies are lower than 2kHz. Their conclusion of testing ambient noise was to use frequencies higher than 8kHz, as ambient noise higher than 8khz is weaker and does not have a strong impact anymore.

Sound usually contains wide spectrum of frequencies and can be formally characterized as a sum of waves of those frequencies. Arp et al. (2017) classified and described the spectrum into three frequency bands: infrasound, audible sound and ultrasound. Infrasound is difficult to produce on small devices, because of a relatively long wavelength. Humans can normally not hear this frequency range. Depending on the persons' age, audible sound between 20Hz and 20kHz can be recognized. The upper bound gets lower the older the receiving person gets. Due to a small wavelength, ultrasound with frequencies higher than 20kHz can be emitted from small devices and are a good base for transmission. Those frequencies are inaudible for humans. Commodity hardware is normally designed for a frequency band of 20Hz to 20kHz, which leaves a range of 18kHz to 20kHz for inaudible data transmission, also called near-ultrasonic frequency range.

2.1.1 Modulation

The following section shows different kinds of modulation schemes, which are needed to transport information via a carrier medium. Starting with the most common digital modulation schemes, which are amplitude-shift keying (ASK), phase-shift keying (PSK) and

frequency shift keying (FSK) (Schiller, 2001). Further, enhanced and combined modulation schemes follow afterwards (Li et al., 2008).

2.1.1.1 Amplitude Shift Keying (ASK)

ASK is one of the simplest modulation algorithms. In this scheme, the values 0 and 1 are represented via different amplitudes of the sine wave. Though it just needs a very small bandwidth, it is very failure-prone to noise and attenuations which strongly decreases the amplitude of the signal (Schiller, 2001). If only two symbols are used, it is also called Binary ASK or sometimes also referenced as on-off keying (OOK) (Li et al., 2008).

2.1.1.2 Frequency Shift Keying (FSK)

FSK is a typical modulation scheme for wireless communication and uses in its simplest form called Binary FSK (BSFK) one frequency for binary 0 and one frequency for binary 1. This can then be sent as a sequence of frequencies, whereas sudden phase shifts should be avoided here. For de-modulation, two bandpass filters are applied, those two frequencies get checked and the stronger one is then decoded to the binary value. (Mobilkommunikation, Schiller) The bandwidth of FSK depends on the space between the carrier frequencies representing the symbols (Li et al., 2008).

2.1.1.3 Phase Shift Keying (PSK)

PSK uses phase jumps to encode binary data. A simple implementation would be the jump between 0 degrees and 180 degrees, which is a binary 0 or 1. That kind of PSK is also called Binary PSK (BPSK). It consists only of one carrier frequency and is more robust as it uses the phase-locked loop circuit (PLL) for synchronisation and reconstruction. Though it is more difficult for the sender and receiver to use (Schiller, 2001).

2.1.1.4 Enhanced and combined modulation techniques

Basic modulation schemes are limited to a relatively small throughput. Therefore, enhanced modulation schemes were created and further, different basic schemes can be combined to increase the effectiveness of a scheme. Yan et al. (2007) investigated an acoustic Orthogonal Frequency-Division Multiplexing (OFDM) transmitter and receiver. A bandwidth of 5kHz around a carrier frequency of 12.5kHz was used. An adopted synchronisation preamble with 512 subcarriers was chosen. 1024 subcarriers were used for each OFDM symbol by using a zero-padded OFDM. A Quadrature Phase-Shift Keying (QPSK) was applied for each data subcarrier. The system is using a TMS320C6713 digital signal processing (DSP) board with an A/D interface. The C6713 handles the sampling and the generating of the sound via the audio codec TLV320AIC23 which runs at a sampling rate of 44.1kHz.

Traditional radio frequency transmission cannot be transmitted through the metal casing. Hosman et al. (2010) developed a multi-tone FSK (MFSK) to counteract that gap. MFSK is perfect for fading channels, as it is a variation of FSK. On ultrasonic transmission through materials like metal, an extreme fading effect can be seen. This effect occurs because of multiple reflections and multiple Lamb waves through the channel. In general, MFSK uses different combinations of frequencies for encoding and can be seen as a set of matched filters. On the receiver side, a simple fast Fourier transformation (FFT) is used to decode the signal by checking the threshold of the chose frequencies. Hosman et al. (2010) used the band of 280 to 320kHz on a steel beam and installed an inexpensive LM567 tone detector and a DSP chip, which is set at low power wait mode. The MFSK symbols were generated with a Tektronix 81150A. Though the data rate is rather low, the reliability of receiving and transmitting data through steel was high.

Lopes and Aguiar (2003) worked on aerial acoustic device-to-device communication in ubiquitous computing applications. Their project called Digital Voices adapted common modulation schemes, which are mainly ASK and FSK. Based on amplitude shift keying, an 8-frequency coding scheme starting at 1kHz was chosen and the respective frequencies were selected based on a pentatonic scale, which leads to a more comfortable sound. Another approach on ASK is Harmonic ASK, which adopts 70Hz harmonics over 128 frequencies starting at 700Hz. This method results in a higher bit rate on using the same signal burst length. On the frequency shift keying side, harmonic FSK was tested and 256 frequencies in a harmonic interval of 20Hz were used to transmit 8-bit symbols. The base frequency was at 1,000Hz and every symbol lasted 20ms. Further, frequency hopping was tried out. There the channel's carrier frequency changes over time.

2.1.2 Acoustic Networks

Sound can be used to set up acoustical networks via available devices. Several projects already researched on this field and tried to build acoustic networks with various functionalities as follows.

Hanspach and Goetz (2013) created a covert acoustical mesh network and figured out a list of features which needs to be fulfilled. A device should work as a sender or a receiver with access to the input and output process. Further, it should not yet be part of a network and be able to communicate in a stealth way to not reveal the covert communication channel. Not only covert communication between two partners was addressed, but also multi-hop communications between more than two systems. This allows the communication to increase the range of signals significantly. The aim is to send between two systems which are not connected yet. After testing such a network with business laptops and HD Audio Controllers, which have a frequency range of 0 to 35kHz, Hanspach and Goetz (2013) were able to show that the mesh air network is feasible.

Audio networking represents a fast way of exchanging data, by eliminating the complexity of other technologies. Two smartphones, which are held close together, can already exchange information on low amplitude to limit the distance for security reasons. Further, a session pin could be sent to keep the communication secure. Madhavapeddy et al. (2005) faced two main threats for such audio networks. An attacker could either replay a pre-recorded pin as a man-in-the-middle attack or learn the session pin to decode the whole message. Though, this would require recording and transmitting hardware to get a nicely recorded message of the conversion, which makes it less possible. In addition, replaying a message would affect both user smartphones, which results in an obvious and therefore useless attack. Moreover, Madhavapeddy et al. (2005) used a dual-tone multifrequency scheme to bypass a band-limit of 3kHz in telephone calls. Furthermore, they created a pairing concept for two computers. Two smartphones get connected via Bluetooth with the computers and the actual pairing message gets sent via ultrasound over a telephone call. The audio network only acts as a platform for pairing, whereas the whole remaining data transmission happens over the Internet.

Sun et al. (2016) proposed a method for obtaining information from devices. Therefore, smartphones were selected for constructing a hidden data transmission network. Two ways of communication were researched, electromagnetic waves and acoustical signals. The system was set up with a computer emitting electromagnetic waves with the help of frequency modulation (FM) and a smartphone microphone receiving those signals. Further, the receiver smartphone can send the information via an acoustical signal to any other smartphone in reach. On the transmitter side, the computer uses a FM carrier frequency of 81.7MHz and carrier frequencies of 2000Hz and 3000Hz on the audio emitting part. The FM signals were received by the FM antenna of the smartphone and demodulated by the FM chip of the receiver. A fast Fourier transformation (FFT) was then used to get the frequency spectrogram of the received message. This system infrastructure resulted in a transmission rate of 9.1 bit/s and a maximum distance of seven meters with high accuracy of 95%. Though it is limited, it is sufficient for the transmission of private data.

Ortega et al. (2014) presented a local area network using acoustic communication as the physical layer. The architecture features any device, like mobile phones, notebooks or tablets, which has loudspeakers as emitters, microphones as receivers and a sound media channel. For communication, a time-hopping code division multiple access (CDMA) modulation was implemented allowing bi-directional communication via two separate channels. Messages were sent in time-slots with the assistance of a cryptographically pseudo-random number generator. As only the receiver has the correct key it allows a secure point-to-point and multicast communication. Further, the system needs asymmetric error correction and synchronisation to properly work and avoid collisions. The experimental results have shown a low error rate, which is acceptable, as it was conceived for low capacity.

2.1.3 Frameworks and Protocols for Ultrasonic Communication

2.1.3.1 Open Protocols: The SoniTalk SDK

Zeppelzauer and Ringot (2019) created an ultrasonic protocol, which focuses on data-over-sound. Information sent with the protocol is sent as an individual message composed of several blocks. For detecting a message, it starts with a "start block" and an "end block", whereas the starting block has half the used frequencies starting with the base frequency upwards. In comparison, the end block consists of the other half of frequencies, the upper ones. Further, every message is encoded in binary and is distributed through the different frequencies. This results in a barcode alike ultrasonic signal. In addition, error detection and error correction code are highly recommended and furthermore, an encryption algorithm would be needed to be implemented as well.

2.1.3.2 Proprietary Frameworks and Protocols

Lisnr is a technology for near-ultrasonic data transmission. It is optimized for different payloads and acts in the audible and inaudible range. In detail, it is designed for the frequency band of 12kHz to 19kHz. Further, robustness, fine-grained adjustable range, multi-channels, a local tone generation and security features like cryptography, key exchange or data packetization are the featured main functionalities of Lisnr. Besides that, they state that Lisnr has a transmission rate of up to 1000/bits-per-second per channel (Lisnr, 2019). Examples, where Lisnr is already in use, are for example the official application American football team for delivering live content in the stadium and the app of an American music festival called "Made in America Festival", where they sent more information to the festival guests (Mavroudis et al., 2017).

Shopkick implemented a framework for providing bonus features on visiting stores and purchasing products via ultrasound. They send out a unique identifier about the exact position in the store the user is visiting and send this to the Shopkick servers through the constantly listening mobile app. The company of the store verifies the visit and the user receives reward points, which can be used for discounts on products. (Mavroudis, on the privacy) Further, Shopkick uses a full frequency spectrum on analyzing the incoming signal (Arp et al., 2017).

Silverpush uses five different symbols divided over five frequencies in the range of 18kHz to 20kHz. As the modulation scheme, M-FSK is implemented to divide the five characters or letters into five different frequencies. Furthermore, as a kind of error correction, every letter can only appear once in a message and the letter 'A' has to appear in each signal. On the receiving side, the Goertzel algorithm is used to retrieve the five frequencies in a message (Arp et al., 2017).

Google Nearby implemented discovering devices through various ad hoc radio technologies. A random token gets broadcasted with a length of three seconds. Receiving devices

can decode the sent token. For reliability reasons, the message is sent repeatedly with a pause of one second in between. All further communication and data exchange, after finding the correct token, happens through the Google Cloud (Legendre, 2015).

Sonarax provides an ultrasonic protocol which should work from 20 cm to 35 m for a machine to machine (M2M) communication. Further, it has implemented encryption methods and supports well-working communication on moving devices. Besides that, indoor navigation is implemented and advanced analytics for collecting data on the communication events is available. It is only on the software side and no special hardware is needed (Sonarax, 2019).

Copsonic features a technology for ultrasonic communication between two devices. It is based on just microphones and speakers and also uses security as one of their features. The operating frequency band is between 3kHz and 40kHz and provides a detection speed of less than 300ms. Moreover, Copsonic offers data transmission speed up to 15 kbp/s and gives the chance of full-duplex transmission without self-interference (CopSonic, 2019).

NearBytes presents contactless communication technology, which is compatible with all computers and mobile phones. They focus on providing an easily, quickly and securely way of communication with microphones and loudspeakers. Further, they are giving the user a security layer and integrated authentication. Besides that, NearBytes can communicate between different operating systems and does not need additional hardware besides speakers and microphones (NearBytes, 2019).

2.1.4 Ultrasonic Applications

2.1.4.1 Context-Awareness

Bisio et al. (2018) created a method for perceiving the environment via ultrasounds. Based on actively sending out ultrasonic signals and listening continuously at the same time for echoes of the emitted ultrasonic signal. The architecture consists of a signal generator, a pre-processor, feature extraction, and classification. Therefore, a smartphone was used for implementing this system. The used method builds upon periodically sending ultrasonic pings and listening to echoes of those. The emitted signals are sinusoidal waves with the carrier frequency of 20 kHz. A bandwidth-filter with the bandwidth of 2 kHz filters then the echoes from the environmental noise and further buffers them to average them together. Next, nine features get extracted and then classified by an algorithm.

Bisio et al. (2018) proposed two scenarios, which were indoor-outdoor detection and state detection on wearing earphones. Testing the indoor-outdoor detection returned several results, but is still lacking on mobile devices. More than 17600 pings of a smartphone were recorded and divided into the two classes: indoor and outdoor. It was discovered that there is an accuracy versus latency trade-off as, the more echoes were averaged together, the longer it takes to retrieve a response. Several classifiers were used and all of them achieved good results. Besides ultrasound, the light sensor, the attenuation of the signal

and the magnetic field were compared and delivered worse outcomes than the ultrasonic method. Another impact, which was found out, was the role of clothes. Normally, the smartphone is in the pocket of the trousers, which lowers the accuracy in comparison to the laboratory settings. Besides that, the Doppler effect could change the frequency of the echoes. Further, the room size could attenuate the signal and lead to weaker echoes. The last impact, they found, was crowdedness, which could produce false echoes on reflecting the signals on other people (Bisio et al., 2018).

The second scenario, proposed by Bisio et al. (2018) was earphone wearing state detection. While smartphones can detect whether an earphone is connected via the audio jack or not, it cannot recognize, if those are worn by the user. Thus, ultrasound can help in detecting that state. An ultrasonic signal can be sent out via the earphones and echoes and attenuation can be analysed. As this acts on the ultrasonic frequency range, the user cannot hear it and it will not effect the quality of the played song. Three scenarios were found and further analysed. Either both earphones are worn, none of them are worn, or only one of the two is worn. The last scenario was separated and substance for another scenario. If the user wears only one earphone, it could be recognized, if it is the left or right one, with using the stereo sound output. Results showed, that the three scenarios had an accuracy of 98%. Only the worn earphone, if only one is worn, was not detectable. Though, this was solved by a separate scenario on exploiting the stereo output (Bisio et al., 2018).

2.1.4.2 Indoor Localization

Ultrasound can be used for localization in closed rooms. Lazik and Rowe (2012) created a system for indoor localization via ultrasonic chirps. Their system consists of many transmitters and a receiver, which can by any mobile device being able to record between 19 and 24 kHz. The transmitters are synchronized and continuously emit a unique identification code. This happens simultaneously for the whole transmitter infrastructure. As they do not synchronize the receiver with the transmitting devices, they use a time difference of arrivals (TDOA) pseudo ranging technique. The signals transmitted can be received with a difference in time and therefore, every receiver can calculate the relative timing itself. In order to have a stable transmission, they decided on using pulse compression on linear chirp signals. This modulation method increases the range resolution and the receiver sensitivity by effectively increasing the signal-noise ratio (SNR). Chirp wave-forms sent with pulse compression linearly increase in frequency and result in a compressed signal. Chirps can be detected much simpler by a correlation with the original chirp. To achieve multiple access transmissions, chirp-rates are used, where every chirp consists of a sequence of two Hamming codes. On the receiving side, matched filtering is applied, which is convolving the incoming signal with a time-reversed version of the expected incoming signal nature. The experiments were done by using an audio DAC/ADC, a higher frequency sensitive

microphone, an iPhone 3G and piezoelectric tweeters. Timing accuracy and location accuracy were evaluated and the result was that 95% of the test points were localized correctly within a 10cm accuracy.

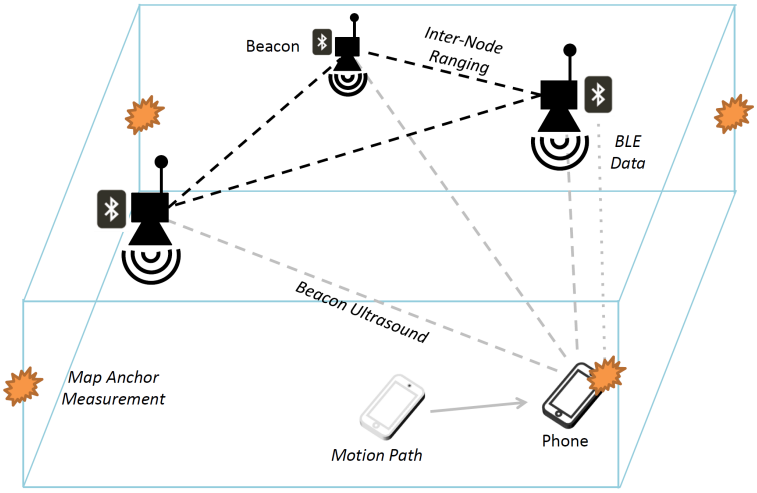

Figure 2.1. System overview of the Acoustic Location Processing System (ALPS), (Lazik et al., 2015).

Lazik et al. (2015) presented a system called Acoustic Location Processing System (ALPS), which augments Bluetooth Low Energy (BLE) transmitters with ultrasound. ALPS consists of three or more transmitters, which are relatively flexible in terms of positioning.This overview can be seen in figure 2.1. Those transmitters are time-synchronised and are compound of an ultrasound transceiver board, a BLE board, a piezo bullet tweeter, and a battery pack. Especially during hardware research, they worked on finding a sensitive loudspeaker, which is capable of transmitting frequencies between 18 and 24kHz. Two modulation methods were used in ALPS, on the one hand, ultrasonic chirps and on the other using the ultrasonic carrier for ranging information. They were using time-division multiplexing TDMA for the transmission. Between the beacons themselves, a Micro-Electro-Mechanical System (MEMS) microphone on the level of transceiver board was implemented, to achieve inter-beacon ranging from two beacons at the same time, by triggering them. One beacon is sending and the other listening. For evaluation of the system, user-assisted mapping was applied and an algorithm was used, based on the inter-node ranging, estimation of z plane and user-specified x - y plane. Lazik et al, could achieve an average error of 19.8cm and 16.1cm in the experiments. Overall, the user's location can be tracked within less than 100cm.

Murata et al. (2014) proposed another concept of indoor positioning, where the user is the transmitter via their smartphone. The system consists of several microphone sensors and microcontrollers positioned inside the place and the emitting device, which is fulfilled by the user's smartphone. The transmitter and one receiver are not synchronised with each other, therefore only a relative delay time can be captured. At least four microphones are

needed to capture three delay times, with the synchronized microphones. Thereby, the x, y and z coordinates can be calculated. Feasibility tests were done by using the frequency 17kHz as the main transmission one and sending out small 400µs bursts in intervals of 35ms. For measuring, a simple fast Fourier transformation (FFT) was applied. Static tests as well as dynamic tests with movable objects were conducted and showed that the system is accurate enough for both conditions.

Thiel et al. (2012) described an algorithm for a proximity detection method based on Bluetooth and ultrasonic beacons. As a frequency range, the ultrasonic band of 18kHz to 23kHz was chosen, as it is mostly inaudible to humans. Next, a kind of on-off keying was used on a continuously sent sine wave on the carrier frequency of 18kHz, which is based on amplitude modulation. A rather short signal length was used on getting a fast recognition with a fast Fourier transformation (FFT). Furthermore, the magnitude of the received signal is needed, as the interest was only in the amplitude. The detection is based on two sliding auto-correlating windows with different sizes. A simple threshold is then the indicator of the detection. Further, experiments were done on standing and walking scenarios and showed a big discrepancy between those two. While the standing scenarios work well, the walking ones could still be improved.

Carotenuto et al. (2018) realized a method for ranging radio frequency identification (RFID) tags through ultrasound. Several RFID chips get placed into a room, whereas they get an ultrasonic chip. The beacon sends out a signal via a RF radio transmitter, which activates the listening of the ultrasound chip on the RFID tags. The ultrasound chirps get received by the RFID tags with delay, which is saved on the tags. Further, the beacon consists of a microcontroller, a DAC, a RF transmitter, an acoustic emitter, and a power amplifier. The RFID-based sensors have a microphone for receiving ultrasonic chirp signals. Moreover, a central processing unit with a RFID reader is needed to gather all-time data of the RFID tags. Though the RF synchronisation delay can be figured out and calibrated, a jitter is still available. Furthermore, a frequency band of 15kHz up to 40kHz is chosen and a single chirp message consists of 1024 samples at a 192 kSamples/s sample rate. The length of one signal is 5.33 ms and gets smoothed by a Hamming window. Experiments on the processing chain, accuracy and reliability resulted in a ranging accuracy of 1.2 mm within a range of 2.3 m.

2.1.4.3 Ultrasonic Communication

Arentz and Bandara (2011) implemented a short-range directional near ultrasound communication channel for iOS devices and other smartphones. Based on their research, they created a system architecture using frequencies between 20kHz and 23kHz. Tested devices were able to produce and receive frequencies up to 23kHz, because of the maximum sampling rate of 48kHz, which many of their devices were able to. The encoding is using different duration for the transmission pulses. Those pulses are 16-bit sample values fre-

quency based on a sine wave function. On receiving this ultrasonic message, a bandpass filter is applied as well as normalization and a low pass filter as pulse edge detection. After using a binary decoder, a string of bytes from those pulses is the result. Their transmission protocol is working on trains, in busses and outside without a loss in quality, but is restricted to about 80 cm and a rather small tilt angle until the reception of signals gets impossible.

Getreuer et al. (2018) presented a protocol for ultrasonic communication using the frequency band of 18.5kHz to 20kHz. The main target platform was mobile devices. The quick and localized exchange of sound delivers the perfect base for interactive mobile applications. Direct-sequence spread-spectrum modulation (DSSS) is in use as of its robustness against noise. The data will be modulated so that every frame is a period of the DSSS code and has one symbol of data encoded. Further, this modulation method should avoid the Doppler effect and weak signal strength, which can both impede communication between mobile devices. After experiments, the results in real-world environments showed that the noisier surroundings are, the more unreliable the system gets. Still, the protocol is robust enough against background noise and motion up to two-meter distance indoors.

Ka et al. (2016) proposed a method of near-ultrasound communication for 2nd screen services. On achieving a transmission data rate of 15bps and using a low volume on a non-line-of-sight scenario over a few meters, chirp quaternary orthogonal shift keying (QOK), novel synchronization and carrier sensing algorithms were needed. Target applications collected were, on the one hand, the automatic data reception during TV shows and, on the other, commercial tracking purposes of companies. Only receiving instead of constantly pulling as in previous 2nd screen implementations, makes the method less energy-consuming.

PriWhisper, a keyless secure acoustic short-range communication system was implemented by Zhang et al. (2014). The goal was to build another channel that resembles near-field communication (NFC). It is based on aerial acoustic signals and can perform non-line-of-sight communication. The target platform was smartphones, which had the advantage over NFC that every smartphone has a performant speaker and microphone for performing PriWhisper. Similar to NFC, PriWhisper works on kind of touch, but it allows bigger distances than NFC. Though for security reasons, the space between two devices should be rather small. PriWhisper consists of, on one side, a channel encoder, a modulator and on the other, a channel decoder, a demodulator, and a speaker. To avoid transmission errors, PriWhisper is using a cyclic redundancy check (CRC-8). Furthermore, the modulation of the signal is frequency-shift keying (FSK) which uses one multi-bit symbol per frequency on a carrier frequency of 9kHz. Further, PriWhisper uses adaptive signal strength selection by recording the ambient noise level 0.1s after the MFSK modulation happens. On the jamming side, a random white Gaussian noise signal around the carrier frequency is computed, which will be filtered out again on the receiver device by taking its own jamming signal and subtracting this from the received signal. In tests, PriWhisper resulted in a small package error rate of 1.5% in noisy indoor and outdoor situations.

Lin et al. (2015) created a near-field communication channel over inaudible sound called A-NFC. The advantages over other technologies would be that it is easy to port, as it only happens on the software side. Further, A-NFC can perform two-way communication only over the inaudible acoustic channel under several factors like channel efficiency, system latency, and channel reliability. The chosen frequency band of 15.8kHz to 20.6kHz should provide an inaudible channel for data transmission. In addition, real-time communication is difficult to achieve, as A-NFC is dependent on the OS and the hardware of the device like RAM. Thus, it requires a tolerance for non-real-time communication. Besides that, the varying hardware of the devices needs to be included. A-NFC further uses quadrature phase-shift keying (QPSK) and automatic gain control (AGC) to adjust the amplitude to a suitable level on different chipsets. Next, Testing different center frequencies resulted in the frequency 18.525kHz on a symbol length of 1.81ms.

2.1.4.4 Internet of Things and Wearables

Microcontroller-based systems are called embedded systems. They receive inputs via different simple interfaces like cameras, buttons, switches or touchpads. Those complete systems connected to the Internet result in a basic Internet of Things (IoT) system. Things can be cars, tools or devices of any kind which have a connection to the Internet and are able to communicate with each other. An IoT system normally consists of three components: a device that can handle the Internet connection, the connection to the Internet itself, for example via Wi-Fi or Ethernet, and backend server, which handles the storing, processing and sharing of the information. Alzahrani (2017) set up a simple and general framework and created two interfaces for it. One implementation was based on Arduino and another one on Raspberry Pi. Both were upgraded with Wi-Fi shields or adapters via USB, whereas the Arduino is easy to attach and lightweight option and the Raspberry Pi with a faster processor and a larger memory. The framework has several input sensors such as GPS, smoke sensors or humidity and temperature sensors, and on the other hand, output sensors like LEDs and buzzers. Further, a simple web server with a connected Android application is needed and between the sensors and the backend, the previously described interface has its place. Several sensor scenarios were established and tested like GPS acquisition, weather sensing, liquid levels, smoke detection, soil sensing, and distance measuring.

Santagati and Melodia (2017) presented a framework for ultrasonic communication on wearable devices called U-Wear. Currently used wearable medical devices are based on radio frequency (RF). U-Wear is completely implemented on the software side, as it just needs speakers and microphones for sending signals in the frequency band of 17kHz to 22kHz. Ultrasonic signals do not go through walls and cannot be that easily distorted by jammers, only if they are near the victim. Further, there is no conflict with the existing RF technologies and this spectrum is unregulated, which gives more freedom to adjusting

the signals. In general, it is more adaptable in terms of features and functionality than other RF technologies. U-Wear uses Gaussian minimum-shift keying (GMSK), which is a special form of FSK and produces a clicking-free transmission because of its phase-continuity. The framework is made for either the master/slave-concept between devices or peer-to-peer communication. Experiments showed a data rate of up to 2.76kbit/s with a bit-error-rate of 10-5 and therefore a reliable framework to exchange data between medical devices.

2.1.5 Security Aspects of Ultrasonic Communication

Deshotels (2014) categorizes the abuse of sound-based channels in two different segments. Data exfiltration can be either performed in an intra-device way or in an inter-device way. Both ways are based on inaudible sound. The intra-device communication happens between applications on the device, which would otherwise not be able to communicate with each other. For example, application A handles secret data and has no other privileges. Application B has network access and several other permissions. Application A can send data over inaudible sound and Application B can receive this message via the microphone on the same device. Second, the inter-device method happens either by a trojan horse, which sends data via sound or via any application, where the company wants to abuse the possibility of ultrasound transmission. Another device or even networks of devices can then receive the unwillingly sent audio data.

Deshotels (2014) set up two proofs of concept to show, how well sound-based covert channels work. They completed two different experiments, one on isolated sound, which was rather roughly built, and one on ultrasonic sound, which should show the limits of attacks. The isolated experiment was performed by using a Samsung Galaxy S4 as transmitter and a Google Nexus 7 (2003 Edition) as a receiver. The transmission was achieved via vibrations of the accelerometer and received by an accelerometer monitor application. They proofed successful communication between these two devices by sending a pattern of vibration and pause. Though this experiment used two independent devices, they wrote that this method would work on one device.

For the ultrasonic approach, Deshotels (2014) used a transmitter based on frequency shift keying (FSK) to send digital data. The FSK was set up on two frequencies for 0's and 1'S, which were 18kHz and 19kHz. Their ultrasound file consisted then of three parts: 128 symbols of the same frequency as an identifier, a 128 symbols large predefined pattern of high and low frequencies to accurately locate the start of the data and the digital message converted from an ASCII string to binary data. The receiver, which was an Android application, always knows the first two parts of a received message and decodes and displays the third part converted back to a string again. In addition, optimization methods were added by decoding the message 10 times with a slightly moved starting point. Whereas, a reduction of the amplitude at the beginning and at the end of a symbol improved the steadiness

of the decoding against audible clicks. Based on that data transmission, they tested on getting an effective bitrate, which resulted in a maximum bitrate of 345 bits per second, and on finding the maximum distance for transmission, which was 100 feet with a bitrate of 8.61 bits per second before the error rate increases significantly.

Hanspach and Goetz (2013) described several acoustical mesh network applications exploiting security measurements. A keylogger could be implemented through covert acoustical networks. One communication partner gets infected and sends every keystroke via acoustic signals to other devices in the network. Furthermore, tunneling over the Internet is a possible threat. The attacker could gather all data from a covert acoustic network and sends the information to a SMTP server. Two-factor authentication based on acoustic networks could be broken by extracting the authentication feedback and using it to get access to whatever the other devices asked for. In addition, every small text file, limited through the bit rate, could be periodically sent out to the covert acoustic network. One solution found out, was audio filtering, if the input and output were not able to shut off. A simple bandpass for the concerning frequencies would be a first step.

Thinking one step further, Zeppelzauer et al. (2018) developed a technology-agnostic and robust to noise firewall for ultrasonic signals, called SoniControl. It is based on a robust background model that is continuously updated. The input is high-pass filtered for removing everything hearable and gets computed through a short-time FFT. Further, a cyclic buffer is started, where all input frames are stored and processed as soon as the buffer is full. Next, the Kullback-Leibler (KL) divergence is computed between the buffer and the background model. With the help of a simple threshold, a detection can be identified. After that, an alert pops up and lets the user decide if they want to block the signal or not. Blocking happens either by jamming the incoming signal with white noise in the inaudible frequency range or by using the exclusive access to the microphone of the smartphone. No other applications can use the input as long as the firewall obtains it. The firewall has been implemented on Android starting with Android 4.1 and above.

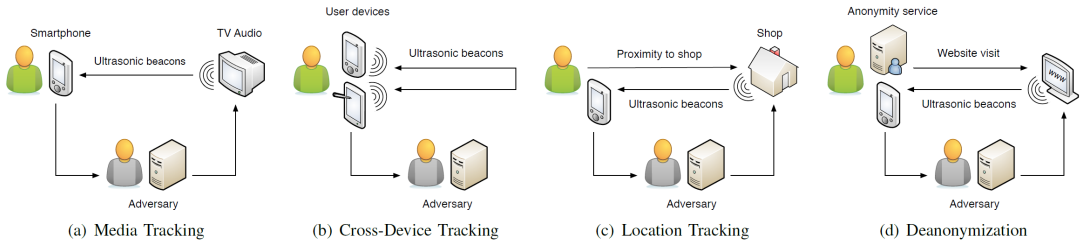

Figure 2.2. Four privacy threats which are generated by the ultrasonic channel: Media Tracking, Cross-Device Tracking, Location Tracking and Deanonymization (Arp et al., 2017).

Arp et al. (2017) gathered several privacy threats via the ultrasonic side channel on end-users mobile devices, as seen in figure 2.2. An application, which is listening to ultrasonic

signals via the microphone, is just needed to raise privacy issues. The four threats are working like follows:

- Media tracking was identified at the beginning. During commercials in Web, TV or Radio, ultrasonic messages can be hidden and those messages can carry information like timestamp or an identifier of the watched commercial. On the receiver, the malicious application can create a media profile based on the watched commercials, though they are sent out anonymously. This method results in a detailed user profile on what the user watched, where he watched it and when he watched it.
- Cross-Device-Tracking gets enabled through the ultrasonic channel. Devices of one user, which are not connected yet, can be assigned together if the same signal gets received several times. This opens the possibility to create a more defined user profile on gathering data over several devices. Even private and business devices can be logically connected, though they are normally used separated from each other.
- The possibility to track the location of the user exists through ultrasound. A simple location identifier can be sent out and reveals the when and where the user was. This method bypasses the use of GPS and tracks every indoor movement the user performs.
- On services, which feature anonymization, can reveal the real identity of a user, by sending out an ultrasonic message while executing the service, like payment with bitcoin. The de-anonymization happens then through the listening mobile device.

Zhang et al. (2017) presented the vulnerability of speech recognition systems on ultrasonic attacks and designed an inaudible attack called DolphinAttack. Technologies like Amazon Alexa, Siri or Google Now convert identify speech and convert it to machine-readable actions. Such systems are based on three subsystems: voice capture, speech recognition, and command execution. The threat model of DolphinAttack needs to meet several conditions. During the attack, there is no access to the target device, which makes altering the system impossible. The owner is not interacting, while the attack is performed and further, the commands will be inaudible. Furthermore, malicious actions like spying, injecting fake information, denial of service, concealing attacks and the visiting of malicious websites can be achieved through DolphinAttack. The attack exploits the nonlinearity of the electric components which create new frequencies. Having that in mind, the attack starts with either text-to-speech bases commands or commands through a concatenative synthesis of pre-recorded voice samples. The commands are then modulated by amplitude modulation. Further, the tests were performed with a powerful transmitter with a sampling range of 300MHz and a portable transmitter, which was represented by a smartphone with a sampling rate of 48kHz. The results of the feasibility experiment of the attack were successful.

Mavroudis et al. (2017) described considerations about privacy and security regarding ultrasonic ecosystems. Applications, which use ultrasound-based actions, need full access

to the microphone. This gives the application not only access to the ultrasonic frequencies, but also audible frequencies could be further processed. This circumstance violates the least privilege principle. Furthermore, developers who want to use ultrasound are rated as potentially malicious. Especially, bad information and no-opt out of companies using ultrasonic communication are strengthening this opinion. Mavroudis et al. (2017) pointed out several vulnerabilities of ultrasonic ecosystems and their active participants. Users of anonymity networks like Tor could be deanonymized through hidden inaudible messages. Those signals are sent from the opened service and get captured by the device of the user. Through this connection, user data, which reveals the identity of the user, could be forwarded to the attacker. Further, the lack of authentication could be abused in terms of spamming the users devices with signals, which should forward them to the service of the attacker and influence the content they receive. Last, information from the device of the user could be requested through ultrasonic signals. The attacked device responds with personal data saved on the phone.

2.2 Related Communication Technologies

Beside ultrasound, alternative technologies like Bluetooth or NFC are already in use for communicating between different devices. The following section shows the features of those alternative technologies more in detail, lists additionally complementary technologies, which can be combined with ultrasonic communication, and gives information on the relation to ultrasonic communication.

2.2.1 BLE - Bluetooth Low Energy

Bluetooth low energy (BLE) needs only a part of the energy that Classic Bluetooth is consuming. Therefore devices using BLE can be powered by small, coin-cell batteries and makes it possible to use in areas like sports, fitness, health care or human interfaces like mice or keyboards. BLE can reach distances up to 200 feet and beyond, which makes it capable of devices in-home. Bluetooth low energy features, for example, a low cost, multi-vendor interoperability, enhanced range, different power consumption modes and the ability to run for years powered from just a coin-cell battery. BLE can either be used together with Classic Bluetooth or as a standalone sensor (Bluetooth SIG, 2013).

Bluetooth low energy supports data packet sizes of 8 octets as a minimum of up to 27 octets as the maximum. The data rate is 1 Mbps. Further frequency hopping can be used in BLE to minimize the interference of other technologies in the used Band of 2.4 GHz. In addition, BLE chips have high intelligence in the controller, which wakes up the host only if it is really needed. Therefore more energy can be saved and only the controller needs power. BLE performs connection setup and data transfer with 3ms and creates only a low latency. Furthermore, a high range over 100 meters can be achieved and still be

robust as a strong 24 bit CRC is used on all packages sent. Besides, the data packets get encrypted by Full AES-128 and every packet has a 32-bit access address, which allows billions of devices (Bluetooth SIG, 2013).

In comparison to ultrasound, Bluetooth goes walls, whereas ultrasonic communication can be limited to rooms. Further, ultrasound can be configured for small distances of several cm or for bigger distances with several meters, which is more imprecise on the Bluetooth side.

2.2.2 Wi-Fi

Wi-Fi operates within the electromagnetic spectrum and belongs to the radio frequency (RF) technologies. The wireless network gets spread by access points (AP), which broadcast the wireless signal, for example, an Internet connection. On connecting with the AP, a network adapter is needed. Wi-Fi is described in several wireless technology standards (Webopedia, 2008).

The first wireless network standard was called 802.11 by the Institute of Electrical and Electronics Engineers in 1997. (Spectrum, 2016) The first standard only supported 2 Mbps, which was too slow for most of the applications. Since then it got iterated several times. Next, the first iteration 802.11b was already capable of 11 Mbps and was called Wi-Fi 1. Further, the next version 802.11a supported a bandwidth of 54 Mbps and was using 5GHz as the main frequency instead of 2.4GHz. The first version, which supported the dual-band wireless connection was 802.11ac. Both 2.4Ghz and 5GHz were in use and a bandwidth of 450 Mbps and 1300 Mbps was possible (Mitchell, 2019).

Like other forms of radio communication, wireless networks have to cope with interferences. Those interferences can arise on the usage of different building materials in-house, like concrete, drywall, glass, wood or metal, or other devices, which operate at the same frequency band as access points. Wi-Fi supports encryption by having several schemes implemented. The most known ones are Wired Equivalent Privacy (WEP), Wi-Fi Protected Access (WPA), and Wi-Fi Protected Access 2 (WPA2). WEP is already old and not really secure anymore. Therefore WPA or WPA2 should be used, in combination with longer and more complex passwords (Webopedia, 2008).

Wi-Fi uses a much higher operating band than ultrasonic applications. Further, the signal is, like with Bluetooth, sent through walls and cannot be limited to a room like with ultrasound.

2.2.3 NFC - Near Field Communication

NFC, short for near field communication, represents a short-range, low-power communication protocol between two devices. A radio-wave field gets set up by one device, which is detected by a second one. Small amounts of data can be transmitted through this channel.

Though it has a rather slow data transfer rate of 0.424Mbps compared to other technologies, it only consumes 15mA of power, has better security possibilities and does not need a "pairing" process like Bluetooth (Nosowitz, 2011).

NFC consists of three basic principles "sharing, pairing and transaction". The third one, transaction, is probably the most frequently mentioned when talking about near field communication. A simple device with a NFC chip, like a smartphone, can be quickly set up as a credit card or debit card. It can be used as one just with a tap on the transaction console. Further, things like library cards, hotel room key cards, public transit passes or office building passcards can be replaced by NFC. Though security concerns are still there, near field communication is easy to configure and objects like an ID or keys could be sooner or later replaced by the phones NFC. Sharing data is a bit more difficult than the other two principles. Related to the low data rate, videos or music cannot be shared via NFC. Near field communication acts here more like QR code, which can be scanned and is leading to the shared resource. More or less it is made for small portions of data like exchanging contact information (Nosowitz, 2011).

NFC needs a special chip for communicating with other devices, whereas ultrasound just needs a microphone and a loudspeaker. Further, in comparison to ultrasonic communication, NFC is constructed for only small distances of several millimetres and centimetres. Ultrasound can achieve both communication on small distances, as well as on larger distances of several metres.

2.2.4 LoRaWAN

Low-Power, Wide-Area Network (LoRaWAN) operates as the physical layer to set up a long-range communication link. Long Range (LoRa) uses the chirp spread spectrum and still resembles FSK modulation, though it increases the communication range. LoRa represents the first low-cost version of that communication. A simple gateway enables LoRa communication over hundreds of square kilometres. LoRaWAN especially fits for Internet of things (IoT) application, which needs a long lifetime without constant maintenance and a small amount of data transmission of sensors over long distances in a non-critical time span. Further, network nodes are mostly connected in mesh networks and forward packets from the end-node to the cloud-based network server. The network server then inter alia manages the network, filters redundant packets and performs security checks. End-nodes work asynchronous and send data as soon as it is ready. Therefore, a LoRaWAN gateway has to cope with a very high capacity of nodes. Furthermore, LoRaWAN is using AES encryption for security (Alliance, 2015).

LoRaWAN can be seen as a complementary technology, which is an add for ultrasonic communication. Whereas, ultrasound operates on small distances, LoRaWAN can send small data packages over long distances with low energy consumption. This makes the system independent from the Internet. In combination both play well together for specific

kinds of use cases. Whereas, ultrasound would handle the short distance communication and LoRaWAN the sending of data over longer distances.

2.2.5 5G

5G will enable the delivery of data within delays of less than milliseconds and bring download speeds of 20 gigabits per second, which is a big step from 4G, where speeds up to 1 Gb/s can be achieved. Five technologies should help in making 5G successful. Millimetre waves could solve the problem of more and more devices used at the same time. This could enable using frequencies between 30 and 300 gigahertz. I comparison to the length of radio waves, which measures tens of centimetres, millimetre waves are only between 1 and 10 mm long. They were already tested with stationary points. A major problem occurs in sending those signals through buildings. Those waves will get absorbed and lead to the next technology called small cells. Small cells prevent from dropping signals throughout cities. Those portable miniature base stations can be set up in short distances of about 250 metres and work as an extension to the user. The smaller size of the new cells makes an advantage over traditional cell antennas, as more small antennas can be set up on houses. Though frequencies can be reused in different areas, rural landslides make it harder to set up a good network of such antennas. To overcome this problem, 5G base stations can have more than one antenna. This is called massive multiple-input multiple-output (MIMO) (Spectrum, 2017).

While old 4G base stations only can handle eight transmitter and four receiver antennas, 5G stations hold more than a hundred of those. This raises the capacity by a factor of 22 or greater. MIMO is the acronym for multiple-input multiple-output and stands for wireless systems with many more transmitters and receivers on a single array. Further, MIMO achieves new records in spectrum efficiency but also brings interferences, where another technology called Beamforming can help. Beamforming calculates the most efficient way to deliver user data. The best transmission route will be identified by signal-processing algorithms, sending data packets into many different directions. This happens under a coordinated pattern including arrival time and packet movements. Furthermore, another problem can be solved by Beamforming, as objects, which block the millimetre waves can be avoided. Last, full-duplex helps in achieving low latency and high throughput on 5G. It enables the possibility of sending and receiving a signal on the same frequency at the same time, by using silicon transistors, which act like high-speed switches. All these technologies should provide a base for ultralow latency and record-breaking data speeds opening new opportunities for smartphone users, VR gamers or autonomous cars (Spectrum, 2017).

5G can be seen as a complementary technology to ultrasound. For projects with a higher throughput needed and time-critical transmissions 5G fits as an addition to ultrasonic communication.

2.3 Alternative Communication Protocols to the Thesis

Besides ultrasound, different non-acoustic communication protocols are already in use in the area of IoT. UNITA uses WebSockets and REST, which are further described in the chapter 3.4.3. The following protocols would be alternatives to those used protocols in UNITA.

2.3.1 MQTT

MQ Telemetry Transport (MQTT) is designed as a lightweight broker-based message protocol. It is using publish and subscribe and is simple, open and easy to implement. MQTT is favoured in use cases where the network is expensive or has a low bandwidth and when it should run on an embedded device due to limited computing resources. Further, MQTT features a provision of TCP/IP network connectivity, an agnostic view on payload content, a small transport overhead and notifications on the disconnection of a client. On message delivery, there are three states of how messages arrive. Either message delivery with best efforts, but with possible message loss, covered message, where duplicates could occur, or message receipt exactly once, if loss or duplicates distort the result. MQTT has a fixed header size for different message types. In addition, a variable header size for specific command messages is provided by the protocol (Eurotech, 2010).

2.3.2 CoAP

Constrained Application Protocol (CoAP) represents a specialized web transfer protocol. A goal was to create a generic web protocol. It is constructed especially for constrained nodes and constrained networks. CoAP is based on a request and response interaction model between application endpoints. Further, M2M requirements are fulfilled, UDP is used supporting unicast and multicast and asynchronous message exchange is applied. In addition, CoAP owns a low header overhead and parsing complexity, provides URI and Content-type support and implements simple proxy and caching capabilities. Besides that, it uses a stateless HTTP mapping. Similar to the client/server model of HTTP, CoAP focuses on machine-to-machine interaction where both nodes represent the client and server. Furthermore, it works on asynchronous interchanges over a datagram-oriented transport method. Four types of messages are defined, Confirmable, Non-confirmable, Acknowledgement and Reset (Shelby et al., 2014).

2.3.3 AMQP

Advanced Message Queuing Protocol (AMQP) is an interoperable enterprise-scale asynchronous messaging protocol. AMQP includes a defined set of messaging capabilities and represents a network wire-level binary protocol. The messaging model consists of

three main types of components: exchange, which receives messages and routes it to the message queue, message queue, which is storing the messages until it can be used, and binding, which represents the routing criteria between exchange and message queue. Further, this model is a classic message-oriented middleware and can be compared to an email server. AMQP can be described as multi-channel, secure, asynchronous, negotiated, portable, efficient and neutral. (Vinoski, 2006)

2.4 OSI-Model

UNITA is based on different layers, which are inspired or directly refer to the OSI-Model. The following sections give a basic understanding on the functionality of each OSI-Model-layer. The Open Systems Interconnection model (OSI model) sets a standard for telecommunication functions and computing systems. It is a conceptual model and consists of seven layers: Physical, Data Link, Network, Transport, Session and Presentation layer. The OSI model determines, what a layer needs to provide for the next layer (ITU, 1994b).

Layer 1 - Physical Layer: The physical layer is the bottom layer and provides mechanical and electrical tools for activating and deactivating physical connection as well as sustaining this connection and sending bits. The connection can be established via electrical signals, optical signals, electromagnetic waves or sound. Furthermore, the physical layer handles how bits get sent and in which kind of bit/symbol sequence they get transferred (ITU, 1996a).

Layer 2 - Data Link Layer: The data link layer guarantees the correct transmission of the information. It divides the information into data blocks (frames) and adds checksums. In addition, the data link layer handles the sending speed of the frames. The resending of the frames itself is not part of the layer (ITU, 1997).

Layer 3 - Network Layer: The network layer handles the correct shifting of connections and the onward transmission of data packets. Further, routing for packets with the best combination of nodes is one task and creating and maintaining cross-network routing tables comes along with that (ITU, 2002).

Layer 4 - Transport Layer: The transport layer splits the data into segments and provides congestion avoidance. Further, the layer implements multiplexing to handle multiple endpoints on one node via ports. The transport layer checks the previously added checksum and starts an automatic repeat request to retransmit wrong received or lost segments (ITU, 1996c).

Layer 5 - Session Layer: The session layer ensures an organized and synchronized data exchange between two systems. Checkpoints get configured to restart and resynchronize the transport process after a blackout. This session restoration feature prevents data transmission from full restarts. Further, it handles authentication and authorization (ITU, 1996b).

Layer 6 - Presentation Layer: The presentation layer works on the data compression and on the encryption of the information so that no other application layer can read the data. Encryption could also be handled in the application, session, transport or network layers, which brings different advantages and disadvantages. Further, it translates the data from a system-dependent format to an independent one (ITU, 1994c).

Layer 7 - Application Layer: The application layer provides all functions for the applications. On this layer, all input and output happen whereas the application itself is not part of the layer (ITU, 1994a).

3 UNITA - An Ultrasonic Beacon

UNITA is a SDK for developing applications using ultrasonic communication, as well as a hardware solution for sending and receiving ultrasound signals, called UNITA beacon. It consists of a development kit for implementing applications on the beacon and for client applications. Further, a server setup is included. This chapter is structured in two parts: the hardware implementation and the software implementation. The hardware part revolves about the creation of the beacon and the software part again structured in four subsections:

- Operating system
- UNITA SDK for the client side
- Used protocols
- UNITA Server

3.1 General Requirements of UNITA

Within UNITA, a software development kit and a hardware setup for creating own projects using ultrasonic communication, should be developed. On the one side, hardware for an ultrasound beacon needs to be gathered and assembled. On the other, a software base for those beacons and for client applications is needed. In addition, server software for data upload and download, and further processing is required.

3.2 Environment and Architecture of UNITA

The architecture of UNITA consists of three main components: ultrasonic beacon (UNITA beacon), server and mobile device, which can be seen in figure 3.1.

The UNITA beacon and the mobile device represent the local part of the environment and the server stands for the cloud side. Further, different communication technologies get used, whereas the most important one is the ultrasonic communication between the beacon and the mobile device. The two communication participants can exchange data locally via this acoustic channel without being connected to the internet. To connect with the internet via the server and to send data like messages or location, the UNITA beacon establishes a WebSocket connection with the server. The WebSocket protocol was chosen, as the connection needs to be bidirectional and constant as both server and beacon need to request data from the other component. In addition, the mobile device needs to retrieve

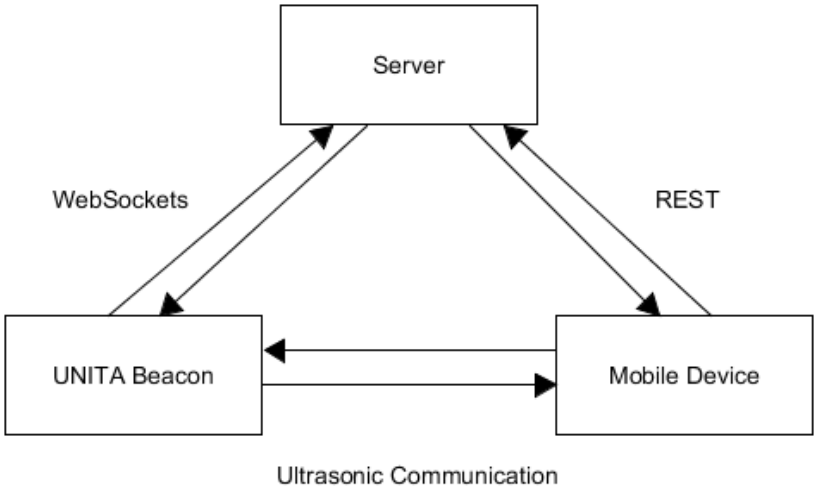

Figure 3.1. The UNITA Beacon, the server and the mobile device with their corresponding communication method.

information from the server, which happens via REST interface, as only the mobile device has to request. On the server-side, a database is attached to the server for data saving and retrieving purposes.

The UNITA beacon can be roughly structured into three layers: SoniTalk, the UNITA SDK and the application, as seen in figure3.2. SoniTalk, as the base, handles the generating, sending and receiving of ultrasonic signals. It further processes the outgoing and incoming audio data and hands it over to the SDK. The UNITA SDK provides the functionality for creating messages, including message data types and utils for processing them. Besides that, the development kit is interpreting the incoming audio data as ultrasonic messages and additionally includes controller for the receiving and sending process, location updates, local message storage options, . The application on top uses then the functionality to send and receive different messages, depending on the use case. This layer has an user interface and direct user interaction.

3.3 Hardware Implementation

3.3.1 Requirements

The Unita beacon should be set up with low-cost components, to build a cheap beacon for ultrasonic communication. Further, hardware components need to be easy to assemble and quick to change. The single-board computer or microcontroller, which handles the computing, should be strong enough to cope with digital signal processing methods like Fast Fourier Transform (FFT), but as energy-saving as possible to minimize the power consumption. On the software side, the computing element should be capable of running

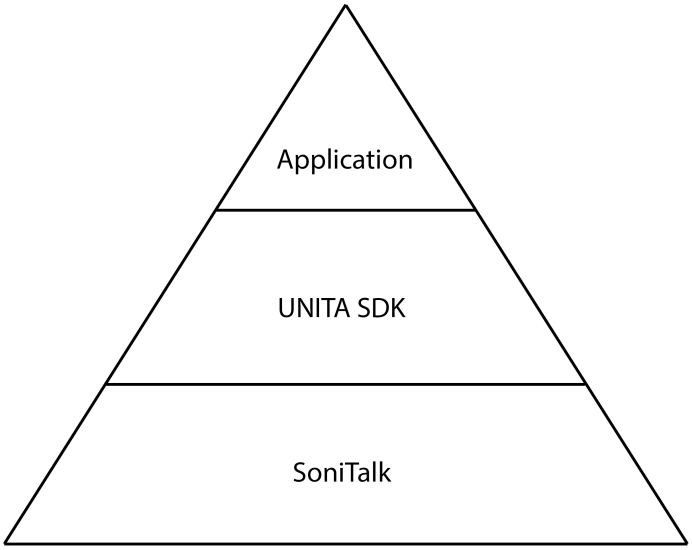

Figure 3.2. Overview of the three layers, which are used in UNITA: SoniTalk, UNITA SDK and an Application.

Android, as the SDK is needed to be written for Android. This limitation comes from the SoniTalk dependency, as this ultrasonic communication protocol implemented for Android in Java, is the first layer of the Unita SDK. Furthermore, Wi-Fi or equivalents like LoRaWAN are an optional add, for enabling communication with the server, and could be either on-board or equipable. Besides the processing board, a microphone and a loudspeaker were needed. Both should be capable of receiving or sending ultrasonic frequencies and should be rather cheap. Furthermore, depending on the board, an input for the microphone and an output for the speaker needs to be added. If it is not available yet, a sound processing chip or an external sound card could achieve that. In addition, a case for assembling all hardware components needs to be found or crafted.

3.3.2 Challenges

First, the trade-off between computing power and energy consumption was first a challenge to handle. Microcontrollers have lesser computing power but consume less energy. Whereas, single-board computers have much better and stronger processing units and can run full operating systems on them. The disadvantage, which comes with using such boards, is the higher energy consumption. That challenge was really important, as it needs to handle real-time FFT, which is the basis for the whole audio processing. Another challenge was the sensitivity of ultrasonic signals of the loudspeakers and microphones. In addition, the hurdle of signal strength appeared during the research phase. Further, finding the minimum needed supply voltage was a smaller challenge appearing in the process of testing hardware. Handling the input and output signal turned out, was not as easy as thought, as sound cards with a sample rate of minimum 44.1kHz and compatibility to the

microcontroller or single-board computer are rather difficult to find. Additionally, the need of running Android on it, has decreased the number of available boards strongly. Finally, a ported and running Android OS version for the chosen board has to be found.

3.3.3 Hardware Selection

A first research on microcontrollers and single-board computers was pursued. The goal was to get a list of components, which could do the processing. The searching started on simple search requests via search engines. Further, going through maker forums and digital sound processing forums should give a more detailed perspective on available boards and microcontrollers. Next, the specifications and features of several components were collected and compared with each other. Another aspect was the price and the availability of those computing devices. This was checked by using relevant shops on maker hardware. Besides research on the computing board, suitable microphones and loudspeakers were searched. This was happening in the same maker hardware relevant shops as for the microcontrollers. Several components with different construction types were found and bought. These types include piezo loudspeakers, miniature loudspeakers and piezo buzzers of different sizes. The first research part led to the selection of a Raspberry Pi 3 B, which was used for further research on specialized hardware parts.

The gathered list of microphones and loudspeakers consists of eight boards and microcontrollers. Those components were further rethought to determine the most suitable ones. This detailed evaluation is described in the chapter 5. Furthermore, the found microcontrollers and single-board computers are summarized in table 3.1.

See table 3.2 for the full list microphones and speakers:

Next, a USB sound card as an audio interface has to be found, for the Raspberry Pi 3 B, which was chosen for further tests. The Raspberry Pi 3 B has an audio output but lacks an audio jack for the input of sound. The first sound card tested, was a 7-in-1 USB soundcard, which was rather cheap and was found under several distributing companies with the same hardware, but under another name. The USB sound card was plugged to the Raspberry and audio jack elements were connect with a breadboard via simple wire pieces. One microphone and one loudspeaker were soldered on wire pieces and also connected to the breadboard. Next, a spectrogram application was installed and was used to test the microphone input. The same happened for the loudspeaker output via an installed music player app. Though the audio interface was recognized from the system and shown in the Android terminal, input, as well as output, were not working. As a backup solution, to exclude defective speakers and microphones, a commercial loudspeaker, as well as a headset microphone, were plugged. This combination still did not work.

Another sound card has to be found. The next try was via the GPIO pins. After searching in the official Raspberry Pi forum, a company creating audio shields was discovered. A specialized signal processing shield for the Raspberry Pi 3 B was then bought and con-

Board	Features	Price in Euro
Arduino Yún Rev 2	5V, own Linux microprocessor	∼60,00
Tinkerforge	Brick und Brickletbasis Java API Bindings	SPL 29,99 Master 29,99 WIFI 29,99 Power 19,99
BeagleBone Black	Audio über HDMI BeagleBoneAudio Cape (not available in onlineshop anymore) Power cape	 ∼60,00 power cape 30,00
Banana Pi M2+	Android	40,00 – 50,00
Pandaboard	3 to 6 Watt	∼290,00
Cubieboard 4	Android Lubuntu/Cubian maximum of 5 V, 2,5 A and 12,5 W one 3,7 V Li-ion battery needed	125,00
Raspberry Pi3+	Android	40,00
Teensy 3.6	Specialized on signal processing	30,00
Odroid	Android	∼70,00

Table 3.1. List of microcontrollers and single-board computers to choose from.

Component name (all from company Ekulit)	Component type
EMY-602N	condenser microphone
EMY-625N	condenser microphone
EMY-63M/P	condenser microphone
EMY-63M	condenser microphone
EMY-6027P/N-R-42(IP67)	condenser microphone
EMY-9765P	condenser microphone
MCE-101	condenser microphone
RMP-05	piezo loudspeaker
LSP-3015	piezo loudspeaker
BM 15B	piezo buzzer
LSF-50M/N	miniature loudspeaker
LSM-36M/B	miniature loudspeaker
LSM-S19K	miniature loudspeaker
LSF-40M/N/G	miniature loudspeaker
LSF-23M/N/G	miniature loudspeaker
LSM-57F	miniature loudspeaker

Table 3.2. List of microphones and loudspeakers for further testing.

nected. On testing it, it was not working nor it was found via the terminal, though the correct driver was part of the Raspberry Pi Lineage image. The search for a sound card went on

and resulted in two other USB interfaces from different companies. Whereas the first one was not working at all, the second one was a well-working solution. Furthermore, several microphones and loudspeakers were tried out and both, input and output side, showed, that without amplifying the signal, the signal was not strong enough. Additional research on small amplifiers was done and a set of amplifiers for a microphone and a loudspeaker was found. Next step, those amplifiers were interposed between the speaker or microphone and the respective output or input. On the output side, the amplifier worked well and increased signal strength. In comparison, the microphone amplifier was too strong and produced too much noise. This circumstance demands another amplifier, as the bad signal-noise-ratio (SNR) was not acceptable.

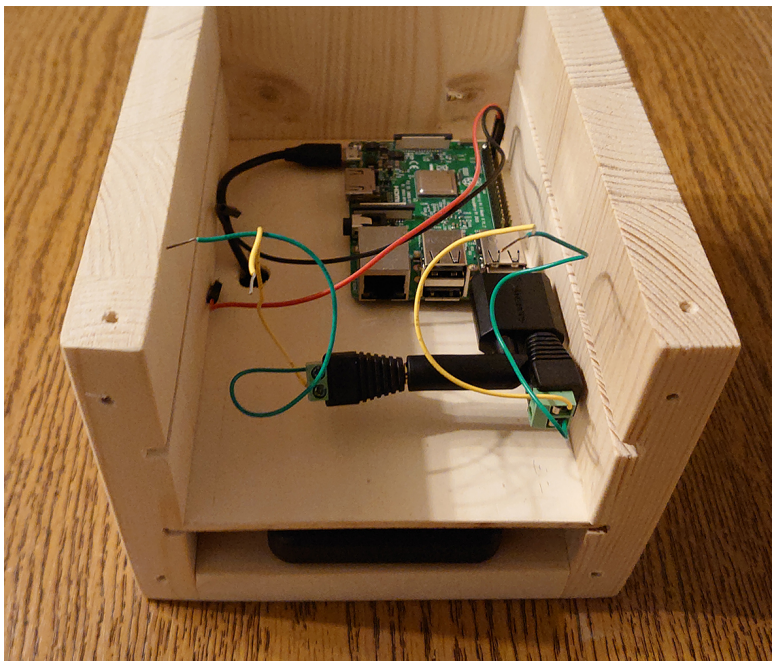

Figure 3.3. The beacon case with first hardware parts assembled.

Besides the hardware research and tests, a beacon case was planned as soon as the general hardware components were found. Figure 3.3) shows the current beacon case. First, the decision was on the material of the beacon. The chosen material for the first prototype was wood, as it should be hard to break and robust during testing. The outside of the case has thicker material and two holes on the top for the microphone and the loudspeaker. Inside the box, different layers with thinner wood were applied. The first features the power supply, which is a power bank with 5V. The next level above is reserved for the Raspberry Pi board, which owns the plugged USB sound card as well as audio jacks with wire pieces. The top layer has a breadboard where the microphone and the loudspeaker are plugged. A construction manual can be found in chapter 3.3.6.

3.3.4 Problems

Several problems occurred during researching and testing the hardware for the beacon. First, finding a working USB audio interface was rather difficult. It should not be too cheap as many similar cheap sound cards are from the same company. Those were rather bad made and were not working with the Raspberry Pi. Further, the audio interface via the GPIO pins sounded promising but was not compatible with Lineage/Android. Another huge problem was the signal strength. The microphones and loudspeakers were not working without increasing the amplitude but amplifying them too strong was having a negative effect. The noise gets increased by the same level and makes it nearly impossible to detect a signal. Finding a well working pre-soldered one with suitable settings was hard to find. In addition, the perfect operating system to run applications on it, in combination with the lifetime of a started application, its permissions and the setup itself, was a hurdle that needed to be mastered.

3.3.5 Results of Hardware Selection

The hardware research and testing resulted in several valuable outputs. The chosen single-board computer was the Raspberry Pi 3 B, because of the availability of the OS Android for it, the cheap price, the strong enough processor and the on-board Wi-Fi functionality. In combination with the USB sound card from Sabrent, which handles input and output with a sufficient sampling rate of 44.1 kHz, the Raspberry can send and receive acoustic signals. The Adafruit 2.5W class D audio amplifier was chosen because it is boosting the loudspeaker signal without artifacts and is already pre-soldered for most of its parts. The receiving part needs an amplifier too, which is described in the next paragraph. Further, a powerbank as a voltage supply with 5V and 10.000 mAh is enough for running the beacon and can be easily exchanged with a more powerful one. The finished wooden case then holds all components. Parts of the hardware can be easily changed within the beacon.

The solution for the not working microphone amplifier was soldering an own pre-amplifier with a set of transistors, resistors, capacitors and a prepared chip for soldering the components on it. This amplifier is bigger and needs to be soldered, but allows a more fine-grained amplifying process. A circuit diagram of it is shown in figure 3.4. This still creates a problem, as the Bias voltage for the condenser microphones is too weak. The amplifier itself works fine, as it was tested with a frequency generator connected to it. Using a direct Bias voltage, led into overheated microphones. This is not solved yet and needs to be addressed in future. Figure 3.5 shows the finished beacon. The manual for constructing the beacon is described in the following chapter.

3.3.6 Hardware Setup of the UNITA Beacon

The following is a step by step description of how to setup the UNITA Beacon.

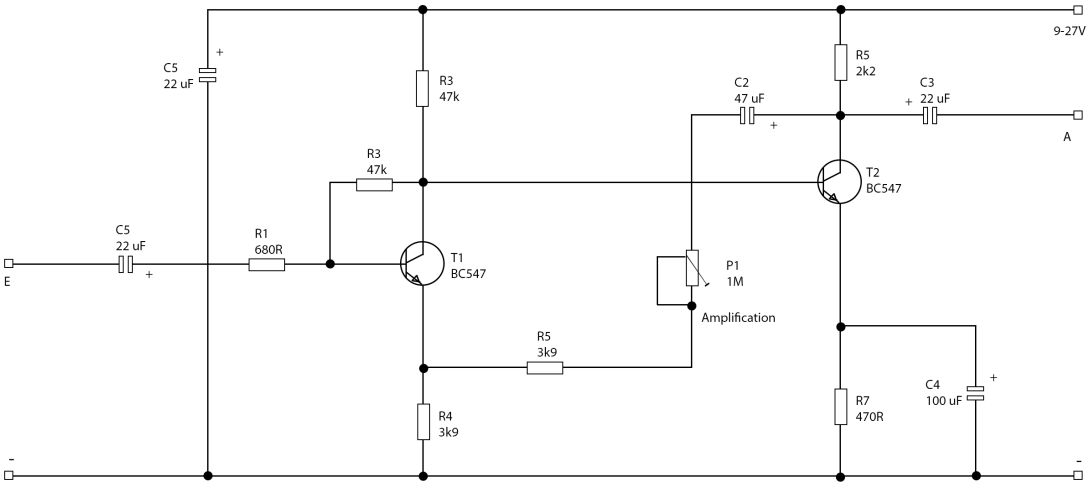

Figure 3.4. The self-soldered amplifier consisting of transistors, resistors ans capacitors.

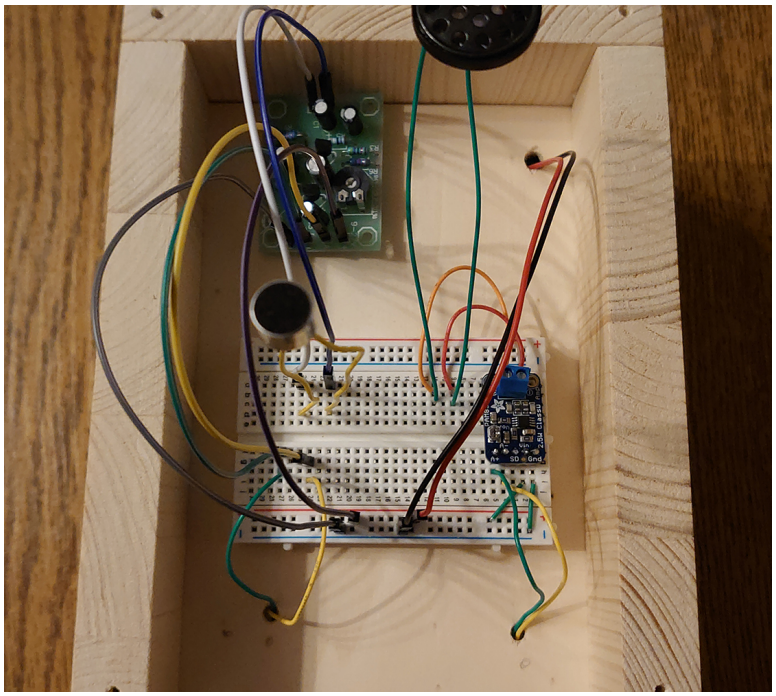

Figure 3.5. All components assembled in the beacon case.

The construction starts with putting the SD-card into the Raspberry Pi 3 B and plugging the USB cable for power. The USB soundcard is then plugged to one of the four USB ports. Next, the two terminal blocks are used to connect the soundcard and the microphone/loudspeaker. The used terminal blocks have on the one side a 3.5 mm audio jack plug and on the other two 2-pole female plugs for wire cables. The two terminal blocks get plugged into the USB sound card and for microphone input and the speaker output. In each of the two terminal blocks, two wire pieces for positive and negative charge get pinned. In addition, the first and the third GPIO pin of the Raspberry Pi, respective 5V and

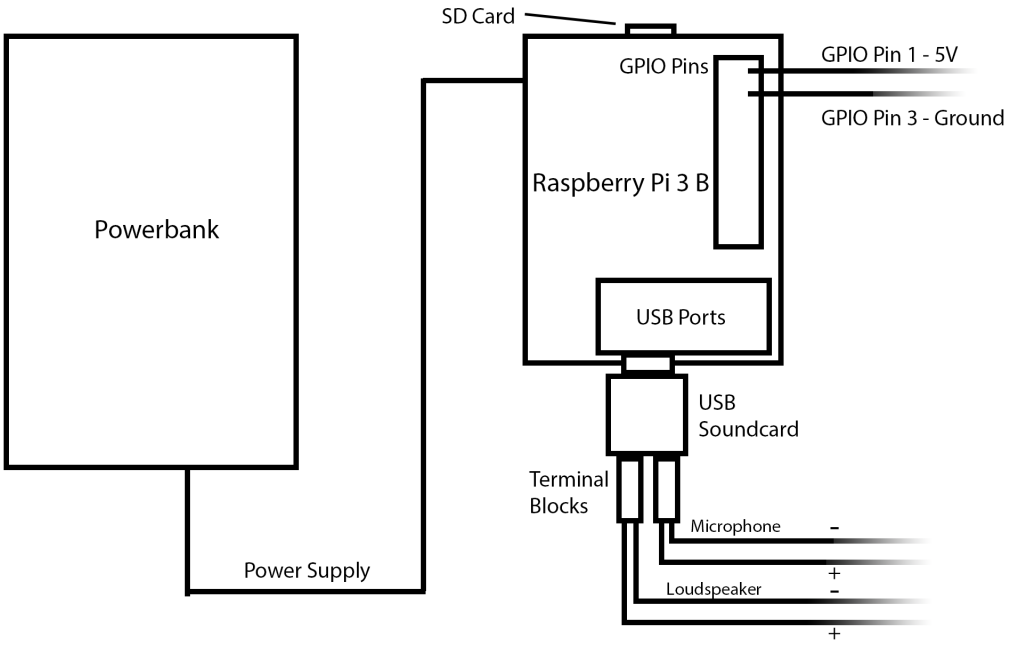

Figure 3.6. The first components including power supply, the Raspberry Pi, the soundcard, and the wire cables for the next components.

ground, get equipped with jumper cables. This will be the power supply for the amplifiers. See figure 3.6 for the circuit diagram and the assembled parts can be seen in figure 3.7.

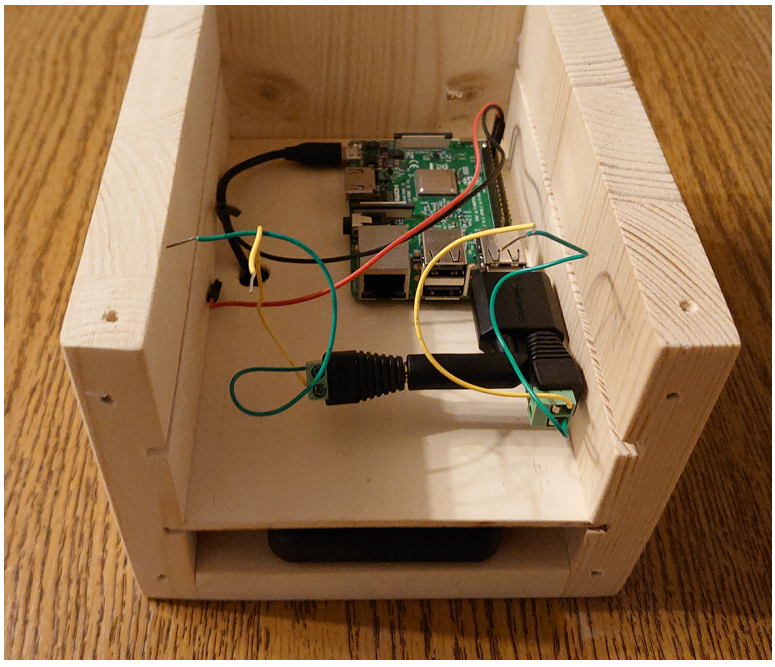

Figure 3.7. The powerbank, the Raspberry Pi, the soundcard, and the wire cables assembled in the beacon.

Next step, the six wire cables from the Raspberry Pi are plugged into the breadboard, seen in figure 3.8 It starts with connecting the two GPIO pins for power into the horizontally connected breadboard holes. Every components which needs energy can access it then.

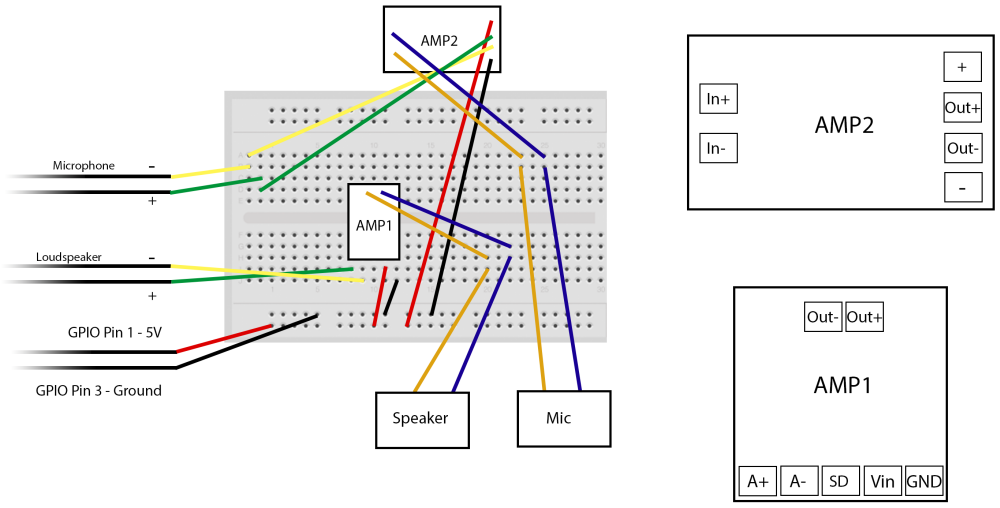

Figure 3.8. The microphone, the loudspeaker and the amplifier wired on the breadboard.

Starting with the AMP1 for the loudspeaker, the two cables from the microphone terminal block get connected to the A+ and A- port of the amplifier. Next step, the positive charge gets plugged on the Vin port of the amplifier and the ground and GND ports get connected. The loudspeaker is connected then via the two output pins, Out+ and Out-. On the microphone side it is similar and starts with connecting the power with the outer two ports '+' and '-' of the ampllifier AMP2. The two wire cables from the terminal block plugged to the soundcard, are then connected to the two output ports as the microphone is plugged to the two input ports. Figure 3.9 shows every component assembled in the beacon.

3.4 Software Implementation

3.4.1 Operating System

Further, a version of Lineage, which is an open source version of Android, was downloaded and an image put on the Raspberry. On the Lineage image side, it was discovered that the OS version 14 runs more stable on a Raspberry Pi 3 B than the version 15. Those were the two latest versions of KonstaKANG[1] corresponding to Android 7.1 (Lineage 14) and Android 8 (Lineage 15). Besides Lineage, an alternative version with Android Things is described in the construction manual. Both options work, but have different demands on launching and running them, described in detail at the end of subchapter 4.3.1.

[1] Lineage images for Raspberry Pi 3 from KonstaKANG https://konstakang.com/devices/rpi3/

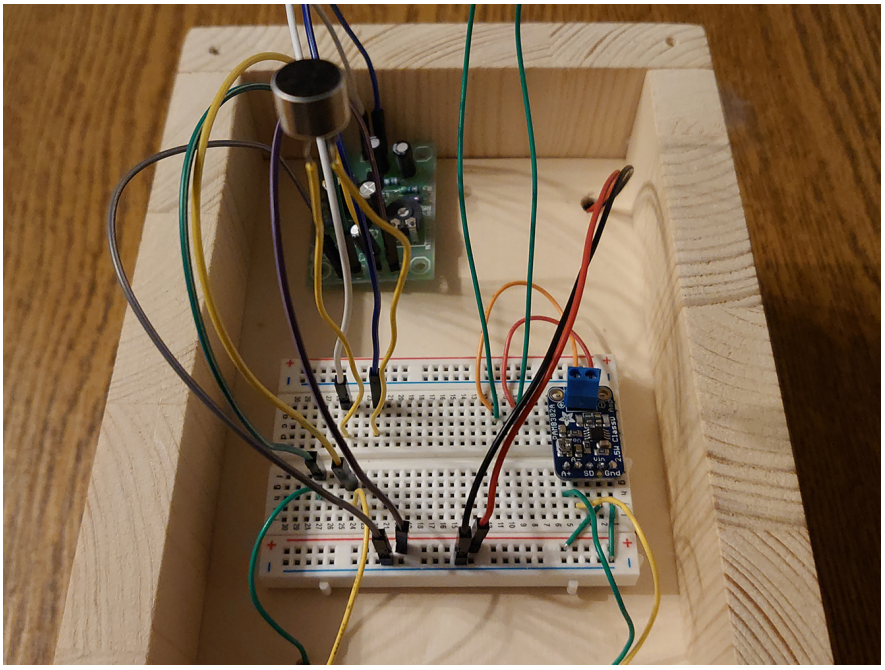

Figure 3.9. The amplifiers, the wiring and the input/output component shown within the beacon case.

3.4.2 UNITA SDK - Client-side

3.4.2.1 Requirements

The main goal of the UNITA SDK was to set up a development basis for other developers, which is easy to use and provides the base functionality for creating an own ultrasonic beacon or client application. Developers do not need to implement the lowest level, represented by the ultrasonic communication protocol SoniTalk, by themselves. The first stack with tasks like creating audio messages, handling the error check or the audio resource release. It needs to offer sender and receiver objects, which interpret the abstract SoniTalk messages. This message interpretation demands a basic structure of message objects, which deliver different information. Further, local storage and database handling should be covered within the SDK, so developers do not need to care about that. Besides that, coping with other communication technologies to communicate with devices like a server, need to be in the SDK. The main functionalities need to be split into controller classes with additional utility classes.

3.4.2.2 Software Architecture

UNITA SDK is split into three parts: the message classes, controller classes and peer classes. Figure 3.10 shows that there are three peer types, five message types and six controller. Further, there utility classes as a support, which are listed in the appendix A,

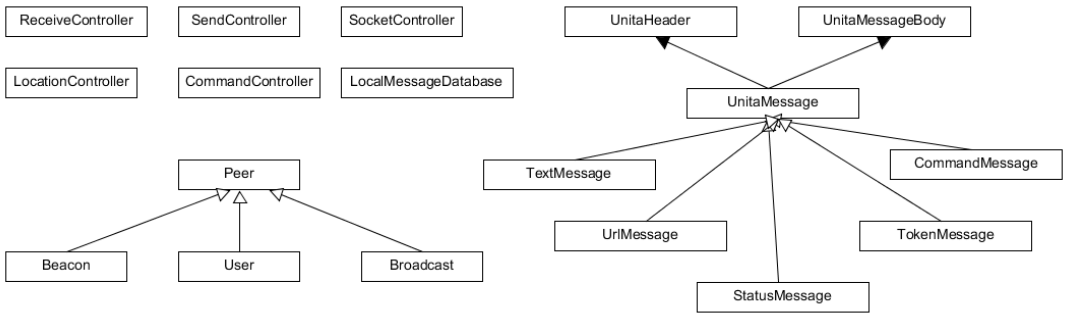

Figure 3.10. Class diagram of the main classes in UNITA. Description in the text.

beside more detailed listings of every class. Further, table 3.3 shows an overview of the features of each controller class.

Class	**Functionality**
ReceiveController	creates ultrasonic receiver to start receiving, handles checking of receivied message type provides interface for received messages
SendController	creates a \texttt{SoniTalkMultiMessage} object, which is described in the following section handles sending and resending of messages
SocketController	opens socket connection to the server handles login, saving messages and getting URLs with the server sends location to the server periodically on demand from the server
LocationController	gets current location of the beacon
CommandController	loads local commands of an implemented application provides a list with all avaible commands
LocalMessageDatabase	operates a local MongoDB database saves messages locally retrieves local messages keeps database open to any kind of message

Table 3.3. List of all implemented controllers and their corresponding functionality.

As mentioned in chapter 3.2 and shown in figure 3.2, the SoniTalk protocol was included for handling the ultrasonic communication. In the SDK, a message is represented by the `UnitaMessage` class which consists of `UnitaHeader` object and a `UnitaMessageBody` object. The structure of the base `UnitaMessage` is important, as every subclass, in applications using the SDK, is inheriting from Further, it has two peer objects representing the sender and receiver and a `headerMessageCode`, which states the message type and is used for decoding a received message. With the `headerMessageCode` value and the two peers, the `UnitaHeader` is created. This leads to a fixed header size of three bytes, as each attribute takes one byte and is therefore limited to 256 values at the moment. In ad-

dition, the `UnitaMessageBody` object just includes the raw message array. Based on this `UnitaMessage`, five types of more concrete messages were established, inherited from the base class and given a unique `headerMessageCode`. Each of those message types stands for a specific ultrasonic communication type. Table 3.4 shows the message types with their main function.

Message type	**Function**
TextMessage	handling private and public messages
UrlMessage	triggering specific functions on the beacon
CommandMessage	handling dynamic REST calls on the client to the server
TokenMessage	handling tokens for different purposes
StatusMessage	handling simple status responses

Table 3.4. List of message types and their corresponding functionality.

Inherited message types can be extended, like the `TextMessage`, which was extended by a `communicationPartner` attribute. Those extensions will be handled as part of the message body when converting the `TextMessage` back to a basic `UnitaMessage`. New message types can be created based on the base `UnitaMessage` in a client application, but are still interpreted as `UnitaMessages` in the SDK.

Next, three peer types were implemented to address all common receivers and senders:

- Beacon with dynamic name and id
- User with dynamic name and id
- Broadcast with only id, which is fixed to 1

3.4.2.3 SoniTalk Extension for Longer Messages

As previously mentioned, the open source ultrasonic communication protocol SoniTalk was chosen as the first layer of UNITA. This part handles the hardware resources and the generating, sending and receiving of ultrasonic signals. As soon as the ultrasonic layer receives a correct message or the UNITA SDK wants to send a message, the two layers communicate with each other. Besides the ultrasonic part, the permission system of SoniTalk was taken over to the UNITA SDK and needs to be implemented and checked. This permission is checking on sending and receiving ultrasonic signals.

Further, UNITA needs to send longer messages then SoniTalk previously defined. Therefore, the open source protocol needed to be extended. This happened in the form of a `SoniTalkMultiMessage` in addition to the existing `SoniTalkMessage`. This new class just needs the raw byte array message, seen in the listing 3.1.

Listing 3.1. SoniTalkMultiMessage constructor

```
1 public SoniTalkMultiMessage(byte[] message)
```

Further, the single `SoniTalkMessage` got extended by a header on the SoniTalk level. The header consists of:

- message-id
- packet-id
- number of packets overall

The size of the header is fixed, seen in listing 7.1. For every header entry one byte is available, which allows the use of the values 0 to 255. Next, the `SoniTalkSender` gets extended by a sending function for `SoniTalkMultiMessages`. Depending on the length of the message, the number of packets get calculated and `AudioTracks` get created within a `SoniTalkMessage`, see listing 7.2 and listing 7.6. On calculating the number of packets, the header has to be included into the calculation for every single `SoniTalkMessage`. A `SoniTalkMultiMessage` is split into several `SoniTalkMessages` by this process, ready to send. See listing 7.4 and listing 7.5 for the detailed split.

On the receiving side, correct received ultrasonic messages get converted back into a `SoniTalkMessages` again. Depending on the value for number of packets, it will either be directly forwarded to the UNITA SDK layer, if it is just one packet, or collected depending on the `message-id`. This stored `SoniTalkMessages` will be concatenated to a `SoniTalkMultiMessage` object again, as soon as all packets were received.

3.4.3 Used Protocols

3.4.3.1 WebSockets

The WebSocket protocol uses a signal TCP connection for creating a bidirectional communication channel. An existing bidirectional polling HTTP infrastructure, which was used previously, can be used for the WebSocket protocol as it can use the same HTTP ports 80 and 443. Further, it consists of two parts: a handshake and the data transfer. First, the client and server have to fulfill a handshake. On a successful one, the data transfer starts and "messages" can be sent in both directions. The WebSocket protocol is conceptually seen, a layer on top of TCP, which provides a web origin-based security model for browsers, adds a naming mechanism and addressing for multiple service support. Furthermore, a framing mechanism is attached on top of TCP without a length limit and adds a closing handshake to securely close the socket connection (Fette and Melnikov, 2011).

3.4.3.2 REST

Representational State Transfer (REST) stands for an architectural style of a distributed hypermedia system with software engineering principles and interaction constraints. REST uses a client-server style, which is creating the possibility of portability over multiple platforms and improves scalability. Further, it is stateless and therefore a client needs to save

information about the server in a session state. On improving the network efficiency, a cache is implemented on the client-side to store information within responses from a server. In addition, REST has a uniform interface that is decoupled from the service provided. Furthermore, REST builds upon a layered system, where every layer has a restricted view on only the next layer they are interacting with. Finally, code can be downloaded on demand in a form of applets or scripts via REST. Whole features can be requested, but it reduces visibility, which makes it an optional constraint on REST (Fielding, 2000).

3.4.3.3 Functionality

The following section shows important parts of the functions of the SDK, which are needed to get and send messages. Starting with the receiver and sender, and followed by the socket functionality and a brief look on the utility functions.

The `ReceiveController` is a Singleton, as the SoniTalk protocol, on which UNITA is based of, has a limitation of only one receiver at the same time. The initialization can be seen in listing 7.8. The controller starts then receiving by creating a `SoniTalkDecoder` object and executing the starting function of it. Further, the interface for receiving the by SoniTalk forwarded messages, was implemented, which can be seen in listing 7.9. Either correct messages or errors will be received. On correct messages, the `ReceiveController` checks the message code in the header of the received message to identify the message type and convert the `UnitaMessage` to the new type, see listing 7.10. An interface was implemented to forward the converted message to the application layer, as seen in listing 3.2 and 7.11.

Listing 3.2. Creation of message listeners for all implemented message types.

```
1 public interface BeaconListener {
2     void onUnitaMessageReceived(UnitaMessage receivedMessage);
3     void onTextMessageReceived(TextMessage receivedMessage);
4     void onCommandMessageReceived(CommandMessage receivedMessage);
5     void onTokenMessageReceived(TokenMessage receivedMessage);
6     void onUrlMessageReceived(UrlMessage receivedMessage);
7     void onStatusMessageReceived(StatusMessage statusMessage);
8     void onUnitaMessageError(String errorMessage);
9 }
```

The `SendController` implements the functionality for creating messages and converting the `UnitaMessages` into `SoniTalkMultiMessages`, seen in listing 7.12. Next, a separate thread is started for the sending process. Depending on the number of packets of the `SoniTalkMultiMessage`, the sender emits the same amount of messages. Further, it has the possibility to resend messages, if they are not received by someone. The sending process in detail can be found in listing 7.13.

The `SocketController` creates a socket connection to the specified server. Besides listeners for standard events like connect and disconnect, the login, the URL responses and the location retrieval got listeners implemented. These listeners get triggered by the server either automatic by the server or get activated by sending a JSON object to the server. This gives the server the possibility to check the location of the beacons via the `LocationController`. An example can be seen in listing 7.14.

Additionally, to notify the layer above, four listeners were implemented, see listing 3.3.

Listing 3.3. Several listeners for socket events.

```
1 public interface SocketListener {
2     void onSendMessageResponse(JSONObject messageResponse);
3
4     void onGetUrlForCommandGetAllMessagesResult(JSONObject
        ↪ urlResponse, JSONObject senderResponse);
5
6     void onGetUrlForCommandgetAllBroadcastMessages(JSONObject
        ↪ urlResponse, JSONObject senderResponse);
7
8     void onGetUrlForCommandgetAllMessagesFromContact(JSONObject
        ↪ urlResponse, JSONObject senderResponse);
9 }
```

In addition, to the controllers, utility functionalities were added to support the main functions. The main utility is the `MessageUtils`, which handles the conversion of different message types with the base `UnitaMessage` and with the `SoniTalkMessage` and `SoniTalkMultiMessage` form the lowest layer, seen in listing 7.16. Besides the utility for the messages, the `LoginUtils` handles the saving and getting of logged-in beacons, for retrieving data on every occasion, seen in listing 7.15.

3.4.3.4 Classification in the OSI-Model

UNITA can be categorized into the OSI-model. Starting with layer one, the physical layer, which is represented by the ultrasonic sending via the loudspeaker and receiving via the microphone. The encoding protocol on this layer is done via the SoniTalk protocol. The second layer, the data link layer, is performed through ordering the bits in a kind of barcode over several frequencies. Further, adding a cyclic redundancy check (CRC) is happening on this layer, which makes a frame out of one packet. Next, the third layer, the network layer, is represented through the list of connected beacons, which serves as an address table. In addition, it adds the sender and receiver data in form of the `SoniTalkHeader` and splits the multi messages (segments) into `SoniTalkMessages` (packets). The fourth layer checks the CRC at the `ReceiveController` of UNITA SDK, whereas the resending

of segments happens on the level of the `SendController`. The fifth layer, the session layer, is authenticating through the login in the UNITA SDK. The sixth and seventh layer, presentation and application layer, are fulfilled by the use case application on top of the UNITA SDK.

3.4.3.5 Challenges and Remarks

The biggest challenge was keeping the SDK as atomic as possible to stay independent and flexible. Though, as many common functions as needed should be in the development kit. The main problem during development was to decide, which features should be part of the SDK. Often, components, which fit more into the application part, were already implemented in the SDK level and had to be removed again. Due to the low data rate, problems in deciding for the header element appeared. Creating a fixed header on the SoniTalk extension, was on the one side helpful as it is quite slim, but limited the number of bytes, which was still marking a problem. Furthermore, the decision, on which message types for which purpose need to be included, was rather difficult. Additionally, the save of local messages was quite a hurdle at the beginning, because several options were available, like saving it on a file, saving it inside the app as a SharedPreference or using a full-featured local database.

3.4.4 UNITA Server - Server-side

3.4.4.1 Requirements

To get access to the internet and exchange and maintain data of different beacons, UNITA needed a kind of backend. This backend needs to fulfill several requirements:

- It should offer a platform for maintaining application and beacons.
- The information processing should take place on the backend.
- Messages should be uploaded.
- Messages should be further stored.
- Stored messages should be retrieved.
- Continuous location tracking should be possible.
- The tracking of active beacons should be implemented.
- The login of beacons and clients should be handled by the backend.
- Beacons should be shown on a kind of map.

3.4.4.2 Structure

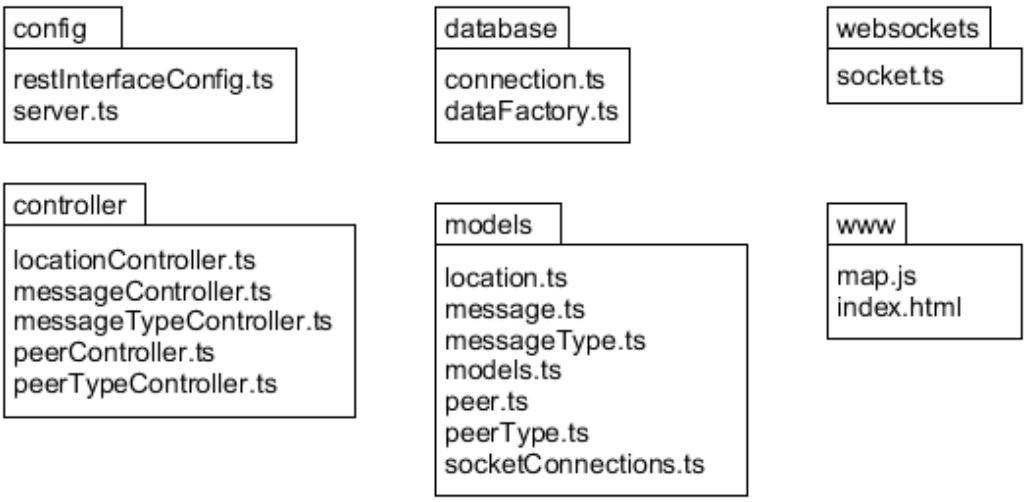

Figure 3.11. Class diagram of the UNITA server with the six packages, which contain the corresponding TypeScript-files.

The programming base of the server was a Node.js application with the extension Express.js written in TypeScript and JavaScript. Express.js provides a simple REST interface and can be easily used to set up all endpoints of the interface. The server was split into six packages, as seen in figure 3.11: config, controller, database, models,websockets and www.

The `config` package consists of the initialization of the server in the `server.ts` file and is responsible for the REST interface. Further, it includes the names of the REST interface endpoints in the `restInterfaceConfig.ts`. Next, the controller package keeps the logic of the server. It handles the location retrieval on the beacons current position in the `locationController`. Besides that, messages get saved to the database by the `messageController` and `messageTypeController`. Then they get retrieved and the message types get stored.

Additionally, the `peerController` and the `peerTypeController` keep track of the peers, like beacon and users, and its types. Further, the `peerController` takes over the login of beacons and clients. Next, the database package consists of a `connection.ts` file, which handles the connect functionality to the database, and the `dataFactory.ts`, which initializes the database with the static data on the start. The `websockets` package includes the `socket.ts`, which is implementing the socket listeners and emit functions for all socket events like login or save message to database. Last, the `www` package contains the map, which is split into a html-file and a JavaScript-file, for visualization.

3.4.4.3 Database

All peers and several messages should be saved on the server-side. Therefore, MongoDB was chosen to get a mostly dynamic database infrastructure. MongoDB operates as a NoSQL database and uses a document-based way of saving data. Further, this allows UNITA to save messages from several applications, without creating new tables, with its own structures. UNITA messages have a basic structure defined in the SDK, which is kept on the server-side as mongoose models. Mongoose is a plugin for Node.js to simplify the work with MongoDB. Those models were created for messages, message types, locations, peers, peer types and socket connections. An example of one model is seen in listing 3.4, describing the message object. It shows the different attributes of the model like sender, receiver or message, and its corresponding type for the database. Further, the field unique says, if the value of the attribute needs to be unique like for example an identification number. Additional information can be stored as a separate field, shown in the following subchapter.

Listing 3.4. Database model example scheme.

```
1  const messageSchema: Schema = new mongoose.Schema({
2          headerMessageCode: {
3                  type: Number,
4                  unique: false,
5          },
6          sender: {
7                  type: Number,
8                  unique: false,
9          },
10         receiver: {
11                 type: Number,
12                 unique: false,
13         },
14         communicationPartner: {
15                 type: Number,
16                 unique: false,
17         },
18         message: {
19                 type: String,
20                 unique: false,
21         },
22         messageRaw: {
23                 type: Array,
24                 unique: false,
```

```
25          },
26          timestamp: {
27                  type: Date,
28                  default: Date.now,
29          },
30 },
31 {
32          strict: false
33 });
```

Before using MongoDB it needs to be connected, which looks like in listing 7.17 in the connection.ts file. On a correct connection, the predefined static data is created through a data factory object. This includes peer types, message types and fixed peers like the broadcast address.

3.4.4.4 Functionality

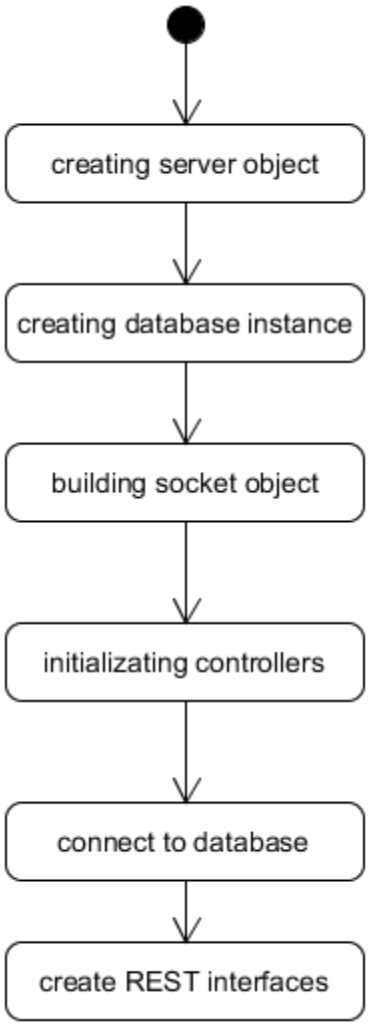

Figure 3.12. Activity diagram of the server initialization.

The `server.ts` starts with creating a server object of Express.js, seen in figure 3.12. Next, a instance of the database is created for later. Further, a socket object is built and the controller, which are used in the server.ts, get initialized, see listing 7.18 for details. Afterwards, the connection to the database happens via the previously created database instance, as seen in listing 7.19. Last, all endpoints of the REST interface get created, example in listing 7.20 and 7.21. The endpoint addresses are either static, for fixed access points, or dynamic through taking the endpoint name from the `restInterfaceConfig.ts` file. These dynamic endpoint addresses are used for the clients, which have to ask for the specific name via a beacon. This way, the endpoint name can be changed easily.

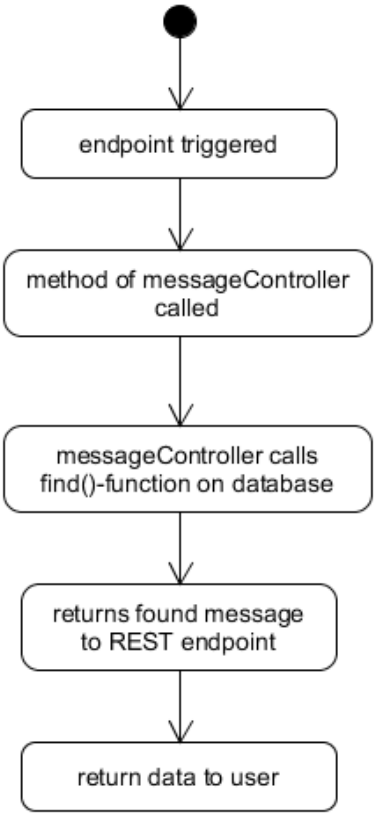

Figure 3.13. Activity diagram of an endpoint call for retrieving messages.

On requesting to a REST interface for messages, the corresponding endpoint gets triggered, as seen in figure 3.13. Next, a method of a controller gets called, for example the `messageController`, an example seen in listing 7.22 The controller calls then the find()-function on the database and gets messages returned. These are returned to the REST interface. From there, they are returned to the user. Another example is the saving of a message to the database, shown in listing 7.23. To stay independent from the message types and different applications, every value, which is not a key value, is saved in the field `additionalData`.

The `peerController` keeps track of the login functionality. Both client and beacon login, are handled inside this controller. If the peer exists, the peer can get logged in, otherwise a new peer objects gets created in the database, seen in listing 7.24. Further, the socket.ts has all WebSocket events implemented and handles the input and output via this channel. Examples are in listing 7.26 and 7.27. All connection and data exchange with beacons happens through the socket implementation. Examples of integrated socket endpoints are: saving a message on the server, getting a URL for the REST interface call of the client or sending a location check. For example, the `"sendMessage"` socket event is emitted from the beacon by calling `"socket.emit('sendMessage', messageData)"` and received on

the servers event listener seen in listing 3.5. Next, a controller is triggered and process the provided data or retrieves data from the server depending on the request. Last, a new socket event with the new data gets emitted and the listener on the beacon side receives the information.

Listing 3.5. Socket listener for sending messages.

```
1 socket.on("sendMessage", (message) => {
2         this.messageController.saveMessageToDB(message).then((
      ↪ savedMessage) => {
3               socket.emit("sendMessageResult", savedMessage);
4         });
5 });
```

3.4.4.5 Map

For keeping a good overview on how many active beacons are available, a map was integrated into the server. It should show further information and the position where the beacon is located. This is realized by a simple HTML file with a script written in JavaScript behind it. Further, this script communicates through WebSockets and retrieves the data from there. The used map source is OpenStreetMap and active beacons are shown as markers with popups containing additional data. First, active beacons are identified by taking all open socket connections and caching their IDs. Next, those IDs are used to request all connected location points in the database. This amount of position data is then filtered by the timestamp for the last added entry. These entries get forwarded to the map and represent the markers, as seen in figure 3.14.

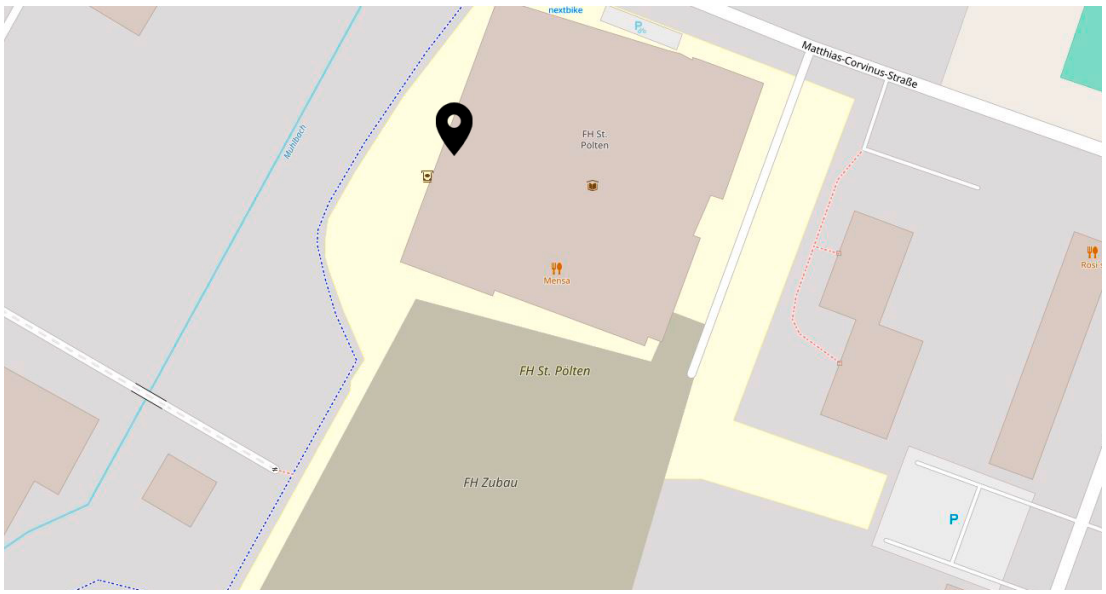

Figure 3.14. One beacon placed at the entrance of the university.

3.4.4.6 Challenges and Remarks

The server needed to store different message types with different sizes of attributes, without creating an overhead for handling each type in a different way. Besides what to store, the question was in which form and structure and where to save it, to keep the server and database independent and easy to expand. Further, how to implement the interfaces for the beacon and the clients was another hurdle to take, because it needed to be generic and quickly changeable. Additionally, how to provide the endpoints to clients to keep them as independent and flexible as possible, was another challenge. The main problem was finding a way to add additional data for each database entry, which was solved by checking the incoming data for all key values of the predefined model. Object attributes not included, were collected and stored as a separate database field. Further, keeping the REST endpoints dynamic was a bigger question at the beginning. In addition, tracking active beacons and deleting them without getting a disconnect from them, was an issue during development.

4 Proof-Of-Concept App "SocialWall"

The following chapter includes the process of creating use cases, choosing the main use case and implementing the proof-of-concept app based on the chosen use case. Further, other possible use cases get described.

4.1 Use Case Definition

A major goal for UNITA was implementing a use case as a proof-of-concept application. In the beginning, several use cases were conceptualized, to get an overview of what functionality the UNITA SDK should contain. First, four topics were defined, which give the basis for the following use case brainstorming. These topics include home automation, data sharing, security, and gaming, further described in the following subchapter. Further, twelve use cases were gathered and briefly described, whereas one was chosen to be the proof-of-concept application. This application gets a separate app for the beacon and client.

To get a good view of possible areas of application, four superior topics were defined more precisely. The areas were taken in a balanced way to get more different use cases. Starting with data sharing is probably the first-mentioned when talking about ultrasonic communication. This contains the simple sending and receiving of non-sensitive data in terms of extended information purposes. Information can be left or retrieved at a specific point. Next, security represents an own topic, consisting of areas like authentication and cryptography. Ultrasound communication stands here for sending sensitive data and getting access to something or somewhere. Further, gaming is a topic where ultrasound can be used as either the interaction method or the game base itself. Last, an important field is the area of home automation. Here ultrasonic communication works as an extension of sensors or as a network between devices.

The ten created use cases withiout the main use case are:

- Information exchange for fair booths - A beacon would serve as an extension for fair booths. Users can collect information about, whoever is running the booth, just by interacting with the beacon. A general query can be sent and the beacon is responding with information and further interaction possibilities. It starts a conversation at the ultrasonic level and is built like a dialog tree.

- Additional information for museums exhibits - Items of an exhibition in a museum could be equipped with ultrasonic beacons. Those would send out the same sequence of symbols all the time and trigger something in the client application. In this case, the client would only listen to the beacon and gets more information by unlocking new parts of the application through ultrasound.

- Broadcast of important information in public areas - Important information can be broadcasted for people. Ultrasonic beacons operate as information antennas and forward data in a continuous inaudible acoustic stream. The client application then notifies the users and shows them what happened.

- Unlocking of doors for entrance - Ultrasonic beacons can be used for unlocking doors in a secure way. Via ultrasound, a special identification number is exchanged between the client and beacon to give the user entrance somewhere. This process is kept even more secure by decreasing the volume of the message and therefore, reducing the distance for possible attacks.

- Ultrasonic tickets for events - Instead of using physical paper or digital file-based tickets, a specific ultrasonic sequence can be used as authentication. The audio message gets triggered by the ticket check beacon and sends as a response the signal to authenticate the user. The acoustic ticket is kept secret until the process at the beacon and prevents thereby illegal copies.

- Mobile payment - Mobile payment can be achieved by using ultrasound. Multi-factor authentication through an acoustic token is a way of getting an additional security channel. In combination with an existing method, like pin code, another method gets established. A one-time password is created on the payment server, is then forwarded to the application of the client and is emitted as an ultrasonic message for the payment beacon. The ultrasonic beacon then receives the signal and decodes it with a further comparison of the original password on the server.

- Geo-caching - A playful approach for ultrasonic beacons is the game geo-caching. The user needs to find tags and has to scan them to get points or leave messages for other people. These tags get represented by ultrasonic beacons which the user needs to find by listening to them or triggering them via the client. This depends on the way the beacon behaves, either active or passive. The active option uses continuous messages which the user needs to scan with the client. The passive would require triggering through signals from the client application.

- Ultrasonic beacon as a gaming console - An ultrasonic beacon can represent a gaming console interface. It reacts to different ultrasonic messages and executes the respective command on the game behind displayed somewhere. The client application stands for the controller and sends user inputs to the beacon. Several different

games, which are not time-dependent, as the communication needs time, and are designed for single-user inputs, can be developed.

- Home automation - Home automation can be supported by ultrasonic beacons. Smart sensors can be equipped with a beacon and send an ultrasound signal to the device of the user as soon as a specific threshold is reached. For example, a humidity sensor of a plant sends a message if the moisture is reaching a specific low level. The client application then receives this signal and displays a message for watering the plant.
- Communication between beacons on specific conditions - A machine-to-machine communication (M2M), more precisely a beacon-to-beacon communication, is a possible use case for ultrasonic beacons. A network of different sensors and home automation devices can be built by using the beacons. If a predefined condition is fulfilled a specific action will be executed. Those events can be implemented before, including various actions for different available components.

The eleventh use case was chosen as the main use case, the ultrasonic blackboard was chosen and called SocialWall, which is further described in the next section.

4.2 Main Use Case - SocialWall

From the previously described use cases, the SocialWall offers a broad implementation of the UNITA SDK features and further, a good way for testing it with users. It uses most of the communication features and protocols, which makes it even better for proof-of-concept. SocialWall needs bidirectional ultrasonic communication, which means both the beacon and client need to send and receive signals. The application uses the pairing functionality in combination with status messages. It creates two new inherited message types and uses all of the by the SDK provided message types. Besides that, peers play a role in the application as well as the private and public in the shape of text messages and a local database storage. For testing, SocialWall further provides a common interaction model which is already known by users, as they are accustomed to messaging apps. This gives the possibility of getting better data about the ultrasonic communication process, while not focusing on understanding the application itself. In addition, suggestions about the application interface can be gathered.

SocialWall belongs to the data sharing topic and is, in general, an ultrasonic blackboard. The idea is to leave and retrieve messages with a smartphone. This only works on-location at the blackboard, which is represented by the developed ultrasonic beacon. A user can register and log in with the client application and then needs to pair with the black board via ultrasonic pairing messages. Further, the user can write either private or public messages, which differ in this respect by the storage method. As the blackboard has an internet

connection, public messages get stored online, whereas the private ones are directly saved on the beacon. Using ultrasonic commands, the user can receive left messages. Though a user can only get the belonging private messages, all private messages as they are shared by the other users. For sending private messages, a user has to add contacts to the contact list. This application/blackboard can be used in different ways. It could serve as a guestbook, as a kind of mailbox or as a replacement for a physical blackboard.

4.3 Implementation

The following part describes the implementation of the beacon side and the client side application. Figure 4.1 shows the process of SocialWall. The continuous lines represent ultrasonic communication, wheareas the dashed arrows are for the Internet connection and the dotted lines stand for internal communication. Starting with User A sending a private message to User B. This message is then saved on the local beacon database as it needs to be at a private level. User B asks then for the latest private message of User A from the beacon. The beacon retrieves this message from the local database and returns it to the user. The same happens the other way round, as User B sends a private message for User A and User A retrieves this message. The private communication is shown as blue arrows. The green arrows represent the public messages, shown with User C, who sends a private message to the beacon, which is then sent to the server via the Internet. The server further saves this message on its database. User C then checks for all available messages and the beacon retrieves them from the server and its database and returns it to the user.

4.3.1 Implementation Beacon

The beacon application is written in Java and implemented for Android like the UNITA SDK. Figure 4.2 shows the first steps of the initialization of the beacon application. The process starts first with the creation of the socket connection for further communication with a server. Next, the beacon logs in on the connected server. Further, the `ReceiveController` gets initialized, beside the `LocationController` for providing the current location of the beacon. Last, the `SendController` for emitting ultrasonic messages. See listing 7.28 for source code details.

In addition, listeners for the responses of the server, respectively the `SocketController` get implemented. The listing 7.29 shows an example of a listener in the wake of the login. Further, the response of the login is saved on the beacon. Besides the login, listeners for all message types are implemented, to retrieve forwarded messages from the lower layer, UNITA SDK. An example of a message type listener is shown in listing 7.30. Additionally, every important socket event gets listener from the `SocketController`, see listing 7.31 as an example. Besides that, the settings functionality with using presets and changing them,

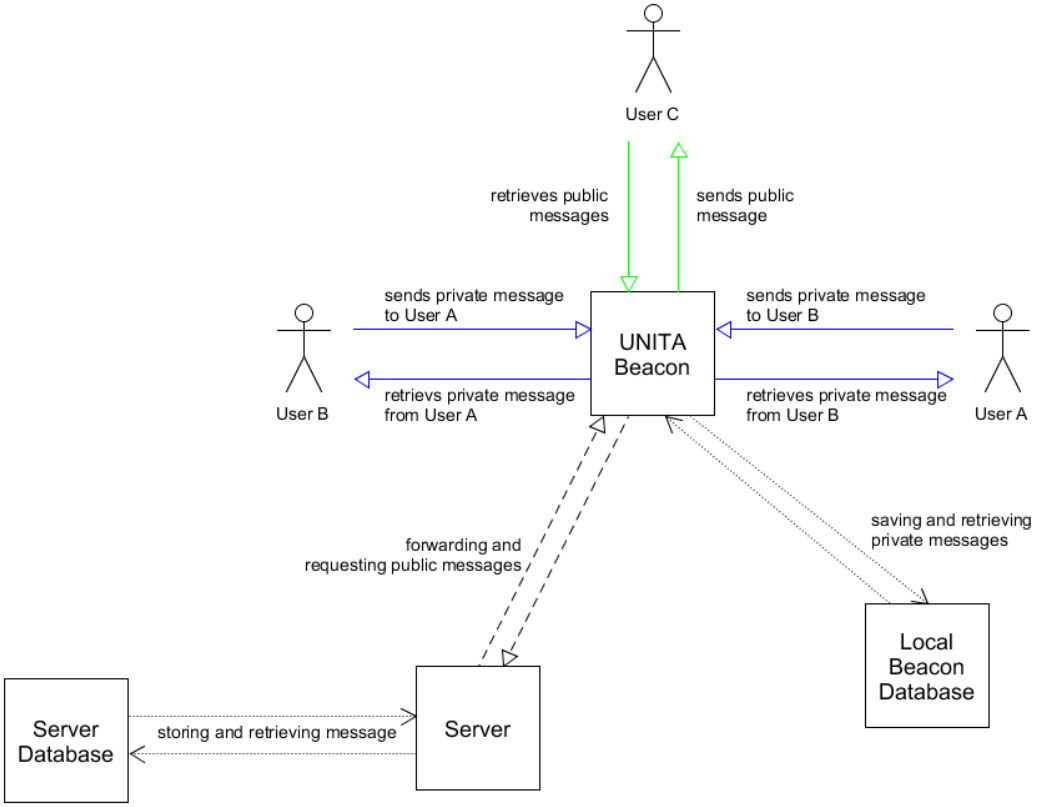

Figure 4.1. Use case visualization of SocialWall, with all communication possibilities and interacting components.

is implemented in the `MainActivity` as well as all listeners for the implemented SoniTalk data-over-sound permission.

Every sending and receiving process is handled via the Routines class. What happens on and with which message type is described there. Examples are shown in listing 7.32. For more flexibility, SocialWall inherited two new message types. The `PairingMessage` gets inherited directly from the `UnitaMessage` class. This message type is used for the pairing process of a client with the beacon and does not implement new attributes. The second new type is inherited from the `TextMessage` of UNITA SDK and introduces the new attribute `isPublic` to check between private and public messages. In addition, status codes, service constants and several commands are created.

There are two ways to start the beacon. The first would be with a Lineage image on the Raspberry Pi. Starting the Raspberry Pi a screen needs to be connected. After the boot, the Wi-Fi connection needs to be established like with a normal Android device. Next the application will be uploaded to the device for example via the Internet connection an a cloud service of your choice. After installation, the app can be started, all permission needs to be accepted and then it should already run.

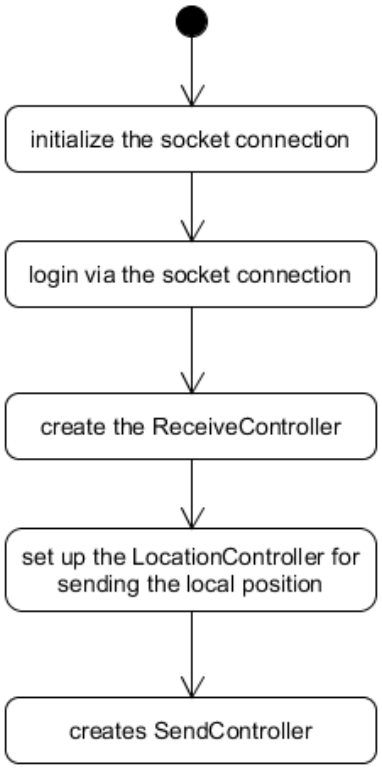

Figure 4.2. Initialization of the beacon.

The second way is using the Android Things console. After logging in with a Google account, a new project is added. Besides entering the name and description of the application, the device model has to be chosen, which is Raspberry Pi in the case of SocialWall. Tapping on the model, forwards to the build and release page, where a new application can be added via the new button. Here, the option "Start from scratch" is used and opens a new menu for entering the application data. First, a build configuration name needs to be written and then the newest Android Things version (1.0.15.5796897) is chosen. Further, the built apk-file of the beacon application has to be uploaded. Before building, the manifest of the Android app needs modifications to open the launch activity in the correct way, seen in listing 4.1 with the two new lines and the commented line of code for the previous described build version.

Listing 4.1. Changes of manifest file for launch activity.

```
1 <category android:name="android.intent.category.HOME"/>
2 <category android:name="android.intent.category.DEFAULT"/>
```

In addition to work with Android Things, the corresponding library needs to be included as described in the GitHub repository[1]. The manifest needs two further entry, which de-

[1]Native library for Android Things https://github.com/androidthings/native-libandroidthings

clares the usage of the Android Things library and the accompanying permissions. (see listing 4.2)

Listing 4.2. Permissions and usage of Android Things library.

```
1 <uses-permission android:name="com.google.android.things.
    ↪ permission.USE_PERIPHERAL_IO"/>
2 <uses-permission android:name="com.google.android.things.
    ↪ permission.MANAGE_INPUT_DRIVERS" />
3
4 <uses-library android:name="com.google.android.things"
5         android:required="false"
6         tools:replace="android:required" />
```

After the apk upload, all permissions have to be accepted and no changes to the hardware tab and to the partition tab are needed. The build is available in the overview and can be downloaded as an image to be flashed on the SD card of the Raspberry Pi. Furthermore, after starting the Raspberry Pi with the application, it needs to be connected with a router to connect to it from the computer via the Android Things setup utility tool [2]. Running it, the setup of the Wi-Fi can be achieved and the device will be found via the tool. Then, it can be used.

4.3.2 Implementation Client

The client consists of five activities and four fragments.

- Activity:
  - BaseActivity
  - LoginActivity
  - ContactActivity
  - MainActivity
  - PairedBeaconActivity
- Fragments:
  - PairingFragment
  - MessagesFragment
  - SendFragment
  - SettingsFragment

[2] Download of setup utility tool https://partner.android.com/things/console/u/0/#/tools

The `BaseActivity` implements the options menu and the action bar items. The remaining four activities inherit from the basis one. Starting with the `LoginActivity`, users can enter a name to login via REST and a correct response from the server leads to the `MainActivity`. This activity contains the four fragments, which are described in the next paragraph. Further, the contacts and paired beacons have their own activity, to show them in a list and to allow them to remove them again.

Figure 4.3. Screenshot of client application of the pairing fragment.

The four fragments handle pairing, sending, messaging and keeping the settings. The first fragment is responsible for the pairing process and can be seen in figure 4.3. It contains a button for starting the process and shows additionally the current status of it.

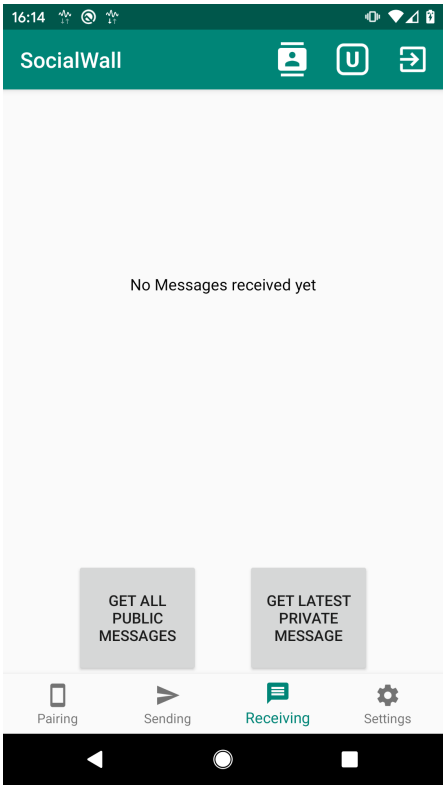

Figure 4.4. Screenshot of client application of the messages fragment.

Further, messaging fragment shows all retrieved messages in a list. In addition, commands for retrieving messages at a beacon are offered, shown in listing 4.4. Next, the sending fragment follows and gives the user the opportunity to send private an public textmessages, shown in figure 4.5. On the client, the corresponding paired beacon, which the message should get, can be chosen and the message text can be added. By switching the radio button, an option for a communication partner is enabled, if a private message wanted to be sent. The source code example can be found in listing 7.34.

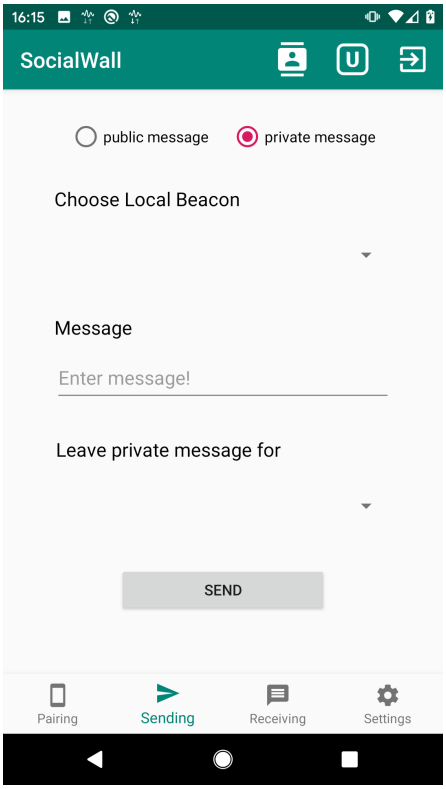

Figure 4.5. Screenshot of client application of the sending fragment.

The fourth fragment keeps the settings handling. Presets are offered and parameters for receiving and sending messages can be changed individually. As the user has to communicate with the server via a REST interface, two classes were implemented to enable this. The interface `SocialWallAPI` handles the endpoints which can be addressed, see listing 7.35, and the `RESTController` creates a REST instance for connection via the external library Retrofit.

5 Evaluation

The evaluation happened in the form of a user study based testing with the proof-of-concept application SocialWall. The goal was to get to know the user acceptance and usability of ultrasonic communication, as well as an evaluation of the use case itself.

5.1 User Study

More information about the possible usage of ultrasonic beacons needed to be gathered. This happens through testing the proof-of-concept application and asking further question about acceptance and trust on ultrasound. The following sections show the aims, the study itself, the research questions, and the tasks which need to be performed.

5.1.1 Aims of the User Study

On the one hand, the application SocialWall should be tested on how intuitive the smartphone client is and what improvements can be achieved. On the other, the communication itself with the UNITA beacon is part of the test. Moreover, the users' attitude on ultrasonic communication will be inquired.

5.1.2 Target Group

The target group of the UNITA beacon with the application SocialWall are people, regardless of their gender, between 20 and 40 years. The education level and their income are not relevant. Though, their smartphone usage and experience should be at least at the average or higher. They want to try out digital version of physical items or processes. Further, the target group already uses conventional messaging applications and is interested in trying out new apps.

5.1.3 Study Design

Figure 5.1. Steps of the user study, described in the text.

The user study, see figure 5.1, starts with question about the demographics and smartphone usage of the test person. Next, a questionnaire based interview about previous experience with ultrasonic communication was added, followed by a system acceptance scale to measure the acceptance of ultrasound in general (Van Der Laan et al., 1997). Afterwards, tasks on using SocialWall have to be executed. Further, a system usability scale on the usage of the beacon and the application, after fulfilling the tasks, was included. Last, a second questionnaire based interview about the usability of the proof-of-concept application SocialWall was included. Different types of questions were used for the interview, either yes-or-no-questions in special cases, open questions or Likert scales with situational enquiries. The full questionnaire is listed in the appendix C.

5.1.4 Research Questions

Following questions should be answered by testing the ultrasonic beacon with the application SocialWall. They are split into two parts, general questions about ultrasonic communication and specialized ones on the UNITA beacon and the application itself:

- Ultrasonic Communication
  - Q1 Would people accept and trust ultrasonic communication?
  - Q2 Would ultrasound be a useful alternative for other technologies?
  - Q3 Which use cases would be suitable for ultrasonic communication?
- Unita Beacon and SocialWall-Application
  - Q4 Would people like the idea of a location-based ultrasound black board?
  - Q5 How intuitive is the use of the beacon with “SocialWall”?
  - Q6 Would people need a kind of additional feedback?
  - Q7 What improvements are needed to be more user-friendly?

5.1.5 Task Description

The target group can leave public and private messages for other people. They pair with the UNITA beacon to use the application SocialWall. The application is location-based and private messages can only be retrieved on location. There are several reasons why they use the application SocialWall. First, they can use the possibility of communication without enabling further technologies like Wi-Fi or Bluetooth. They can connect with the UNITA beacon just via the smartphone loudspeaker and microphone. Second, they can leave messages for other people which are only retrievable at the beacons place. Further, they can retrieve messages on-location.

5.2 User Study Results

The following section shows the results of the user study. The interpretation of the results, as well as the answering of the previously defined global question, follows in the chapter 6.

5.2.1 Demographics

The demographics of the tested users is shown in table 5.1:

	Number of Tested Users
Gender	
Female	6
Male	5
Age	
20-24	1
25-29	6
30-34	1
35-40	3
Profession	
Developers	4
Usability/UX Experts	4
Non-technical jobs	3

Table 5.1. Overview on the demographics of the tested users.

This distribution was chosen, because of getting information on different aspects of the implementation of SocialWall. Further, the smartphone usage of the tested users is shown in table 5.2. The main tasks performed on the smartphone are social media, email, messaging apps like Whatsapp, telephone calls, calendar, and internet in general. The question, if they already used a location-based service, was positively answered by 9 people and most frequently mentioned applications were the game Pokemon Go and a map app.

Smartphone Usage	Number of Tested Users
Never/I do not have one	0
Not daily	0
Less than 10 minutes	0
10 to 15 minutes	0
15 to 30 minutes	0
30 to 45 minutes	2
45 to 60 minutes	2
more than 61 minutes	7

Table 5.2. Overview on the smartphone usage of the tested users.

5.2.2 Ultrasonic Communication

Starting with the general part about ultrasonic communication of the user study, the results are as follows:

The question, if the users heard about ultrasonic communication before, was answered with yes by 8 people. Mostly, they heard about it from research projects or news articles and their knowledge about it is minimal. Whereas, only 4 users knew before that their smartphone can send and receive ultrasound messages. The number of active users is even smaller as only 1 person uses a ultrasonic firewall for blocking unwanted inaudible messages. The mean of liking the idea of sending data over inaudible sound is 4 and users mentioned the privacy of the transmission and the possible low cost feature of it. In addition, the question about the use of ultrasound to communicate with other devices was rated by a mean of 3.72. The users further commented that they would use it, if it is easy to use, safe, cheap, secure, makes something easier and offers a good alternative to an existing technology.

Furthermore, the handling needs to be easy to learn and it has to be somehow controllable as it is not hearable. 10 of 11 users already pair their devices with Bluetooth speakers and besides that, they use it for media boxes (5 of 11), entrances with NFC (3 of 11) and two users us NFC for mobile payment and ticketing. For the replacement with ultrasound, everyone would use it for pairing with loudspeakers, 7 would pair with their media box, 9 would use it for tickets and entrances and mobile payment were mentioned by some users under the condition that it is safe. Next, the usefulness and the satisfying scale were measured by the system acceptance scale from -2 to 2. The usefulness was rated with a mean of 1.04, which is already a good rating. Whereas, the satisfaction was rated with a mean of 0.53, which is positive but could be improved (Van Der Laan et al., 1997). The satisfaction scale is lower as more users think ultrasound a good alternative to existing technologies, though they are not sure if ultrasonic communication meets their expectations of it.

5.2.3 Usability of SocialWall

The following section shows the results of the proof-of-concept related questions.

After the user tests, the system usability scale (SUS) on the SocialWall application was performed. The SUS has a value range of 0 to 100 and interprets the like in following table 5.3:

SUS Score	Grade	Adjective Rating
>80.3	A	Excellent
68 – 80.3	B	Good
68	C	Okay
51 – 68	D	Poor
<51	F	Awful

Table 5.3. Overview on the smartphone usage of the tested users.

The result of the performed SUS, is a mean of 75.91 and stands for a B, which means good (UXUITrend, 2017).

Additionally, further questions on the use case were asked, starting with the question of hearing something unusual while interacting with the beacon. This was answered by 8 people with no, and only three users had either a slight pressure on their ears or heard a silent cracking noise one or two times. Next, the question, on how intuitive SocialWall is was answered by a mean of 4.27 and comments like the concept was intuitive, good icons were used, the setup is easy, and it is similar to existing messaging apps. But further, the need of an onboarding was mentioned as well as a sometimes misleading wording.

Following with the question, about how easy the application was to use, resulted in a mean rating of 4.36 and comments about the not overloaded interface and the fact that the process and the used symbols are already known. Further on, the question about convenience of the pairing process was rated with 3.82 and the main fact asked by the users was more feedback. This shows also the question about having problems during the tasks, which was mostly always missing feedback on sending and receiving. On the question for additional feedback, most users mentioned adding feedback for the sending and receiving process in form of a kind of loading animation symbol or a progress bar to be sure that the send of the message works or the listening still is active. Besides that, a LED for the beacon was requested, which should represent the active status of the beacon.

In addition, the question about how the users like the idea of such an ultrasonic black board resulted in a mean rating of 4.73. The test people mentioned that it is a funny idea and they would use it for example for communication with colleagues, for work, or like a kind of geocaching. An important aspect is the security and encryption of the sent messages. The question about the on-location aspect was admitted as a good idea, which makes it physical to the real world and boosts interaction to go to the beacon, though it still depends on the use case. The tested people further mentioned, that they would likely use it, if it is

set up on a public place. Only 4 of 11 users were not sure or would definitely not use it. Last, for general improvements, it should be more stable, the design needs to be updated, feedback has to be added and manual, how to use it, would be nice. Besides that, the LED status indication was noted again.

6 Discussion and Future Work

6.1 UNITA Beacon and Ultrasound Communication

The hardware research showed, that with low cost components and a single-board computer a beacon for ultrasonic communication can be assembled. The chosen Raspberry Pi 3 B was able to do real time audio processing and to keep the socket connection with the server. Though this was not the smallest, cheapest and most energy saving option, it was perfect for building a first stable version of the beacon. The other components cost in sum less than 20 euros and are in combination with a 20 euro power bank already runnable in theory. Though, the biggest point to tackle, is getting the amplifying for the microphone working. The tested amplifiers were either to strong and produced to much noise or were fine-grained enough, but did not deliver enough Bias voltage to run the condenser microphones. On the loudspeaker side, more detailed and sized tests have to be run. The current setup shows that they are able to send signals, but need further tests on stability and reliability on different settings. These two things have to be addressed to get a fully working beacon. The beacon itself, components inside as well as the case, can be further improved and possible extensions are mentioned in this chapter.

6.2 UNITA SDK and Server

The first version of the SDK is already a good base for developing an own application. It consists of many atomic features, which still are partly incomplete and can be extended. Many message types for various use cases are in and can be used and inherited. The conversion between lowest layer and the application on top is given and can be made more generic in terms of headers. At the moment, the message headers are rather static and operate on a byte level instead of bit-wise. In addition, one communication method beside ultrasound is set up, though more technologies can be added. Further, the extension of the open source ultrasound communication protocol offers a multipackage system and provides an important part of the system architecture. The server side, has a both the REST endpoints and the WebSocket listeners implemented. It receives and processes messages but still the processing needs further improvement on efficiency and expansion of the functionality. In addition, the storage of the messages and peers was developed independently and flexible on different message types.

6.3 User Study Interpretation: SocialWall

The implementation of the main use case SocialWall showed, that with the UNITA SDK a functional application with ultrasonic communication can be developed. The basic proof-of-concept app was a good base for user tests and already showed several interesting results. Overall, the result of the system usability scale of SocialWall resulted in a mean rating of 75.91 which is rated as B and already acceptable and good (UXUITrend, 2017). Further, answering the question Q5, listed in chapter 5.1.4, about how intuitive the application was, gave back a rating of 4.27 of 5 and a very intuitive setup, though the wording is still misleading sometimes. The application was rather easy-to-use as the result was a rating of 4.36 and only the pairing process underperformed a bit with 3.82. The users mentioned that the interface was not overloaded and the app interaction was similar to existing messaging applications. Several improvements were mentioned, whereas the biggest one is more and better feedback on sending and receiving messages, which answers the corresponding questions Q6 and Q7 about additional feedback and improvements,listed in chapter 5.1.4. Especially loading symbols and animations, and a kind of progress bar were mentioned. Further, LEDs for the beacon status were asked.

Last, the question Q4 about the idea of such a location-based ultrasound black board, listed in chapter 5.1.4, was answered by a rating of 4.73 of 5 for liking the idea. The users said, that it is a "funny" idea and they would be curious to test it out, especially for the on-location aspect as this boosts interaction with the beacon and the real physical world. Depending on the use case and how stable and secure it is most of the test persons would use it at public places. Overall the idea was well received.

6.4 User Study Interpretation: Ultrasonic Communication

The user tests then showed that many people already heard about ultrasound, though they mostly do not know how it works and which devices are able to perform ultrasonic communication. To answer the question Q2, if it would be a useful alternative for other technologies, listed in chapter 5.1.4, several things need to be given. People need more basic knowledge about ultrasound before using it, as the process for sending and receiving needed a special position of the smartphone. They have to know that ultrasound needs a line-of-sight to the speaker or microphone in comparison with Bluetooth or Wi-Fi. Holding it wrong or covering the device, could already lead to errors. Further, security and easy handling are important to meet the users requirements and expectations. This conditions also need to be met to give the users trust as the messages are not hearable. The system acceptance scale then confirmed the answer of question Q1, listed in chapter 5.1.4, as the ratings were rather positive on the usefulness part as well as on the satisfaction scale. When asking the users, for which use cases they would use ultrasound instead of Bluetooth and NFC, they mostly agreed on pairing with loudspeakers and media boxes, and using it for ultrasonic

tickets and opening entrances, which answers question Q3, listed in chapter 5.1.4. Mobile payment is also covered by this opinion, but only in a safe environment.

6.5 Future Topics and possible Extensions

During the development and user test phase, several extension ideas appeared. This extensions would apply for the beacon and the hardware, as well as for the UNITA SDK, server and use case application SocialWall.

First, a smaller beacon case with more compact assembled components could be created. The current version is relatively big, because of stability and testing purposes. Further, the beacon could be then made waterproof, to be set up outside without worrying about rain, snow or wind. Accompanying with that, instead of a Raspberry Pi 3 B a Raspberry Pi Zero could be used and tested to create an even smaller beacon. On the connection side, the previously described technology LoRaWAN would be an addition for communication with the server. Depending on the use case this could be a substitute for the network connection. This would require low data transmission and non-time critical exchange of information. A further advantage would be the resulting lower energy consumption. As currently, the beacon is relying on a local Wi-Fi, a GSM module would be an addition for more flexibility. It would need more power and a SIM card to work, but high data rates can be achieved.

On the UNITA SDK side, the static header can be made more flexible by making it dynamic by changing the length depending on one value. Furthermore, the header structure can be changed from byte-wise entries to bit-wise splitting. This means, header values which only need one or two bits, do not need to reserve a whole byte. This would allow to add more smaller header values, using one or two bits, and makes the header more meaningful and customizable.

Another extension would be the possibility of more than one recorder and sender at the same time on different frequencies. By enabling several recorders, different messages could be received in parallel. In addition, defining and measuring distances would create more possibilities for sending information on different levels. Besides that, the security side would need a separate controller called AuthenticationController for handling everything, which relates to encryption, authentication and key- and token-handling.

The server could be extended by adding logging for developers. A basis for writing logs to files and including that in the source code would be possible. Further, reading those logs and generating statistics and evaluations is an extension. Additionally, a full admin panel for maintaining the applications and beacons would be a simplification for the developers. With having this in mind, the maintenance of different applications on one server needs to be extended. Besides that, a management option for commands via the server needs to be created.

The use case application can be extended by more feedback mechanisms. For further development of SocialWall, commonly used software parts of the client and beacon application could be refactored into a base module. This would be easier to maintain and extend. Both applications would then extend from the module. On the private messaging part, more local message options for retrieving data can be implemented. Last, more use cases can be implemented, to create different applications for different purposes using ultrasonic communication.

7 Conclusion

The goal of UNITA was to build hardware and software for creating an ultrasonic beacon representing a novel communication endpoint. The beacon should be easily reproducible and keep the costs and energy consumption at a minimum, whereas, the software should give a good basis for other developers, who want to develop applications using ultrasonic communication. At the moment, there is no open source SDK for implementing IoT applications using ultrasound. Additionally, there is no open source beacon specialized development kit for ultrasonic communication available. The master thesis should answer the question of how an ultrasonic beacon could be developed with simple hardware and to which degree users accept and trust the beacon and its communication.

To solve this problem, a hardware beacon, a SDK and a server basis were created, In addition, a proof-of-concept application called SocialWall, was implemented using the SDK. The SDK has implemented basic features for most use case applications, which were gathered during the creating of UNITA. On the beacon side there are still open points, but it represents already a good base for finalizing it to a fully functional version. SocialWall, further proofed while developing that the SDK is already well-designed.

The SDK is published as open source software and can be used and further developed by the community. The application SocialWall is also released under open source and gives future developers an insight on how to create their own application.

UNITA shows that with the SDK and open source hardware, applications using ultrasonic communication are easy to set up. Developers can build upon the UNITA SDK and run a server instance. User tests showed that most people know about ultrasound, whereas they only know it exists and often not more. Results showed, that the additional communication channel is well received and would be used for several purposes. An important factor is the security aspect, which needs to be addressed in a good way and further raises awareness on what ultrasound can do or not. Additionally, the most important factor was the missing feedback mentioned during the tests. Approaching this facets of ultrasonic communication, many rich future applications can be created.

Still missing in the implementation of the SDK are security features through the ultrasound channel. Further, the interaction model with smartphones and beacons needs to be retested and changed. This could lead to a set of rules about how to create the interaction and the interface of an inaudible communication. The beacon offers also a reference point for improvement on cost, size, performance, waterproofness, weather independence, and server connection setup. For future projects, the field of Industry 4.0 could be specifically

addressed and applications based on their needs could be created. Larger settings with more beacons and users can be evaluated and new features and extensions can be implemented. This shows that UNITA has a big potential for the future and can be extended and improved into many directions.

UNITA should boost the development of ultrasonic communication applications and increase the size of the ultrasound developer community. Developers can develop their own useful applications with the SDK and can further extend the SDK itself. UNITA is fully open source under the GNU General Public License Version 3 (GPL 3) and available online on the website unita.fhstp.ac.at.

Bibliography

Alliance, L. (2015). LoRaWAN™ What is it? Technical report.

Alzahrani, S. M. (2017). Sensing for the Internet of Things and Its Applications. In *2017 5th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 88–92, Prague. IEEE.

Arentz, W. A. and Bandara, U. (2011). Near ultrasonic directional data transfer for modern smartphones. page 481. ACM Press.

Arp, D., Quiring, E., Wressnegger, C., and Rieck, K. (2017). Privacy Threats through Ultrasonic Side Channels on Mobile Devices.

Bisio, I., Delfino, A., Grattarola, A., Lavagetto, F., and Sciarrone, A. (2018). Ultrasounds-Based Context Sensing Method and Applications Over the Internet of Things. *IEEE Internet of Things Journal*, 5(5):3876–3890.

Bluetooth SIG, I. (2013). Bluetooth Low Energy Technology | Bluetooth Technology Website.

Carotenuto, R., Merenda, M., Iero, D., and Corte, F. G. D. (2018). Ranging RFID Tags With Ultrasound. *IEEE Sensors Journal*, 18(7):2967–2975.

CopSonic (2019). CopSonic - Ultrasonic authentication.

Deshotels, L. (2014). Inaudible Sound As a Covert Channel in Mobile Devices. In *Proceedings of the 8th USENIX Conference on Offensive Technologies*, WOOT'14, pages 16–16, Berkeley, CA, USA. USENIX Association. event-place: San Diego, CA.

Eurotech (2010). MQTT V3.1 Protocol Specification.

Fette, I. and Melnikov, A. (2011). The WebSocket Protocol. Technical report.

Fielding, R. T. (2000). Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST).

Getreuer, P., Gnegy, C., Lyon, R. F., and Saurous, R. A. (2018). Ultrasonic Communication Using Consumer Hardware. *IEEE Transactions on Multimedia*, 20(6):1277–1290.

Hanspach, M. and Goetz, M. (2013). On Covert Acoustical Mesh Networks in Air. *Journal of Communications*, 8(11):758–767.

Hosman, T., Yeary, M., Antonio, J. K., and Hobbs, B. (2010). Multi-tone FSK for ultrasonic communication. pages 1424–1429. IEEE.

ITU (1994a). Information technology - Open Systems Interconnection - Application layer structure - ITU-T Recommendation X.207. Technical report.

ITU (1994b). Information technology - Open Systems Interconnection - Basic Reference Model: Conventions for the definition of OSI services - ITU-T Recommendation X.210. Technical report.

ITU (1994c). Information technology - Open Systems Interconnection - Presentation service definition - Recommendation X.216. Technical report.

ITU (1996a). Information technology - Open Systems Interconnection - Physical service definition - ITU-T Recommendation X.211. Technical report.

ITU (1996b). Information technology - Open Systems Interconnection - Session service definition - Recommendation X.215. Technical report.

ITU (1996c). Information technology - Open Systems Interconnection - Transport service definition - ITU-T Recommendation X.214. Technical report.

ITU (1997). Information technology - Open Systems Interconnection - Data Link service definition - ITU-T Recommendation X.212. Technical report.

ITU (2002). Information technology - Open Systems Interconnection - Network service definition - ITU-T Recommendation X.213. Technical report.

Ka, S., Kim, T. H., Ha, J. Y., Lim, S. H., Shin, S. C., Choi, J. W., Kwak, C., and Choi, S. (2016). Near-ultrasound communication for TV's 2nd screen services. pages 42–54. ACM Press.

Lazik, P., Rajagopal, N., Shih, O., Sinopoli, B., and Rowe, A. (2015). ALPS: A Bluetooth and Ultrasound Platform for Mapping and Localization. pages 73–84. ACM Press.

Lazik, P. and Rowe, A. (2012). Indoor pseudo-ranging of mobile devices using ultrasonic chirps. page 99. ACM Press.

Legendre, F. (2015). How Google Nearby (really) works – and what else it does?

Li, C., Hutchins, D., and Green, R. (2008). Short-range ultrasonic digital communications in air. *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, 55(4):908–918.

Lin, M.-C., Huang, F.-Y., and Chiueh, T.-D. (2015). A-NFC: Two-way near-field communications (NFC) via inaudible acoustics. pages 1–6. IEEE.

Lisnr (2019). Welcome to LISNR, the data-over-sound leader.

Lopes, C. and Aguiar, P. (2003). Acoustic modems for ubiquitous computing. *IEEE Pervasive Computing*, 2(3):62–71.

Madhavapeddy, A., Scott, D., Tse, A., and Sharp, R. (2005). Audio Networking: The Forgotten Wireless Technology. *IEEE Pervasive Computing*, 4(3):55–60.

Mavroudis, V., Hao, S., Fratantonio, Y., Maggi, F., Kruegel, C., and Vigna, G. (2017). On the Privacy and Security of the Ultrasound Ecosystem. *Proceedings on Privacy Enhancing Technologies*, 2017(2).

Mitchell, B. (2019). 802.11 WiFi Standards Explained.

Murata, S., Yara, C., Kaneta, K., Ioroi, S., and Tanaka, H. (2014). Accurate Indoor Positioning System Using Near-Ultrasonic Sound from a Smartphone. pages 13–18. IEEE.

NearBytes (2019). NearBytes | Contactless Communication Technology.

Nosowitz, D. (2011). Everything You Need to Know About Near Field Communication.

Ortega, A. A., Bettachini, V. A., Fierens, P. I., and Alvarez-Hamelin, J. I. (2014). Encrypted CDMA Audio Network. *Journal of Information Security*, 05(03):73–82.

Santagati, G. E. and Melodia, T. (2017). A Software-Defined Ultrasonic Networking Framework for Wearable Devices. *IEEE/ACM Transactions on Networking*, 25(2):960–973.

Schiller, J. (2001). *Mobilkommunikation: Techniken für das allgegenwärtige Internet*. Net.com. Addison-Wesley, München. OCLC: 247974797.

Shelby, Z., Hartke, K., and Bormann, C. (2014). The Constrained Application Protocol (CoAP).

Sonarax (2019). Sonarax | Data-Over-Sound and Location Based Services.

Spectrum, I. (2016). 802.11-2016 - IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

Spectrum, I. (2017). Everything You Need To Know About 5g.

Sun, D., Wei, D., Zhang, N., Lv, Z., and Yin, X. (2016). Network transmission of hidden data using smartphones based on compromising emanations. In *2016 Asia-Pacific International Symposium on Electromagnetic Compatibility (APEMC)*, volume 01, pages 190–193.

Thiel, B., Kloch, K., and Lukowicz, P. (2012). Sound-based proximity detection with mobile phones. pages 1–4. ACM Press.

UXUITrend (2017). Measuring and Interpreting System Usability Scale (SUS).

Van Der Laan, J. D., Heino, A., and De Waard, D. (1997). A simple procedure for the assessment of acceptance of advanced transport telematics. *Transportation Research Part C: Emerging Technologies*, 5(1):1–10.

Vinoski, S. (2006). Advanced Message Queuing Protocol. *IEEE Internet Computing*, 10(6):87–89.

Wang, Q., Ren, K., Zhou, M., Lei, T., Koutsonikolas, D., and Su, L. (2016). Messages behind the sound: real-time hidden acoustic signal capture with smartphones. pages 29–41. ACM Press.

Webopedia (2008). How do Wireless Networks Work?

Yan, H., Zhou, S., Shi, Z. J., and Li, B. (2007). A DSP implementation of OFDM acoustic modem. page 89. ACM Press.

Zeppelzauer, M. and Ringot, A. (2019). SoniTalk: An Open Protocol for Data-Over-Sound Communication. Technical report.

Zeppelzauer, M., Ringot, A., and Taurer, F. (2018). SoniControl - A Mobile Ultrasonic Firewall. In *2018 ACM Multimedia Conference on Multimedia Conference - MM '18*, pages 1250–1252. arXiv: 1807.07617.

Zhang, B., Zhan, Q., Chen, S., Li, M., Ren, K., Wang, C., and Ma, D. (2014). PriWhisper: Enabling Keyless Secure Acoustic Communication for Smartphones. *IEEE Internet of Things Journal*, 1(1):33–45.

Zhang, G., Yan, C., Ji, X., Zhang, T., Zhang, T., and Xu, W. (2017). DolphinAttack: Inaudible Voice Commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 103–117, New York, NY, USA. ACM.

List of Figures

List of Tables

Listings

Appendices

A Class Diagrams - UNITA SDK

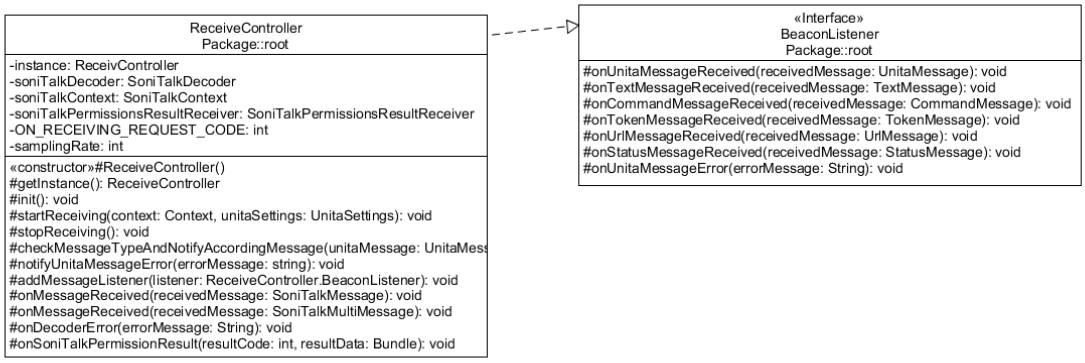

Figure 7.1. Class diagram of the ReceiveController of the UNITA SDK.

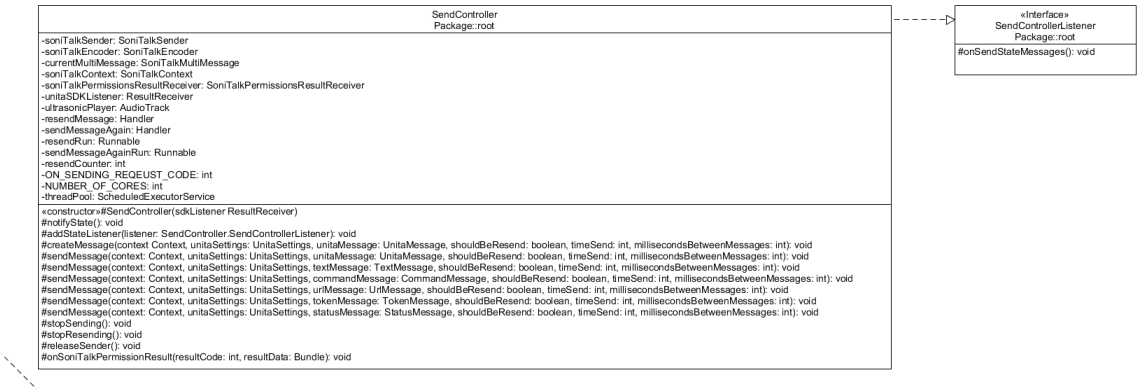

Figure 7.2. Class diagram of the SendController of the UNITA SDK.

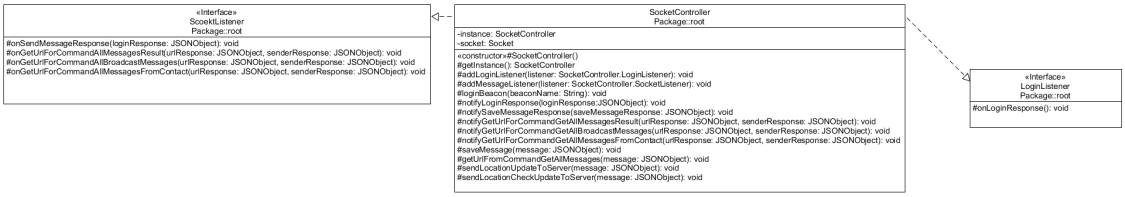

Figure 7.3. Class diagram of the SocketController of the UNITA SDK.

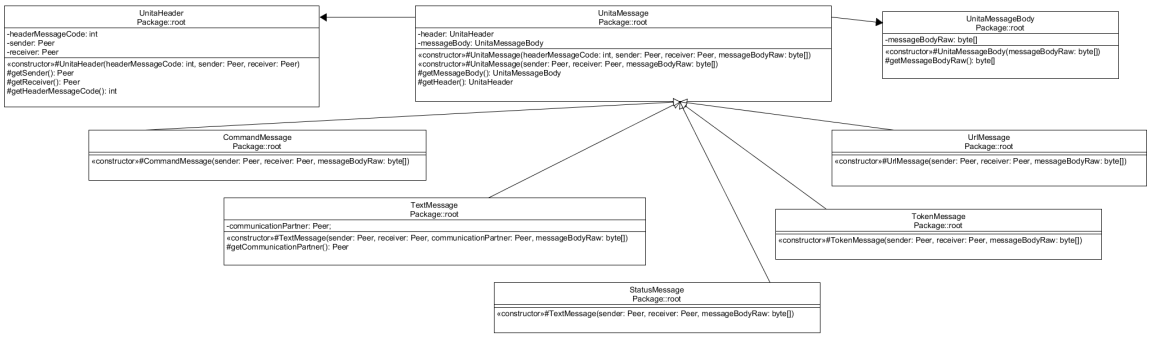

Figure 7.4. Class diagram of the message types of the UNITA SDK.

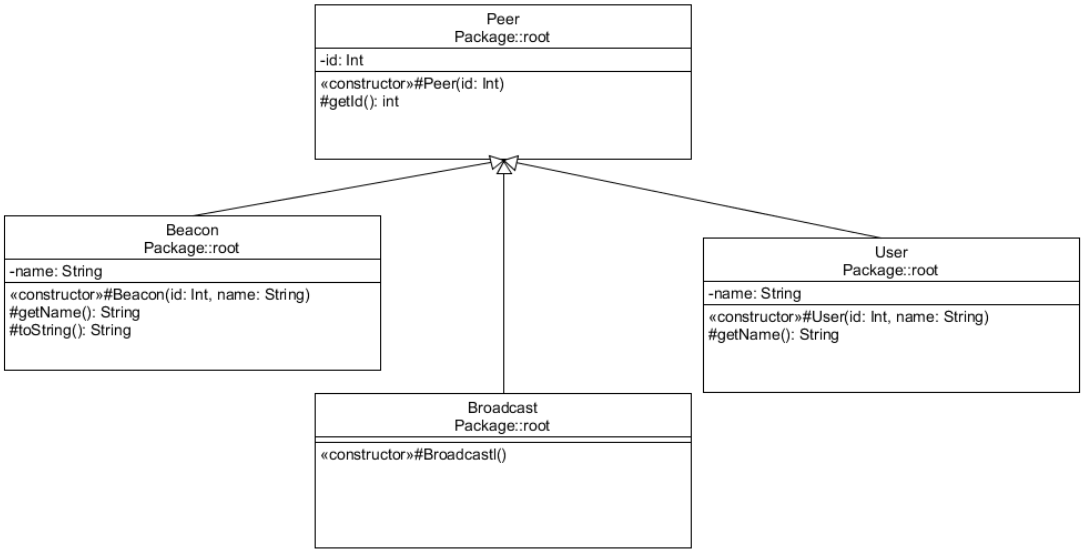

Figure 7.5. Class diagram of the peer types of the UNITA SDK.

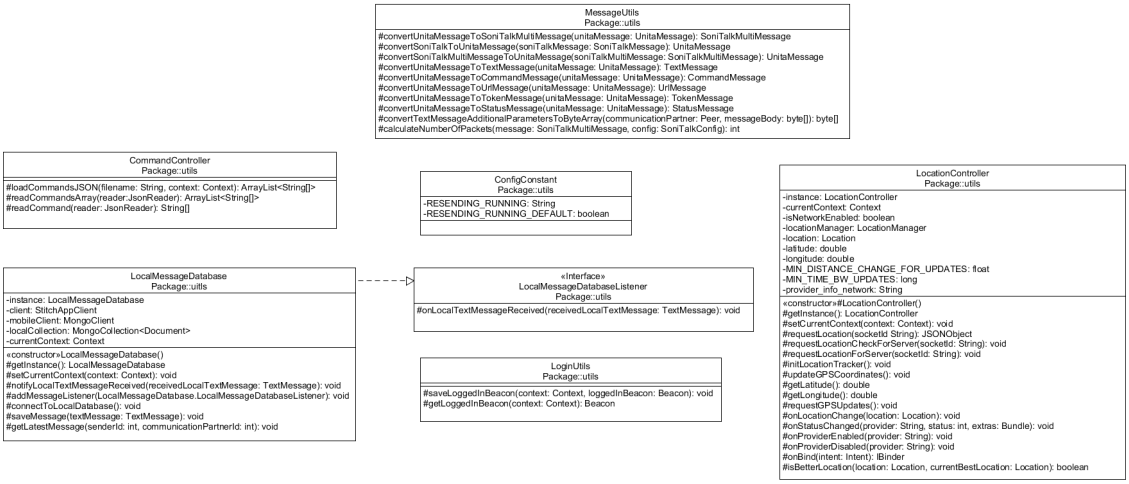

Figure 7.6. Class diagram of the remaining controller classes and the utility classes of the UNITA SDK.

B Source Code Snippets

Listing 7.1. SoniTalkMultiMessage constructor

```
1 public SoniTalkHeader(byte messageId, byte packetId, byte
    ↪ numberOfPackets){
2     this.messageId = messageId;
3     this.packetId = packetId;
4     this.numberOfPackets = numberOfPackets;
5 }
```

Listing 7.2. Calling the calculation for number of packets.

```
1 final int nTimes = EncoderUtils.calculateNumberOfPackets(message,
    ↪ config);
2 currentMultipleAudioTrack = new AudioTrack[nTimes];
```

Listing 7.3. Calculating the number of packets.

```
1 public static int calculateNumberOfPackets(SoniTalkMultiMessage
    ↪ message, SoniTalkConfig config){
2     int numOfBytes = config.getnMessageBlocks()*(config.
        ↪ getnFrequencies()/8)-2;
3     int fixedHeaderSize = 3;
4     final byte[] bytes = message.getMessage();
5     byte[] headerBytesPlaceholder = new byte[fixedHeaderSize];
6     byte[] checkSizeByte = ArrayUtils.addAll(bytes,
        ↪ headerBytesPlaceholder);
7     if(EncoderUtils.isAllowedByteArraySize(checkSizeByte, config))
        ↪ {
8         return 1;
9     } else{
10        if(bytes.length%(numOfBytes-fixedHeaderSize)==0){
11            return bytes.length/(numOfBytes-fixedHeaderSize);
12        }else{
13            return bytes.length/(numOfBytes-fixedHeaderSize)+1;
14        }
15    }
16 }
```

Listing 7.4. Calling the multi message split method.

```
final SoniTalkMessage[] soniTalkMessages = EncoderUtils.
    ↪ splitMultiMessageIntoSoniTalkMessages(message, nTimes,
    ↪ config);
```

Listing 7.5. Splitting a multi message into several single messages.

```
public static SoniTalkMessage[]
    ↪ splitMultiMessageIntoSoniTalkMessages(SoniTalkMultiMessage
    ↪ message, int numberOfPackets, SoniTalkConfig config){
    SoniTalkMessage[] soniTalkMessages = new SoniTalkMessage[
        ↪ numberOfPackets];
    int numOfBytes = config.getnMessageBlocks()*(config.
        ↪ getnFrequencies()/8)-2;
    int fixedHeaderSize = 3;
    int messageId = (int) (255*Math.abs(Math.random()));
    int packetId = 1;
    SoniTalkHeader soniTalkHeader;
    for(int i = 0; i<numberOfPackets ;i++){
        byte[] messageBodyPart;
        if((i+1)!=numberOfPackets){
            int messageLength = numOfBytes-fixedHeaderSize;
            messageBodyPart = new byte[messageLength];
            System.arraycopy(message.getMessage(), messageLength*i
                ↪ , messageBodyPart,0, messageLength);
        }else{
            int restMessageLength = message.getMessage().length-((
                ↪ numOfBytes-fixedHeaderSize)*(numberOfPackets-1)
                ↪ );
            messageBodyPart = new byte[restMessageLength];
            System.arraycopy(message.getMessage(), (numOfBytes-
                ↪ fixedHeaderSize)*(numberOfPackets-1),
                ↪ messageBodyPart,0, restMessageLength);
        }
        soniTalkHeader = new SoniTalkHeader(EncoderUtils.
            ↪ intToByteArray(messageId), EncoderUtils.
            ↪ intToByteArray(packetId), EncoderUtils.
            ↪ intToByteArray(numberOfPackets));
        soniTalkMessages[i] = new SoniTalkMessage(messageBodyPart,
            ↪  soniTalkHeader);
        packetId++;
    }
```

```
23
24      return soniTalkMessages;
25  }
```

Listing 7.6. Filling the arrays with short audio data.

```
1 soniTalkMessages[i] = soniTalkEncoder.generateMessage(
    ↪ soniTalkMessages[i].getMessage(), soniTalkMessages[i].
    ↪ getSoniTalkHeader());
```

Listing 7.7. Decoding of the received single SoniTalkMessage and saving or forwarding them depending on the number of packets.

```
1  if (parityCheckResult == 0) {
2      byte[] numberOfPackets = new byte[1];
3      System.arraycopy(receivedMessage, 2, numberOfPackets, 0, 1);
4      byte[] messageId = new byte[1];
5      byte[] packetId = new byte[1];
6      System.arraycopy(receivedMessage, 0, messageId, 0, 1);
7      System.arraycopy(receivedMessage, 1, packetId, 0, 1);
8      SoniTalkHeader soniTalkHeader = new SoniTalkHeader(messageId
           ↪ [0], packetId[0], numberOfPackets[0]);
9      byte[] receivedMessageBody = new byte[receivedMessage.length -
           ↪  3];
10     System.arraycopy(receivedMessage, 3, receivedMessageBody, 0,
           ↪ receivedMessage.length - 3);
11
12     if ((int) numberOfPackets[0] > 1) {
13         SoniTalkMessage soniTalkMessage = new SoniTalkMessage(
               ↪ receivedMessageBody, soniTalkHeader);
14
15         if (!soniTalkMessageReceiveOverviewDictionary.containsKey(
               ↪ String.valueOf((int) messageId[0]))) {
16             soniTalkMessageReceiveOverviewDictionary.put(String.
                   ↪ valueOf((int) messageId[0]), new TreeMap<String
                   ↪ , SoniTalkMessage>());
17             Map<String, SoniTalkMessage>
                   ↪ soniTalkMessageReceiveDictionary =
                   ↪ soniTalkMessageReceiveOverviewDictionary.get(
                   ↪ String.valueOf((int) messageId[0]));
18             if (!soniTalkMessageReceiveDictionary.containsKey(
                   ↪ String.valueOf((int) packetId[0]))) {
```

```
19                  soniTalkMessageReceiveDictionary.put(String.
                        ↪ valueOf((int) packetId[0]), soniTalkMessage
                        ↪ );
20              }
21          } else {
22              Map<String, SoniTalkMessage>
                    ↪ soniTalkMessageReceiveDictionary =
                    ↪ soniTalkMessageReceiveOverviewDictionary.get(
                    ↪ String.valueOf((int) messageId[0]));
23              if (!soniTalkMessageReceiveDictionary.containsKey(
                    ↪ String.valueOf((int) packetId[0]))) {
24                  soniTalkMessageReceiveDictionary.put(String.
                        ↪ valueOf((int) packetId[0]), soniTalkMessage
                        ↪ );
25              }
26              if (soniTalkMessageReceiveDictionary.size() == (int)
                    ↪ numberOfPackets[0]) {
27                  ArrayList<SoniTalkMessage> receivedMessages = new
                        ↪ ArrayList<>();
28                  Iterator it = soniTalkMessageReceiveDictionary.
                        ↪ entrySet().iterator();
29                  while (it.hasNext()) {
30                      Map.Entry pair = (Map.Entry) it.next();
31                      receivedMessages.add((SoniTalkMessage) pair.
                            ↪ getValue());
32                  }
33                  soniTalkMessageReceiveOverviewDictionary.remove(
                        ↪ String.valueOf((int) messageId[0]));
34
35                  notifyMessageListeners(DecoderUtils.
                        ↪ concatenateMessages(receivedMessages));
36              }
37          }
38      } else {
39          SoniTalkMultiMessage soniTalkMultiMessage = new
                ↪ SoniTalkMultiMessage(receivedMessageBody);
40          notifyMessageListeners(soniTalkMultiMessage);
41      }
42  } else {
```

```
43      byte[] receivedMessageBody = new byte[receivedMessage.length -
            ↪ 3];
44      SoniTalkMultiMessage soniTalkMultiMessage = new
            ↪ SoniTalkMultiMessage(receivedMessageBody);
45      soniTalkMultiMessage.setCrcIsCorrect(false);
46      notifyMessageListeners(soniTalkMultiMessage);
47      Log.d("SoniTalkDecoder", "crc␣incorrect");
48  }
```

Listing 7.8. Instantiating the permission receiver.

```
1  public void init(ResultReceiver sdkListener){
2      if (instance != null) {
3          soniTalkPermissionsResultReceiver = new
               ↪ SoniTalkPermissionsResultReceiver(new Handler());
4          soniTalkPermissionsResultReceiver.setReceiver(this);
5          this.unitaSdkListener = sdkListener;
6      }
7  }
8
9  @Override
10 public void onSoniTalkPermissionResult(int resultCode, Bundle
       ↪ resultData) {
11     unitaSdkListener.send(resultCode, resultData);
12 }
```

Listing 7.9. Implementing interface of SoniTalkDecoder.

```
1  @Override
2  public void onMessageReceived(final SoniTalkMultiMessage
       ↪ receivedMessage){
3      if(receivedMessage.isCrcCorrect()) {
4          UnitaMessage unitaMessage = MessageUtils.
               ↪ convertSoniTalkMultiMessageToUnitaMessage(
               ↪ receivedMessage);
5          checkMessageTypeAndNotifyAccordingMessage(unitaMessage);
6      } else {
7      }
8  }
9
10 @Override
11 public void onDecoderError(String errorMessage) {
```

```
12      notifyUnitaMessageError(errorMessage);
13  }
```

Listing 7.10. Checking for the message type.

```
1  public void checkMessageTypeAndNotifyAccordingMessage(UnitaMessage
       ↪ unitaMessage){
2      for(BeaconListener listener: beaconListeners) {
3          switch (unitaMessage.getHeader().getHeaderMessageCode()){
4              case 0:
5                  listener.onUnitaMessageReceived(unitaMessage);
6                  break;
7              case 1:
8                  TextMessage textMessage = MessageUtils.
                       ↪ convertUnitaMessageToTextMessage(
                       ↪ unitaMessage);
9                  listener.onTextMessageReceived(textMessage);
10                 break;
11             case 2:
12                 CommandMessage commandMessage = MessageUtils.
                       ↪ convertUnitaMessageToCommandMessage(
                       ↪ unitaMessage);
13                 listener.onCommandMessageReceived(commandMessage);
14                 break;
15             case 3:
16                 UrlMessage urlMessage = MessageUtils.
                       ↪ convertUnitaMessageToUrlMessage(
                       ↪ unitaMessage);
17                 listener.onUrlMessageReceived(urlMessage);
18                 break;
19             case 4:
20                 TokenMessage tokenMessage = MessageUtils.
                       ↪ convertUnitaMessageToTokenMessage(
                       ↪ unitaMessage);
21                 listener.onTokenMessageReceived(tokenMessage);
22                 break;
23             case 5:
24                 StatusMessage statusMessage = MessageUtils.
                       ↪ convertUnitaMessageToStatusMessage(
                       ↪ unitaMessage);
25                 listener.onStatusMessageReceived(statusMessage);
```

```
26                  break;
27              default:
28                  listener.onUnitaMessageReceived(unitaMessage);
29                  break;
30          }
31      }
32  }
```

Listing 7.11. Add function to get access to the message listeners.

```
1  public void addMessageListener(ReceiveController.BeaconListener
       ↪ listener) {
2      this.beaconListeners.add(listener);
3  }
```

Listing 7.12. Conversion call for UnitaMessages to SoniTalkMultiMessages.

```
1  currentMultiMessage = MessageUtils.
       ↪ convertUnitaMessageToSoniTalkMultiMessage(unitaMessage);
```

Listing 7.13. Routine for sending messages and resending them after specific time.

```
1  resendRun = new Runnable() {
2      @Override
3      public void run() {
4          SharedPreferences sp = PreferenceManager.
               ↪ getDefaultSharedPreferences(context);
5          boolean stillResending = sp.getBoolean(ConfigConstants.
               ↪ RESENDING_RUNNING, ConfigConstants.
               ↪ RESENDING_RUNNING_DEFAULT);
6          if(shouldBeResend) {
7              if (stillResending) {
8                  if (resendCounter > 0) {
9                      sendMessageAgain.removeCallbacks(
                           ↪ sendMessageAgainRun);
10                     soniTalkSender.send(currentMultiMessage,
                           ↪ soniTalkSenderInterval, TimeUnit.
                           ↪ MILLISECONDS, ON_SENDING_REQUEST_CODE,
                           ↪ unitaSettings, soniTalkEncoder);
11                     resendCounter--;
12                     resendMessage.postDelayed(resendRun, delayTime
                           ↪ );
```

```
13                  }else{
14                      sendMessageAgain.postDelayed(
                            ↪ sendMessageAgainRun, delayTime/2);
15                  }
16              } else {
17                  sp.edit().putBoolean(ConfigConstants.
                        ↪ RESENDING_RUNNING, true).commit();
18                  stopResending();
19              }
20          }
21      }
22  };
23  resendMessage.postDelayed(resendRun, delayTime);
24
25  soniTalkSender.send(currentMultiMessage, soniTalkSenderInterval,
        ↪ TimeUnit.MILLISECONDS, ON_SENDING_REQUEST_CODE,
        ↪ unitaSettings, soniTalkEncoder);
26  resendCounter--;
```

Listing 7.14. Socket event for logging in a beacon.

```
1  public void loginBeacon(String beaconName){
2      JSONObject jsonObject = new JSONObject();
3      try {
4          jsonObject.put("name", beaconName);
5      } catch (JSONException e) {
6          e.printStackTrace();
7      }
8      socket.emit("loginBeacon", jsonObject);
9
10     socket.on("loginBeaconResult", new Emitter.Listener() {
11         @Override
12         public void call(Object... args) {
13             JSONArray jArray = (JSONArray) args[0];
14             JSONObject loginResponse = null;
15             try {
16                 loginResponse = jArray.getJSONObject(0);
17             } catch (JSONException e) {
18                 e.printStackTrace();
19             }
20             notifyLoginResponse(loginResponse);
```

```
21
22              socket.off("loginBeaconResult", this);
23          }
24      });
25  }
```

Listing 7.15. Login utility for keeping the actual beacon in SharedPreferences.

```
1  public static void saveLoggedInBeacon(Context context, Beacon
       ↪ loggedInBeacon){
2      SharedPreferences sp = PreferenceManager.
           ↪ getDefaultSharedPreferences(context);
3      SharedPreferences.Editor prefsEditor = sp.edit();
4      Gson gson = new Gson();
5      String json = gson.toJson(loggedInBeacon);
6      prefsEditor.putString("LoggedInBeacon", json);
7      prefsEditor.apply();
8      prefsEditor.commit();
9  }
10
11 public static Beacon getLoggedInBeacon(Context context){
12     Gson gson = new Gson();
13     SharedPreferences sp = PreferenceManager.
           ↪ getDefaultSharedPreferences(context);
14     String json = sp.getString("LoggedInBeacon", "");
15     return gson.fromJson(json, Beacon.class);
16 }
```

Listing 7.16. Conversion of UnitaMessages to TextMessage objects.

```
1  public static TextMessage convertUnitaMessageToTextMessage(
       ↪ UnitaMessage unitaMessage){
2      byte[] communicationPartnerIdArray = new byte[1];
3      System.arraycopy(unitaMessage.getMessageBody().
           ↪ getMessageBodyRaw(), 0, communicationPartnerIdArray,0,
           ↪ 1);
4      byte[] messageBodyArray = new byte[unitaMessage.getMessageBody
           ↪ ().getMessageBodyRaw().length-1];
5      System.arraycopy(unitaMessage.getMessageBody().
           ↪ getMessageBodyRaw(), 1, messageBodyArray,0,
           ↪ unitaMessage.getMessageBody().getMessageBodyRaw().
           ↪ length-1);
```

```
6
7       int communicationPartnerId = communicationPartnerIdArray[0] &
            ↪ (0xff);
8       Peer communicationPartner = new Peer(communicationPartnerId);
9
10      return new TextMessage(unitaMessage.getHeader().getSender(),
            ↪ unitaMessage.getHeader().getReceiver(),
            ↪ communicationPartner, messageBodyArray);
11  }
```

Listing 7.17. Connection function for MongoDB database.

```
1  private connectDb = () => {
2          return mongoose.connect(process.env.DATABASE_URL, {
               ↪ useNewUrlParser: true}, function(err) {
3                  if (err) { throw err; }
4                  const dataFactory = DataFactory.getInstance();
5                  dataFactory.createData();
6          });
7  }
```

Listing 7.18. Server initialization.

```
1 this.server = new http.Server(this.app);
2 this.database = Connection.getInstance();
3 this.socket = new Socket(this.server);
4 this.peerController = new PeerController();
5 this.peerTypeController = new PeerTypeController();
6 this.messageController = new MessageController();
```

Listing 7.19. Connect to database call and directory creation for possible uploads.

```
1 this.database.connectDb().then(async (connection) => {
2         this.server.listen(process.env.PORT, () => {
3                 const dir = process.env.ROOT + process.env.
                    ↪ NODE_PATH + "/public/uploads/";
4                 mkdirp(dir, function(err) {
5                 if (err) {
6                         console.error(err);
7                 } else {
8                         console.log("Directory␣created!");
9                 }
```

```
10          });
11          console.log("Server runs on Port: " + process.env.PORT + "
                ↪  " + process.env.ROOT);
12          });
13  });
```

Listing 7.20. REST endpoint for map visualization.

```
1  this.app.get("/", function(req, res) {
2          res.sendFile(process.env.NODE_PATH + "/index.html", { root
               ↪  : process.env.ROOT});
3  });
4  this.app.get("/js/*", function(req, res) {
5          let jspth = req.url;
6          let js = process.env.NODE_PATH + jspth;
7          res.sendFile(js, { root : process.env.ROOT});
8  });
9  this.app.get("/assets/icons/*", function(req, res, path) {
10         let iconpth = req.url;
11         let icon = process.env.NODE_PATH + iconpth;
12         res.sendFile(icon, { root : process.env.ROOT});
13 });
```

Listing 7.21. REST endpoint for getting all messages.

```
1  this.app.get(RestInterfaceConfig.getAllMessages + "/:id", async (
       ↪  req, res) => {
2          if (req.params.id !== undefined || req.params.id !== null)
               ↪  {
3                  this.messageController.
                       ↪  getAllMessagesAddressedToUserAnd
4                          BroadcastedMessages(req.params.id)
5                  .then((messages) => {
6                  res.json(messages);
7          });
8          } else {
9                  res.json({status: 1, message: "Get messages failed
                       ↪  "});
10         }
11 });
```

Listing 7.22. Getter for all public messages and private addressed ones.

```
1 public getAllMessagesAddressedToUserAndBroadcastedMessages(userId)
    ↪ : any {
2         return models.Message.find({$or: [{$or: [{sender: userId},
              ↪ {communicationPartner: userId}]},
3         {receiver: 0}]}).then((messages) => {
4                 return messages;
5         });
6 }
```

Listing 7.23. Saving routine for incoming messages.

```
1  public saveMessageToDB(messageData): any {
2          let additionalData = {};
3          let communicationPartner = {};
4          for (let key in messageData) {
5                  if (key !== "headerMessageCode" && key !== "header
                       ↪ " &&
6                          key !== "communicationPartnerUser" && key
                              ↪ !== "communicationPartnerBroadcast"
                              ↪  &&
7                          key !== "messageBody" && key !== "
                              ↪ messageBodyRaw") {
8                          additionalData[key] = messageData[key];
9                  }
10                 if (key === "communicationPartnerUser") {
11                         additionalData["communicationPartner"] =
                               ↪ messageData[key].id;
12                 }
13                 if (key === "communicationPartnerBroadcast") {
14                         additionalData["communicationPartner"] =
                               ↪ messageData[key].id;
15                 }
16         }
17         const message = new models.Message({
18                 headerMessageCode: messageData.header.
                       ↪ headerMessageCode,
19                 sender: messageData.header.sender.id,
20                 receiver: messageData.header.receiver.id,
21                 message: messageData.messageBody.messageBody,
```

```
22                  messageRaw: messageData.messageBody.messageBodyRaw
                        ↪ ,
23                  additionalData
24          });
25          return message.save().then((savedMessage) => {
26                  return [savedMessage];
27          });
28  }
```

Listing 7.24. Handling the login of beacons.

```
1  public async loginBeacon(beaconData): Promise<any> {
2          let additionalData = {};
3          for (let key in beaconData) {
4                  if (key !== "name") {
5                          additionalData[key] = beaconData[key];
6                  }
7          }
8          return models.Peer.countDocuments({name: beaconData.name})
               ↪ .then( async (count) => {
9                  if (count === 0) {
10                         let beacon = new models.Peer({
11                                 name: beaconData.name,
12                                 type: 1,
13                                 additionalData
14                         });
15                         return await beacon.save().then((
                               ↪ savedBeacon) => {
16                                 return [savedBeacon];
17                         });
18                 } else {
19                         return await models.Peer.find({$and: [{
                               ↪ name: beaconData.name}, {type:
                               ↪  1}]}).then((beacon) => {
20                                 return beacon;
21                         });
22                 }
23         });
24 }
```

Listing 7.25. Event for checking location and state of beacons.

```
setInterval(() => {
        this.locationController.getSocketConnections().then((
            ↪ socketConnections) => {
                for (let socketConnection in socketConnections) {
                        if (this.socket.sockets.sockets[
                            ↪ socketConnections[socketConnection
                            ↪ ].socketId] !== undefined) {
                                this.socket.to(socketConnections[
                                    ↪ socketConnection].socketId)
                                    ↪ .emit("requestLocationCheck
                                    ↪ ",
                                {socketId: socketConnections[
                                    ↪ socketConnection].socketId
                                    ↪ });
                        } else {
                                this.locationController.
                                    ↪ deleteSocketConnection({
                                socketId: socketConnections[
                                    ↪ socketConnection].socketId
                                    ↪ });
                        }
                }
        });
}, 1000 * 60 * 5);
```

Listing 7.26. Creating a new WebSocket instance.

```
this.socket = new IO(server);
```

Listing 7.27. Global socket listener for connection.

```
this.socket.on("connection", (socket) => {

});
```

Listing 7.28. Initialization of beacon application.

```
socket = SocketController.getInstance();
socket.addLoginListener(this);
socket.addMessageListener(this);
socket.loginBeacon(createBeaconName());

```

```
6  unitaPermissionsResultReceiver = new
       ↪ UnitaPermissionsResultReceiver(new Handler());
7  unitaPermissionsResultReceiver.setReceiver(this);
8
9  receiver = ReceiveController.getInstance();
10 receiver.init(unitaPermissionsResultReceiver);
11
12 locationController = LocationController.getInstance();
13 locationController.setCurrentContext(this);
14 locationController.initLocationTracker();
15
16 sender = new SendController(unitaPermissionsResultReceiver);
17
18 localMessageDatabase = LocalMessageDatabase.getInstance();
19 localMessageDatabase.addMessageListener(this);
20 localMessageDatabase.setCurrentContext(this);
```

Listing 7.29. Response listener of login from the server.

```
1  @Override
2      public void onLoginResponse(JSONObject loginResponse) {
3          int id = -1;
4          String name = null;
5          try {
6              id = loginResponse.getInt("id");
7          } catch (JSONException e) {
8              e.printStackTrace();
9          }
10         try {
11             name = loginResponse.getString("name");
12         } catch (JSONException e) {
13             e.printStackTrace();
14         }
15         if(id > (-1) && name != null){
16             Beacon loggedInBeacon = new Beacon(id, name);
17             LoginUtils.saveLoggedInBeacon(getApplicationContext(),
                   ↪ loggedInBeacon);
18             runOnUiThread(new Runnable() {
19                 @Override
20                 public void run() {
21                     startSocialWall();
```

```
22                  }
23              });
24          } else{
25      }
26 }
```

Listing 7.30. Listener example for a message type.

```
1 @Override
2 public void onCommandMessageReceived(CommandMessage
    ↪ receivedMessage) {
3         if(checkIfMessageIsNotFromMyself(receivedMessage.getHeader
            ↪ ().getSender(), LoginUtils.getLoggedInBeacon(
            ↪ getApplicationContext())) &&
4                 checkIfMessageIsForMyself(receivedMessage.
                    ↪ getHeader().getReceiver(), LoginUtils.
                    ↪ getLoggedInBeacon(getApplicationContext()))
                    ↪ ){
5             Routines.onCommandMessageReceivedRoutine(
                ↪ receivedMessage, this);
6         }else{
7         }
8 }
```

Listing 7.31. Listener example for url response from server.

```
1 @Override
2 public void onGetUrlForCommandGetAllMessagesResult(JSONObject
    ↪ urlResponse, JSONObject senderResponse) {
3         String url = null;
4         try {
5             url = urlResponse.getString("url");
6         } catch (JSONException e) {
7             e.printStackTrace();
8         }
9         UrlMessage urlMessage = new UrlMessage(LoginUtils.
            ↪ getLoggedInBeacon(getApplicationContext()),
            ↪ MessageUtils.convertJSONToPeer(senderResponse), url
            ↪ .getBytes(StandardCharsets.UTF_8));
10        Routines.sendMessage(urlMessage, sender, unitaSettings,
            ↪ getApplicationContext());
11 }
```

Listing 7.32. Routine example of further TextMessage processing.

```
1  public static void onTextMessageReceivedRoutine(at.floriantaurer.
       ↪ unitabeaconmodule.TextMessage receivedTextMessage,
       ↪ SocketController socket, Activity activity){
2          StatusMessage statusMessage = new StatusMessage(LoginUtils
               ↪ .getLoggedInBeacon(activity), receivedTextMessage.
               ↪ getHeader().getSender(), StatusCodes.
               ↪ MESSAGE_PACKET_RECEIVED_SUCCESS.toString().getBytes
               ↪ (StandardCharsets.UTF_8));
3          sendMessage(statusMessage, MainActivity.sender,
               ↪ MainActivity.unitaSettings, activity);
4          TextMessage textMessage = MessageUtils.
               ↪ convertUnitaTextMessageToSocialWallTextMessage(
               ↪ receivedTextMessage.getHeader().getSender(),
               ↪ receivedTextMessage.getHeader().getReceiver(),
               ↪ receivedTextMessage);
5          if(textMessage.isPublic()){
6              Gson gson = new Gson();
7              String jsonString = gson.toJson(textMessage);
8              JSONObject messageJSON = null;
9              try {
10                 messageJSON = new JSONObject(jsonString);
11             } catch (JSONException e) {
12                 e.printStackTrace();
13             }
14             socket.saveMessage(messageJSON);
15         }else {
16             LocalMessageDatabase localMessageDatabase =
                   ↪ LocalMessageDatabase.getInstance();
17             localMessageDatabase.saveMessage(textMessage);
18         }
19 }
```

Listing 7.33. User login process with REST call.

```
1  private void loginUser(String userName){
2          int NUMBER_OF_CORES = Runtime.getRuntime().
               ↪ availableProcessors();
```

```
3          final ScheduledExecutorService threadPool = Executors.
               ↪ newScheduledThreadPool(NUMBER_OF_CORES + 1);
4
5          threadPool.execute(new Runnable() {
6              @Override
7              public void run() {
8                  final SocialWallAPI restService = RESTController.
                       ↪ getRetrofitInstance().create(SocialWallAPI.
                       ↪ class);
9
10                 restService.loginUser(userName).enqueue(new
                       ↪ Callback<ResponseBody>() {
11                     @Override
12                     public void onResponse(Call<ResponseBody> call
                           ↪ , Response<ResponseBody> response) {
13                         if (response.isSuccessful()) {
14                             //Log.i("LoginActivity", "post
                                   ↪ submitted to API." +
                                   ↪ getStringFromRetrofitResponse(
                                   ↪ response));
15                             String detailsString =
                                   ↪ getStringFromRetrofitResponse(
                                   ↪ response);
16                             Log.i("LoginActivity", "post␣submitted
                                   ↪ ␣to␣API.␣" + detailsString);
17                             JSONObject jObject = null;
18                             try {
19                                 jObject = new JSONObject(
                                       ↪ detailsString);
20                             } catch (JSONException e) {
21                                 e.printStackTrace();
22                             }
23                             Log.i("LoginActivity", "post␣submitted
                                   ↪ ␣to␣API." + jObject.toString())
                                   ↪ ;
24                             JSONObject jsonObject = null;
25                             try {
26                                 jsonObject = (JSONObject) jObject
                                       ↪ .getJSONArray("user").
                                       ↪ getJSONObject(0);
```

```
27
28                              } catch (JSONException e) {
29                                  e.printStackTrace();
30                              }
31                              Log.i("LoginActivity", "post␣submitted
                                    ↪ ␣to␣API." + jsonObject.toString
                                    ↪ ());
32
33                              onLoginResponse(jsonObject);
34                          }
35                      }
36
37                      @Override
38                      public void onFailure(Call<ResponseBody> call,
                            ↪ Throwable t) {
39                          Log.e("LoginActivity", "Unable␣to␣submit␣
                                ↪ post␣to␣API." + t);
40                      }
41                  });
42              }
43          });
44 }
```

Listing 7.34. Processing for a received TextMessage.

```
1 Peer beacon = (Beacon)spnMsgReceiver.getSelectedItem();
2 TextMessage textMessage = null;
3 if(rbtPublic.isChecked()){
4       Peer communicationPartner = new Broadcast();
5       textMessage = new TextMessage(LoginUtils.getLoggedInUser(
            ↪ getActivity()), beacon, communicationPartner,
            ↪ edtMsgMessage.getText().toString(), true);
6 }else if(rbtPrivate.isChecked()){
7         Peer communicationPartner = (User)spnContacts.
              ↪ getSelectedItem();
8         textMessage = new TextMessage(LoginUtils.getLoggedInUser(
              ↪ getActivity()), beacon, communicationPartner,
              ↪ edtMsgMessage.getText().toString(), false);
9 }
10 at.floriantaurer.unitabeaconmodule.TextMessage unitaTextMessage =
     ↪ new at.floriantaurer.unitabeaconmodule.TextMessage(
```

```
    ↪ textMessage.getHeader().getSender(), textMessage.getHeader
    ↪ ().getReceiver(), textMessage.getCommunicationPartner(),
    ↪ MessageUtils.
    ↪ convertSocialWallTextMessageAdditionalParametersToByteArray
    ↪ (textMessage.isPublic(), textMessage.getMessageBody().
    ↪ getMessageBodyRaw()));
Routines.sendMessage(unitaTextMessage, MainActivity.sender,
    ↪ MainActivity.unitaSettings, getActivity());
```

Listing 7.35. REST API endpoints interface for client app.

```
public interface SocialWallAPI {
    @GET
    Call<ResponseBody> sendUrlToServer(@Url String url);

    @GET("/loginUser")
    Call<ResponseBody> loginUser(@Query("userName") String
        ↪ userName);
}
```

C Questionnaires

The following pages include the english and german questionnaire based interview for the user study with the tasks to fulfill:

Test Guide

My name is Florian and will guide you through the test. Before we start, I will give you a short introduction about ultrasound, ultrasonic beacons and the test procedure.

For my master thesis, I test my ultrasonic beacon called “Unita beacon” which uses the application “SocialWall”, I developed as a use case for my beacon. The beacon uses ultrasound for communication. This means the messages, which are sent, are located in the frequency band between 18kHz and 22kHz, which for most people is not audible anymore. Those messages are not emitted through walls, instead it is limited to a room. The application “SocialWall” is a kind of black board, where messages can be left and retrieved.

I will start with some general questions and then there are 6 tasks to work through. Afterwards, there will be a short evaluation of the usability of the system and then an interview about your personal assessment of the beacon and the application.

Since the test will be audio recorded and not used outside of my master thesis, I ask you to sign the consent form please. All data will be saved and handled anonymously.

If you have any questions while testing, please ask right away and express your thoughts on the individual steps through the tasks aloud.

Do you have questions now?

General data

Age: _____

Sex: ○ Female ○ Male ○ Other ○ No answer

How many minutes per day do you spend using your smartphone?

○ Never / I don't have one
○ Not daily ○ Less than 10 minutes ○ 10 to 15 minutes ○ 15 to 30 minutes
○ 30 to 45 minutes ○ 45 to 60 minutes ○ more than 61 minutes

What is your profession?

Which qualification do you have?

For which tasks do you mostly use your smartphone?

Which of these tasks have you performed before?

Mobile Payment with NFC

Tickets (concerts, theater,,..) with NFC

Pairing with Bluetooth speaker

Pairing with loudspeaker with NFC

Entrance with NFC

Connecting with media-stick/media-box with Wi-Fi

Connecting with Chromecast with Bluetooth

Combining loudspeakers with Wi-Fi

Others:

Have you already used location-based services?

Have you heard about ultrasonic communication before? If yes, in which context? What do you know about it?

Did you know that most smartphones can send and receive ultrasonic signals?

Have you actively used ultrasound before? If yes, how?

Do you like the idea of sending data over inaudible sound? Why?

	1	2	3	4	5	
Do not like the idea	○	○	○	○	○	Like the idea

Would you use ultrasound for interaction with other devices? Why?

	1	2	3	4	5	
Would not use	○	○	○	○	○	Would use

Would you accept ultrasonic communication?

1.	Useful	|__|__|__|__|__|	Useless
2.	Pleasant	|__|__|__|__|__|	Unpleasant
3.	Bad	|__|__|__|__|__|	Good
4.	Nice	|__|__|__|__|__|	Annoying
5.	Effective	|__|__|__|__|__|	Superfluous
6.	Irritating	|__|__|__|__|__|	Likeable
7.	Assisting	|__|__|__|__|__|	Worthless
8.	Undesirable	|__|__|__|__|__|	Desirable
9.	Raising Alertness	|__|__|__|__|__|	Sleep-inducing

Would you use ultrasound for following tasks?

Mobile Payment with NFC

Tickets (concerts, theater,,..) with NFC

Pairing with Bluetooth speaker

Pairing with loudspeaker with NFC

Entrance with NFC

Connecting with media-stick/media-box with Wi-Fi

Connecting with Chromecast with Bluetooth

Combining loudspeakers with Wi-Fi

Others:

Task 1 – Pairing

You are passing by the entrance of the university main building and spot the Unita beacon. You want to try it out with the client of the beacon, “SocialWall”. Therefore, you need to pair your smartphone with the beacon. The application is already downloaded, and you are logged in. Pair your smartphone now with the Unita beacon.

Task 2 – Write a public message

You want to leave a message for all people which are entering the university main building. Write a public message with the text “The FH offers many training opportunities!”.

Task 3 – Check for public messages

After a work day you want to have a look what other people wrote on the black board. Retrieve all public messages.

Task 4 – Add a contact to your list

After a talk with one of your colleagues, you found out she is also using the application “SocialWall”. You want to leave private messages for her. Therefore, you asked for his “SocialWall”-ID which is 10. Add her as a contact now with the name “Sonic”.

Task 5 – Write a private message

You want to leave your new contact a private message. Write a private message for the user “Sonic” with the message text “Hey, what’s up?”.

Task 6 – Get the latest private message

After some time, you want to have a look, if the user “Sonic” left a new message for you. Retrieve the latest private message.

UNITA

System Usability Scale

Instructions: For each of the following statements, mark one box that best describes your reactions to the ultrasonic beacon and the application “SocialWall”.

I think that I would like to use this system frequently.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

I found the system unnecessarily complex.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

I thought the system was easy to use.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

I think that I would need the support of a technical person to be able to use this system.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

I found the various functions in this system were well integrated.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

I thought there was too much inconsistency in this system.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Questionnaire-based interview

Did you hear something unusual while interacting with the beacon? (e.g. crack, noise, high-pitched sound)

How intuitive is the application “SocialWall”?

	1	2	3	4	5	
Unintuitive	○	○	○	○	○	Intuitive

Was the application easy to use?

	1	2	3	4	5	
Inaccessible	○	○	○	○	○	Easy to use

Did you find the pairing process via sound convenient?

	1	2	3	4	5	
Inconvenient	○	○	○	○	○	Convenient

Did you have problems somewhere?

Would you need additional feedback while sending and receiving messages? If yes, in which form?

Do you like the idea of such an ultrasound black board to leave messages?

	1	2	3	4	5	
Dislike	○	○	○	○	○	Like

What do you think about the aspect, that “SocialWall” can only be used on location?

Would you use it if such beacons were available in public places? If yes, why?

Do you see possible improvements to add?

Testablauf

Mein Name ist Florian und ich werde Sie durch den Test begleiten. Vor dem Start werde ich Ihnen eine Kurze Einleitung zu Ultraschall, Ultraschallbeacons und dem Testablauf geben.

Für meine Diplomarbeit teste ich mein Ultraschallbeacon genannt „Unita beacon“, dass die Anwendung „SocialWall“ verwendet, die ich als Anwendungsfall für mein Beacon programmiert habe. Das Beacon nutzt Ultraschall um zu kommunizieren. Dies bedeutet, dass die Nachrichten in einem Frequenzband zwischen 18kHz und 22kHz gesendet werden. Die meisten Menschen können diese hohen Frequenzen nicht mehr hören. Die ausgesendeten Nachrichten können nicht durch Wände gesendet werden und sind somit auf einen Raum begrenzt. Die Anwendunge “SocialWall” ist.

Ich werde mit allgemeinen Fragen starten, bevor wir zu 6 Aufgaben kommen. Danach wird eine kurze Auswertung der Usability sein und anschließend, werde ich Fragen zur Interaktion mit dem Beacon selbst wie mit der Applikation stellen.

Da der Test aufgenommen wird und nicht außerhalb meiner Diplomarbeit verwendet wird, bitte ich Sie die Einverständniserklärung zu unterschreiben. Alle Daten werden anonymisiert gespeichert und gehandhabt.

Falls Sie während des Tests Fragen haben, einfach gleich fragen. Bitte sprechen Sie alle Überlegungen während der Aufgaben laut aus.

Haben Sie Fragen?

Allgemeine Angaben

Alter: _____

Geschlecht: ○ weiblich ○ männlich ○ anderes ○ keine Antwort

Wie viele Minuten pro Tag verwenden sie Ihr Smartphone?

○ nie/Ich besitze keines
○ nicht täglich ○ weniger als 10 Minuten ○ 10 bis 15 Minuten ○ 15 bis 30 Minuten
○ 30 bis 45 Minuten ○ 45 bis 60 Minuten ○ mehr als 61 Minuten

Was ist ihr Beruf?

Welche Ausbildung besitzen Sie?

Für welche Aufgaben verwenden Sie ihr Smartphone am meisten?

Welche dieser Aufgaben haben Sie schon durchgeführt?

Mobiles Zahlen mit NFC

Tickets (Konzerte, Theater,..) mit NFC

Verbinden mit Bluetooth Lautsprechern

Verbinden mit Lautsprechern über NFC

Zutritt zu Gebäuden mit NFC

Verbinden von Medienboxen mit Wi-Fi

Verbinden vom Chromecast mit Bluetooth

Verbinden von Lautsprechern untereinander mit Wi-Fi

Andere:

Haben Sie schon standortbasierte Services genutzt?

Haben Sie schon mal von Ultraschallkommunikation gehört? Wenn ja, in welchem Zusammenhang? Was wissen Sie darüber?

Haben Sie gewusst, dass die meisten Smartphones Ultraschallsignale senden und empfangen können?

Haben Sie schon mal Ultraschall aktiv genutzt? Wenn ja, wie?

Finden Sie die Idee gut, Daten über nichthörbaren Schall zu übertragen? Warum?

	1	2	3	4	5	
Gefällt die Idee nicht	○	○	○	○	○	Gefällt die Idee

Würden Sie Ultraschall zum Interagieren mit anderen Geräten nutzen? Warum?

	1	2	3	4	5	
Würde es nicht nutzen	○	○	○	○	○	Würde es nutzen

Würden Sie Ultraschallkommunikation akzeptieren?

1	Nützlich	|__|__|__|__|__| Nutzlos
2	Angenehm	|__|__|__|__|__| Unangenehm
3	Schlecht	|__|__|__|__|__| Gut
4	Nett	|__|__|__|__|__| Nervig
5	Effizient	|__|__|__|__|__| Unnötig
6	Ärgerlich	|__|__|__|__|__| Erfreulich
7	Hilfreich	|__|__|__|__|__| Wertlos
8	Nicht wünschenswert	|__|__|__|__|__| Wünschenswert
9	Aktivierend	|__|__|__|__|__| Einschläfernd

Würden Sie Ultraschall für folgende Aufgaben verwenden?

Mobiles Zahlen mit NFC

Tickets (Konzerte, Theater,..) mit NFC

Verbinden mit Bluetooth Lautsprechern

Verbinden mit Lautsprechern über NFC

Zutritt zu Gebäuden mit NFC

Verbinden von Medienboxen mit Wi-Fi

Verbinden vom Chromecast mit Bluetooth

Verbinden von Lautsprechern untereinander mit Wi-Fi

Andere:

Aufgabe 1 – Verbinden

Sie passieren den Eingang des Fachhochschulgebäudes und sehen das Unita Beacon. Sie möchten es mit der App des Beacons, “SocialWall”, ausprobieren. Dafür müssen Sie ihr Smartphone mit dem Beacon verbinden. Die App ist schon heruntergeladen und Sie sind schon angemeldet. Verbinden Sie ihr Smartphone nun mit dem Unita Beacon.

Aufgabe 2 – Öffentliche Nachricht schreiben

Sie wollen eine Nachricht für alle Leute, die das Fachhochschulgebäude betreten, hinterlassen. Schreiben Sie eine öffentliche Nachricht mit dem Text „Die FH bietet viele Fortbildungsmöglichkeiten an!“.

Aufgabe 3 – Nach öffentlichen Nachrichten prüfen

Nach einem Arbeitstag wollen Sie nachsehen, ob andere Personen Nachrichten ans schwarze Brett geschrieben haben. Rufen Sie alle öffentlichen Nachrichten ab.

Aufgabe 4 –Kontakt zur Liste hinzufügen

Nach einem kurzen Gespräch mit einer Kollegin, haben Sie herausgefunden, dass sie auch die App „SocialWall“ verwendet. Sie wollen eine private Nachricht für sie hinterlassen. Dafür haben Sie sie nach ihrer „SocialWall“-ID gefragt. Diese ist 10. Fügen Sie sie als Kontakt mit dem Namen „Sonic“ hinzu.

Aufgabe 5 – Private Nachricht senden

Sie wollen für ihren neuen Kontakt eine private Nachricht hinterlassen. Schreiben Sie eine private Nachricht für die Userin „Sonic“ mit dem Text „Hallo, wie geht’s?“.

Aufgabe 6 – Letzte private Nachricht bekommen

Nach einiger Zeit wollen Sie nachsehen, ob die Userin „Sonic“ eine neue Nachricht für Sie hinterlassen hat. Rufen Sie die letzte private Nachricht ab.

System Usability Scale

Markieren Sie für jede der folgenden Aussagen das Kästchen, was Ihre heutigen Reaktionen auf das Ultraschallbeacon und die Applikation „SocialWall“ am besten beschreib.

Ich kann mir sehr gut vorstellen, das System regelmäßig zu nutzen.

	1	2	3	4	5	
Stimme nicht zu	○	○	○	○	○	Stimme zu

Ich empfinde das System als unnötig komplex.

	1	2	3	4	5	
Stimme nicht zu	○	○	○	○	○	Stimme zu

Ich empfinde das System als einfach zu nutzen.

	1	2	3	4	5	
Stimme nicht zu	○	○	○	○	○	Stimme zu

Ich denke, dass ich technische Unterstützung benötigen würde, um das System zu nutzen.

	1	2	3	4	5	
Stimme nicht zu	○	○	○	○	○	Stimme zu

Ich finde, dass die verschiedenen Funktionen des Systems gut integriert sind.

	1	2	3	4	5	
Stimme nicht zu	○	○	○	○	○	Stimme zu

Ich finde, dass es im System zu viele Inkonsistenzen gibt.

	1	2	3	4	5	
Stimme nicht zu	○	○	○	○	○	Stimme zu

Ich kann mir vorstellen, dass die meisten Leute das System schnell zu beherrschen lernen.

	1	2	3	4	5	
Stimme nicht zu	○	○	○	○	○	Stimme zu

Ich empfinde die Bedienung als sehr umständlich.

	1	2	3	4	5	
Stimme nicht zu	○	○	○	○	○	Stimme zu

Ich habe mich bei der Nutzung des Systems sehr sicher gefühlt.

	1	2	3	4	5	
Stimme nicht zu	○	○	○	○	○	Stimme zu

Ich musste eine Menge Dinge lernen, bevor ich mit dem System arbeiten konnte.

	1	2	3	4	5	
Stimme nicht zu	○	○	○	○	○	Stimme zu

Fragebogengestütztes Interview

Haben Sie etwas Ungewöhnliches gehört, während Sie mit dem Beacon interagiert haben? (z.B. Knacken, Rauschen, hohe Klänge)

Wie intuitiv ist die App „SocialWall“?

	1	2	3	4	5	
Nicht intuitiv	○	○	○	○	○	Intuitiv

War die App einfach zu bedienen?

	1	2	3	4	5	
Unzugänglich	○	○	○	○	○	Einfach zu bedienen

Fanden Sie den Verbindungsprozess über Schall bequem?

	1	2	3	4	5	
Unbequem	○	○	○	○	○	Bequem

Traten irgendwo Probleme auf?

Bräuchten Sie noch zusätzliches Feedback während des Sendens und Empfanges von Nachrichten? Wenn ja, in welcher Form?

Gefällt Ihnen die Idee eines solchen ultraschallbasierten Schwarzen Brettes um Nachrichten zu hinterlassen?

	1	2	3	4	5	
Gefällt nicht	○	○	○	○	○	Gefällt

Wie finden Sie den Aspekt, dass “SocialWall” nur direkt vor Ort genutzt werden kann?

Würden Sie solch ein Beacon nutzen, wenn es an einem öffentlichen Ort aufgebaut wäre? Wenn ja, warum?

Hätten Sie noch mögliche Verbesserungen?