Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Ultrasonic Smartphone Communication

Bachelor thesis

Pascal Maillard

pascalm@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Simon Tanner, Gino Brunner
Prof. Dr. Roger Wattenhofer

April 23, 2018

Acknowledgements

I would like to thank both my supervisors, Simon Tanner and Gino Brunner, who helped me throughout this thesis with new ideas and advice on how to tackle the problems I encountered.
I also thank my friends who offered me general advice on this project.

Abstract

The goal of this thesis is to implement a chat application that is able to transmit messages to other nearby devices without pairing them first.

We use the ability of current smartphones to play and record audio in the near ultrasonic range to broadcast our messages. We achieve this by using frequency modulation via ultrasound to encode and transmit our data while nearby devices listen for any incoming messages.

From our results, we can see that we achieved to reliably send small messages to other nearby devices over distances of up to 45cm. It is possible to send further by increasing the duration of a message, although with a decrease in reliability.

Contents

Chapter 1

Introduction

1.1 Motivation

When sharing links or chatting on smartphones, the communication partner normally has to be known. Sometimes it would be useful to be able to share a link with your friends or maybe you want to chat with other people in the same room that you do not know the phone number of. In these situations it would be useful to have a simple way to communicate between nearby devices.

Modern smartphones are able to play and record audio in the near ultrasonic range. Therefore, it should be possible to use these signals to communicate and transmit small chat messages between devices.

Compared to other technologies available on smartphones, this connection does not need any pairing of the phones. There is no need for the user to learn the other person's contact information or scan a code on the other device.

1.2 Related Work

Nowadays, there are many different technologies available that can transmit data to nearby devices, such as Bluetooth, audio, NFC, QR-Codes or Wi-Fi.

For example, Google's Nearby Messages API[1] uses a combination of Bluetooth, ultrasonic audio and Wi-Fi to reach other devices and transmit data. A device can listen on some or all of these different channels for messages. It supports messages of up to 100 KB in size, however they recommend to keep them around 3 KB to guarantee faster transmission times.

Chirp.io[2] and their SDK provide the ability to transmit data over audio on a variety of different platforms. Their library implements interfaces to Java, JavaScript, Python, Android and iOS. It works in either audible or ultrasonic frequency ranges and transmits data by using frequency modulation. The transmission rate is about 50 to 100 bits per second.

Chapter 2

Theory

2.1 Frequency Modulation

In signal processing, frequency modulation is the method used to encode information in a signal by varying the instantaneous frequency. The modulated signal is created by starting out with a carrier signal at a constant frequency f_c, which is then modified by the amplitude of the modulating signal. An increase in amplitude of the modulating signal will result in an increase of frequency of the final modulated signal.

The instantaneous frequency over time $f_{inst}(\tau)$ tells us which frequency our signal will have at which moment in time. This is where the information can be encoded directly by taking the modulating signal as $f_{inst}(\tau)$.

Alternatively, we can set our modulating signal to be $x_m(\tau)$. If its amplitude is not within ± 1 we have to scale it down first. Using this method limits the frequency deviation to a specific range f_Δ and the modulated signal can only take on frequencies within $f_c \pm f_\Delta$.

$$y(t) = A \cdot cos(2\pi \cdot \int_0^t f_{inst}(\tau) d\tau) \tag{2.1}$$

$$y(t) = A \cdot cos(2\pi f_c t + 2\pi f_\Delta \cdot \int_0^t x_m(\tau) d\tau) \tag{2.2}$$

2.1.1 Binary Frequency Modulation

Binary frequency modulation is a special case where $f_{inst}(\tau)$, and as a result also the final modulated signal, can only take on two distinct frequencies. A high frequency represents a binary 1 and a low frequency represents a 0. This makes transmitting digital data in binary format straightforward. As seen in Figure 2.1, the final modulated signal has a high frequency where the original data is 1 and a lower frequency for the other parts.

In order to demodulate the data, the frequency of each segment in the signal is checked whether it corresponds to the higher or lower frequency. After we have done this for the entire signal, we have recovered the data that was originally sent.

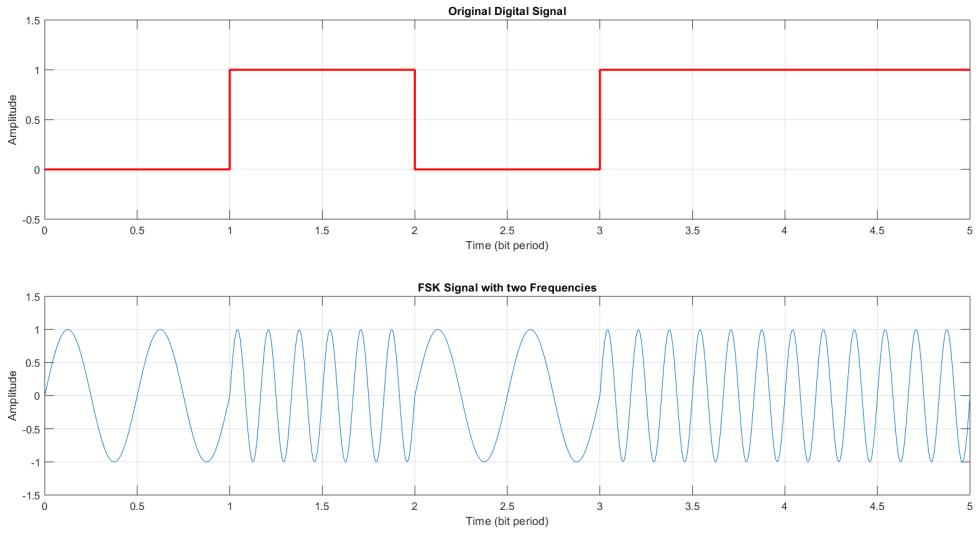

Figure 2.1: Binary frequency modulation[3]. The binary data is converted into a rectangular signal. In parts where this function represents a binary 1, the final modulated signal has the higher frequency. And in the other areas, the lower frequency is used.

Chapter 3

Implementation

3.1 Start Symbol

In order to receive a message, it is necessary for the receiver to somehow know that a message was sent in the first place. It is impractical to try to demodulate all incoming audio data in the hope that it may contain one. Moreover, we need to know where the beginning of the payload is for the demodulation. This is both accomplished with the help of a start symbol.

$$startSym(t) = \begin{cases} cos\left(2\pi \cdot (f_{s1}t + \frac{k}{2}t^2)\right), & \text{if } 0 \leq t \leq T \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

$$k = \frac{f_{s0} - f_{s1}}{T} \tag{3.2}$$

The start symbol consists of a down-chirp that starts at a higher frequency f_{s1} and ramps down linearly to f_{s0} over the duration of T seconds. Such a frequency sweep has a very good auto-correlation, as seen in Figure 3.1. There is a single clearly defined peak and even the slightest displacement in either direction causes the correlation coefficient to decrease. That makes it easy to check whether or not the symbol is present in the audio data and it gives us the exact position in time where the coefficient peaks.

Additionally, to increase the chance that the start symbol was heard successfully it has a duration of 125ms, which is far longer than a normal payload symbol of around 1 to 2ms.

In our implementation, we use a frequency of 20.1kHz for f_{s1} and 19kHz for f_{s0}.

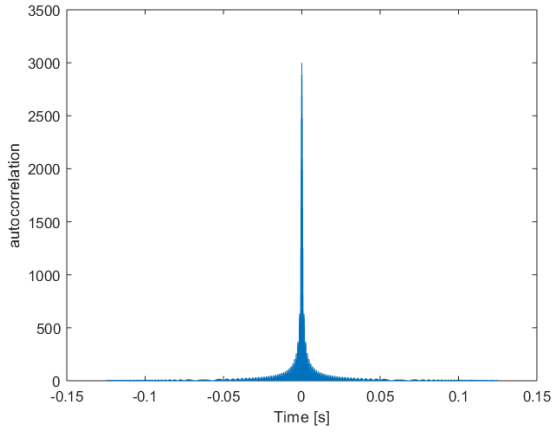

Figure 3.1: Auto-correlation of the start symbol. There is a clearly defined peak at index zero, since the highest similarity is achieved when the signal overlaps itself perfectly and there is no displacement.

3.1.1 Finding the Start Symbol

On the device, incoming audio data is checked for the presence of the start symbol. This is done by calculating the cross-correlation coefficient of the recorded audio signal x with the start symbol y. This gives us a normalized coefficient for the degree of similarity between the two signals in the range of $[0, 1]$. With 1 meaning the signals match perfectly.

$$xcorrCoeff(x, y) = \frac{xcorr(x, y)^2}{max(|xcorr(x, x)|) \cdot max(|xcorr(y, y)|)} \tag{3.3}$$

Performing this calculation over the entirety of the audio data is very costly. To combat this, we can use the convolution theorem, which tells us that taking the point-wise multiplication of the Fourier Transform (frequency-domain) is the same as a circular convolution in the time-domain. However, because we do not want the circular convolution, we pad both x and y with zeroes to get the linear convolution. Lastly, in order to get the correlation and not the convolution, we take the complex conjugate of the second term. This lets us calculate the cross-correlation efficiently as follows.

$$xcorr(x, y) = IFFT(FFT(x) \odot conj(FFT(y))) \tag{3.4}$$

Finally, this gives us the correlation coefficient of the start symbol with the audio signal at every position in the recorded data. If the maximum coefficient (see Figure 3.2) is above a certain threshold, we say that the start symbol was found at that point in time.

For the implementation, the multi-threaded Java-library JTransforms[4] was used for the Fourier Transform. This way, calculating the coefficient is fast enough to keep up with the recorded audio in real-time.

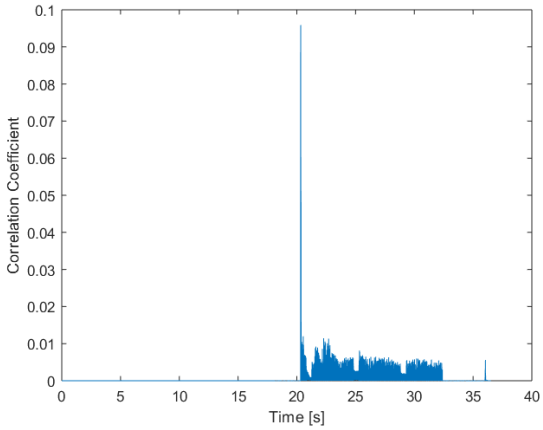

Figure 3.2: Cross-correlation of the start symbol over the entire audio signal. The coefficient has a clear and well defined peak where the start symbol is located. The background noise has a coefficient of nearly zero and the payload of the message can be seen as having a very weak correlation.

3.2 Message Format

Offset	Bytes			
	0 - 50	to 50	1	120
0	payload	zero-pad	checksum	error correction
171	payload	zero-pad	checksum	error correction
342	payload	zero-pad	checksum	error correction

Figure 3.3: A message consists of a payload that is zero-padded to a length of 50 bytes followed by a checksum and then the error correction. This entire block is then repeated three times.

Overall, the transmission is split into two parts. Firstly, every transmission begins with the start symbol to mark the beginning of a message. What follows is the message itself as specified in Figure 3.3.

The data we send is made up of the content of a single message repeated 3 times as a repetition code to correct small and individual bit-errors based on a majority decision. A single message consists of a payload, checksum and error-correcting codes. The forward error correction used here is Reed-Solomon which

is able to correct errors in the payload. More details on this can be found in Section 3.5.

Because the message we send has a fixed size, the payload has to be zero-padded to the proper length. The length of one message including error correction is 171 bytes. This gives us a payload length of 50 bytes which is large enough for us to send small text messages in a chat. Transmitting messages of a fixed size makes it simpler to receive them because there is no need to include information about the length prior or within the message, making it overall more robust.

As an example for how long a transmission takes, a message with a symbol duration of 50 samples is 5.6 seconds long and one with 80 samples is 8.2 seconds long.

3.3 Modulation

To begin a message, the start symbol is generated (as in Section 3.1) and then played. Afterwards, we take the text that we want to send and generate our message as specified in Section 3.2. That includes the text itself, a checksum and the error correction.

To transmit our data over an audio signal we use binary frequency modulation. Because we are only working with two frequencies, a single symbol that we send is 1 bit. Our two symbols for 1 and 0 are generated as follows.

$$\begin{aligned} s_1(t) &= cos(2\pi f_1 \cdot t),\ 0 \leq t \leq T_s \text{ , for bit 1} \\ s_0(t) &= cos(2\pi f_0 \cdot t),\ 0 \leq t \leq T_s \text{ , for bit 0} \end{aligned} \tag{3.5}$$

The symbols consist of a cosine-wave with the frequency f_1 for the value 1 and f_0 if the bit is 0. The symbol duration T_s, for both of them, is a fixed value to ensure that we know the exact position of every symbol when receiving. They usually have a duration of 1 to 2ms.
The symbol length and both frequencies can be changed by the user in the settings of the app, as shown in Figure 3.7.

To generate the instantaneous frequency $f_{inst}(\tau)$, we take the binary data of our message. For every bit in the message we send the appropriate symbol s_1 or s_0. Meaning, if the bit is 1 the instantaneous frequency at that point would be the higher frequency f_1 for the duration T_s of one symbol.

Between each symbol a small transition of duration $T_{transition}$ is inserted to make the frequency curve more smooth and avoid any undesired noises that result from having such large frequency jumps. Note that if the next symbol remains on the same frequency, this transition is also inserted to ensure that all symbols (including the transition) have the same duration. The next symbol

always follows $T_s + T_{transition}$ seconds after the current one, regardless what the previous symbol was.

After we have generated $f_{inst}(\tau)$, we can create our final modulated signal according to the Equation 2.1. An example of this process is shown in Figure 3.4.

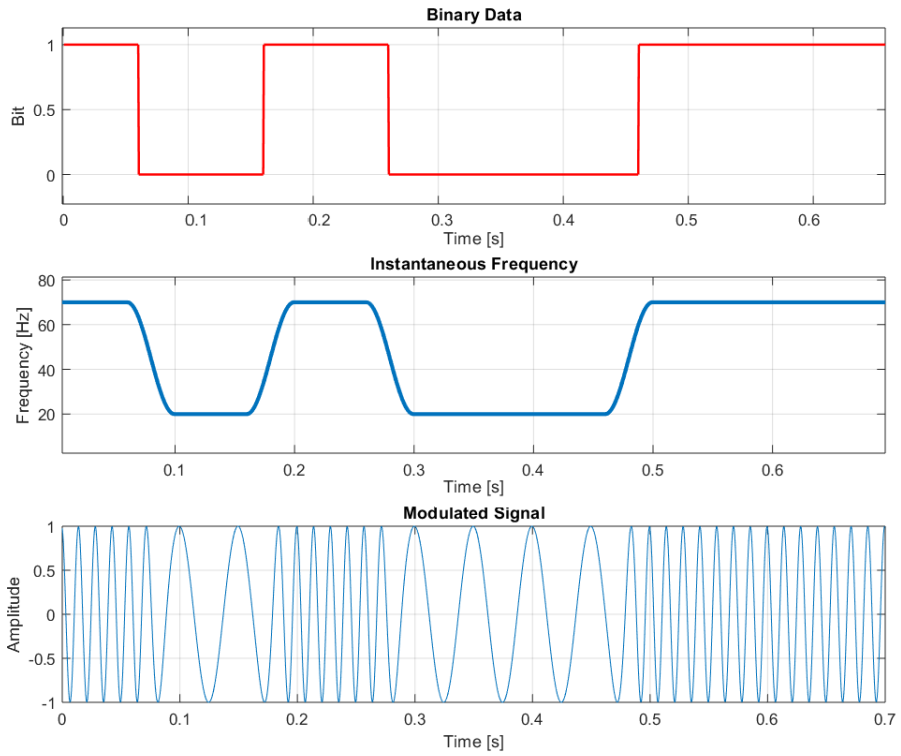

Figure 3.4: Example message of 1010011. First, the binary data is converted into a rectangular signal. Then we generate $f_{inst}(\tau)$. If the bit is 1 we use the higher frequency f_1, otherwise f_0. The instantaneous frequency over time follows the pattern of our binary data with small transitions added in between. Note that in this example the length of the transition is increased to make it more visible. The final modulated signal is shown beneath.

3.4 Demodulation

Internally, recording audio data on the device works by periodically receiving a filled buffer that contains the audio data. The first thing we have to do when receiving a message is to find the starting position of it. This is done by looking for the start symbol as explained in Section 3.1.1.

However, because we now have to work with split segments of audio data, we run into the problem that the start symbol might span across the border of such a buffer into the next one. To avoid this problem, we set an overlap for every incoming buffer that contains the ending of the previous one. If the start symbol is partially present at the end of the current buffer, it will be completely within the overlap of the next buffer and we are able to find it.

After we have found the starting position of our message, we begin the demodulation. Because we have chosen to send every symbol with a fixed duration, we know exactly how many audio samples a symbol occupies in the buffer. We then take the data that contains a symbol plus a small additional overlap on either side of it, to account for the fact that the phase might not be aligned and we get a bad correlation coefficient. On this data we perform a cross-correlation with both symbols s_1 and s_0 (from Equation 3.5). The maximum correlation coefficient tells us which symbol was most likely present.
This is performed for all symbols in the signal until we have reached the amount of bits necessary for a message.

Now, we have received the entire binary data of our message and need to extract the text inside. Firstly, with the help of our error correction in the message, we are able to fix some errors (see Section 3.5) and reconstruct the original message. Lastly, we verify if the checksum matches, meaning we have received the message correctly and were able to successfully fix all errors that occurred.

3.5 Error Correction

Sending data over audio is not highly reliable and errors might occur for various reasons, such as echoes, background noise or a too large distance. Therefore, we introduce error-correcting codes in our message which allows us to recover the data perfectly even if some errors are present.

A Reed-Solomon library[5] was used. For n error-correcting symbols added, it is able to correct up to $\frac{n}{2}$ symbols. In this context, a symbol refers to 8 bits of data in our message. This allows us to correct some bit-errors. It also deals with burst-errors well, because as long as at most 8 bits in a row were affected it only counts as a single symbol-error.

Unfortunately, a drawback of this particular library is that a symbol is always defined to be 8 bits, even tough Reed-Solomon could work with other symbol lengths too.

Additionally, we repeat our message three times as shown in Figure 3.3. This form of redundancy lets us repair additional errors.

3.5.1 Reconstruction

Firstly, we use the repetition of the message to correct small bit-errors based on a majority decision. This gives us a single message with the binary data that was most likely sent according to the repetition.

To further repair that message, we give it to the Reed-Solomon decoder. The library then performs the reconstruction as specified. If it is successful it gives us the original message we entered into the encoder back. If too many errors occurred it might fail, in which case it will return an error letting us know that no proper solution to the underlying mathematical problem was found. Additionally, the checksum of our message will not match the content, further letting us know that the message was not recovered properly.

To get an idea for how the error correction performs, we ran a simulation that corrupts and reconstructs messages directly without transmitting them. In our case, the error correction is able to successfully reconstruct a message about 76% of the time with a bit-error rate of 15% (see Figure 3.5). To calculate this we randomly generated 1000 messages. In these messages, we then randomly corrupted individual bits with a fixed bit-error rate and decoded them directly.

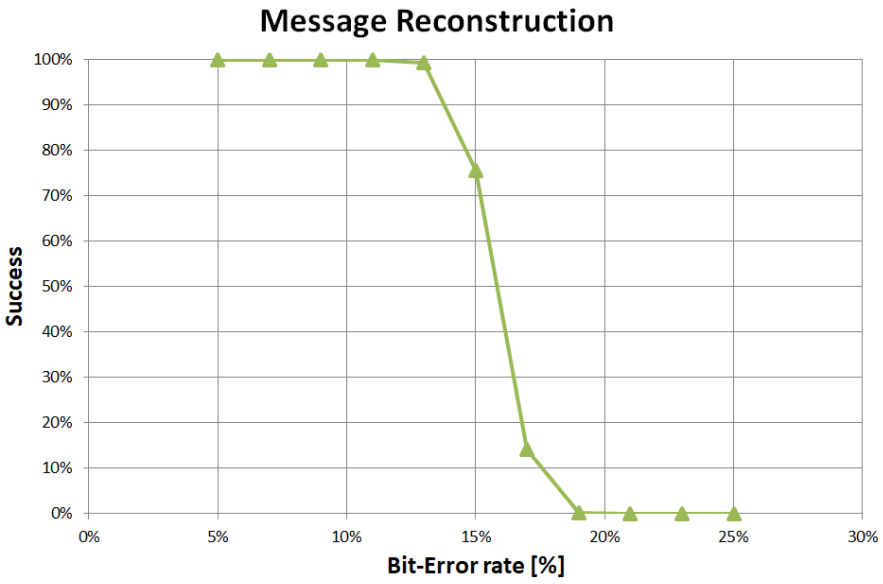

Figure 3.5: Graph that shows the rate of successfully reconstructing a message given a fixed bit-error rate. Usually, all errors can be corrected if the error rate is smaller than 15%. However, error rates above 20% are never recoverable.

3.6 Chat App

The home screen of the application is shown in Figure 3.6. In the top right corner there is a button that leads the users to the settings (see Figure 3.7), where they can change the frequencies and symbol duration for the modulation. Moreover, the users have the option to go to a screen that allows them to either send or receive data separately. Or they can go directly to the chat view (see Figure 3.8) that lets them do both simultaneously.

The chat application consists of two parts, a receiver and a sender, which both run concurrently.
The sender is responsible for taking the user's input from the text field and passing it along to be converted in the appropriate message format and played as audio. While doing so, it pauses the receiver to avoid listening to incoming messages as long as we are in the process of sending one ourselves.

The receiver checks all incoming audio buffers for the start symbol and demodulates them if a message is found, as seen in Section 3.4. If the receiver is in the process of demodulating a message, it prevents the sender from sending one. This prevents a collision of the incoming message with a possible outgoing message, which would make them both unrecoverable.

All received messages are displayed in the chat window together with the time they arrived.

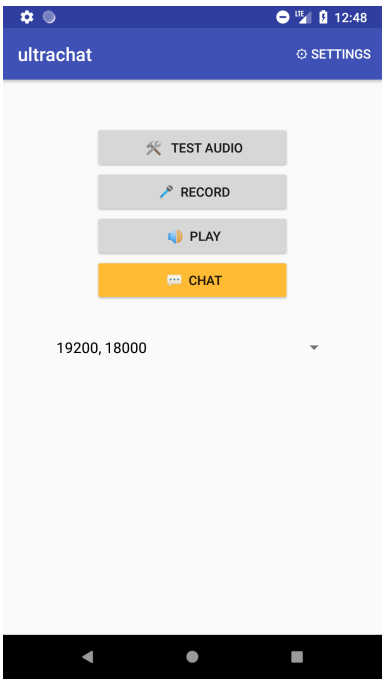

Figure 3.6: Home screen of the application.

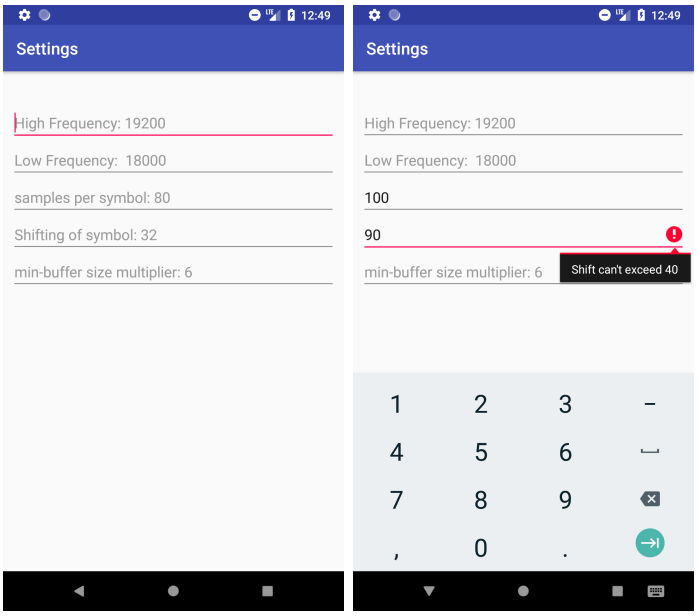

Figure 3.7: Settings where the user can change frequencies, symbol-length and buffer-size. Checks are performed to only allow valid values.

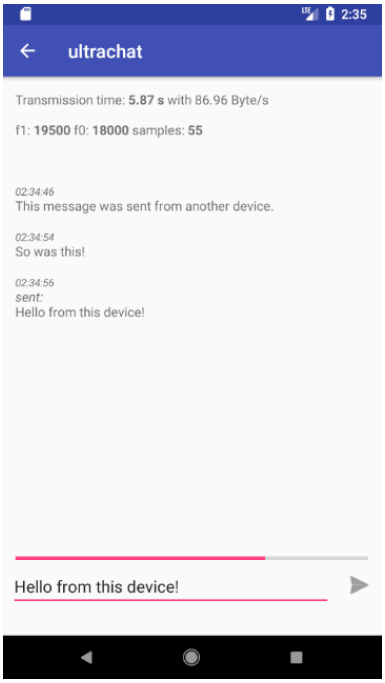

Figure 3.8: This is the chat screen of the application. The correctly received and sent messages are displayed here. The bar above the text field shows the progress of the ongoing transmission.

Chapter 4

Results

For our tests, the devices that were used are a LG Nexus 5X, a Huawei Nexus 6P and a LG Nexus 5. The LG Nexus 5 was always used to send the messages in our tests. This lets us directly compare the results of the two receiving devices since they receive the message from the same smartphone under identical conditions.

4.1 Symbol Size

We wanted to know what effect the duration of a symbol has on the transmission. We take one device and send messages to two other nearby devices over a distance of roughly 10cm. Every message was sent 15 times and we repeated this process with symbol durations ranging from 20 to 90 samples. A symbol of 20 samples is 0.42ms and one with 90 samples is 1.9ms long.

These tests were performed in three different set-ups. With the devices lying parallel on a desk, facing away from each other and also in such a way that the speaker of the transmitting device directly points towards the microphones of the others. The results are shown in Figure 4.1.

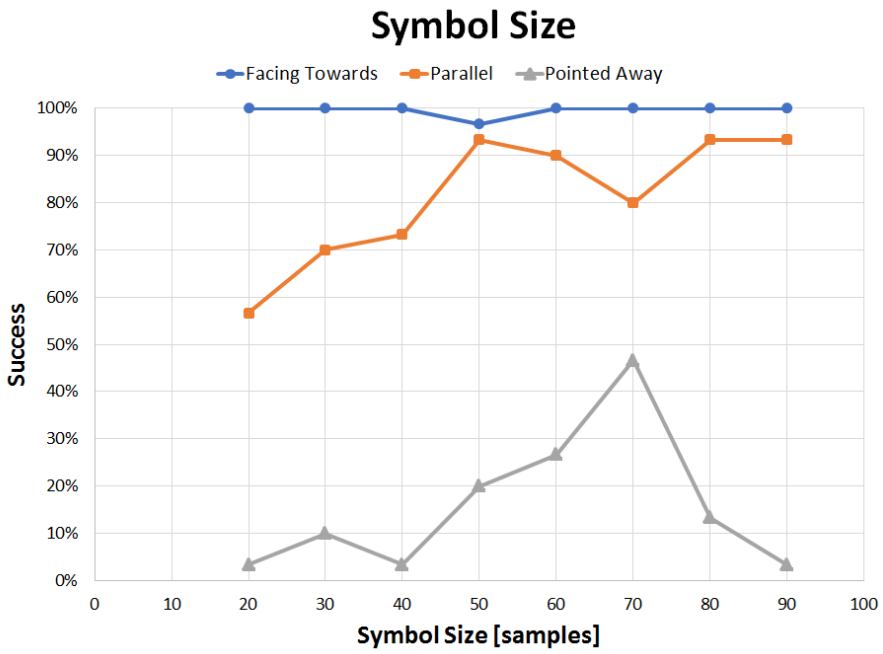

Figure 4.1: This graph shows the effect of varying the symbol size in combination with the device's orientation on messages sent over a distance of 10cm. We perform the test three times with the devices placed parallel, facing away from and towards each other. For every symbol size, we send a message 15 times.

In the test where the devices directly face one another the symbol size has no significant impact since all messages were received successfully. Similarly, when the devices lie adjacent on a desk the success-rate is overall almost as good as under the ideal circumstances from before and we can very reliably transmit our messages. However, if the duration of one symbol is shorter than 1ms ($\leq$ 50 samples), we can see that not quite all messages are received successfully.

On the other hand, when the devices are facing away from each other, there is a noticeable difference. Error rates are too high to properly recover the message in those cases. Interestingly, with an increase in symbol size the rate of success does not always increase but rather peak and then decrease again. This might be caused by reflections coming from a different direction that hit the microphone directly and interfere with the actual signal.

4.2 Microphone Orientation

From the previous test, we can clearly see that the relative orientation of the smartphones has a massive influence on the transmission. Shown in Figure 4.1,

we can see that over the same distance and with an identical symbol size, our message might arrive most of the time or only 45% of the time depending on where the signal is coming from.

It is best to always place the devices parallel next to each other or even better if the speaker is directly pointed at the microphone of the receiving device. Otherwise, we run into the problem that the signal is heard poorly.

4.3 Distance

We want to see at what range our chat application can still reliably operate. To test this we place two devices side by side while the third device is a set distance away. We then send a message 15 times and count how often it successfully arrived. This process is then repeated for the distances of 10, 30, 45, 60 and 100cm and with a symbol duration of 50 and 80 samples. A message with a symbol duration of 80 samples has a transmission time of 8.2 seconds, whereas one with a symbol duration of 50 samples is 5.6 seconds long. The results are shown in Figure 4.2.

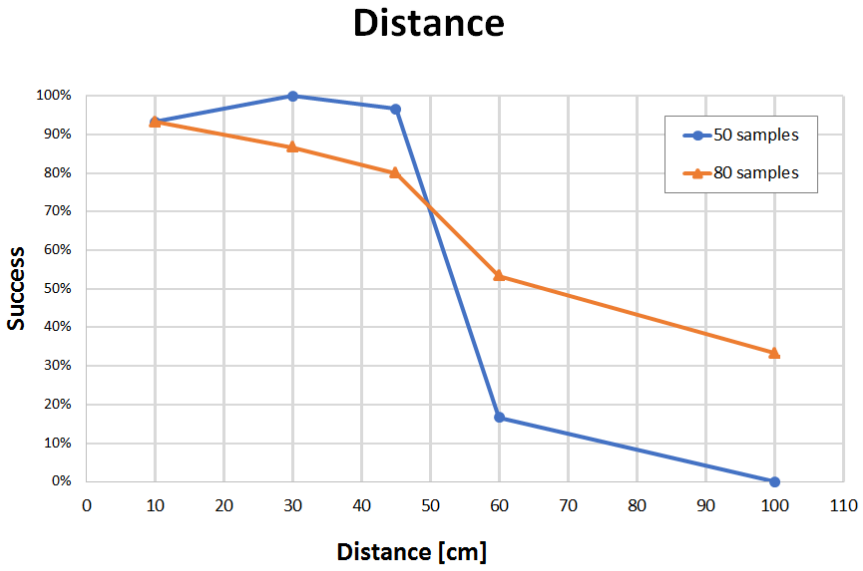

Figure 4.2: Messages transmitted from one device to two others over distances of 10, 30, 45, 60 and 100cm. These tests were performed with symbol durations of 50 and 80 samples. The results are based on 15 transmissions to both devices.

As one might expect, the amount of successful arrivals of our message declines the further away the source is. More symbols are not heard properly and the

number of mistakes rises which increases the probability that the message will not be recovered. In all our tests, it was never the case that the start symbol was not found.

As to be expected, transmissions with a larger symbol size have a greater reach than messages with a smaller one. This is because a larger symbol is played for a longer duration and as a result, has more energy. Disturbances would have to occur for a longer time to corrupt the majority of a symbol.

However, our results also strongly depend on the device itself as shown in the Figure 4.3 below. This is where the hardware of each device makes a difference based on the orientation of the microphone on the device itself, what the frequency response of the device is and if the audio is processed or filtered among other factors.

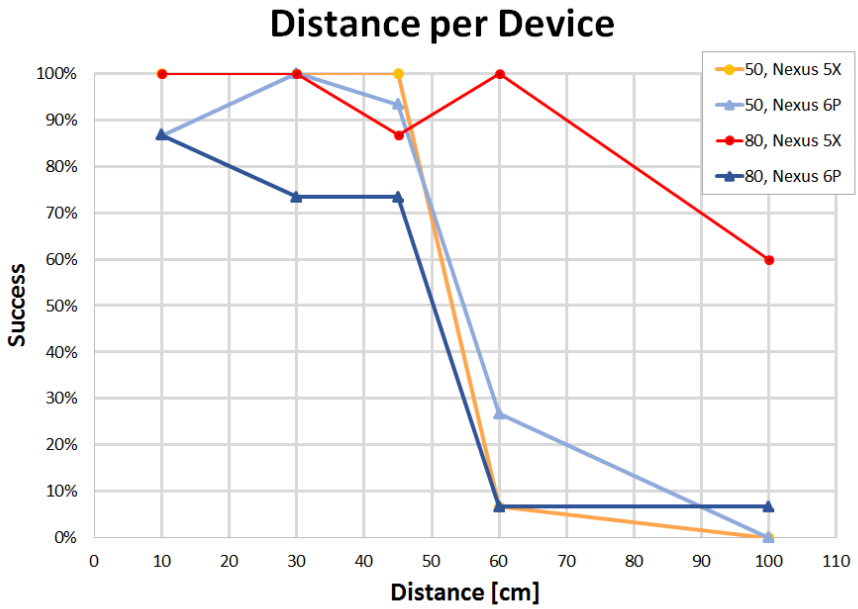

Figure 4.3: The results of the measurements vary from device to device. Here, all the measurements from the previous tests are shown per device. For the Nexus 5X, it still holds true that the longer a symbol is, the greater range the signal has. For the Nexus 6P, both symbol durations perform similarly.

Overall, both symbol sizes perform reliably up to a distance of 45cm. On average, the longer symbol duration is able to better transmit messages over a greater distance compared to the shorter symbol duration. But in our case, only one of the two devices was able to still reliably receive messages over a distance of 60cm even with a larger symbol size.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

From our results, we can see that the chat application is able to send messages over short distances of up to 45cm reliably with a symbol duration of around 1ms, as long as the smartphones are not pointed away from each other. Under good circumstances the symbol duration can be lowered slightly to achieve a higher transmission-rate.

However, greater distances of 55cm or more result in unrecoverable messages most of the time in our tests. It is possible to counteract this somewhat by choosing a longer symbol duration. But this does also vary depending on the device with some being able to receive messages from further away.

5.2 Future Work

The biggest room for improvement lies in making the transmission more reliable. That could mean a wider range, faster transmission times, allowing for a bigger payload and making it overall more robust to interference. This could be done by choosing an error correction method that is more equipped to deal with large number of individual bit-errors. In the current implementation the use of repetition codes results in the message being three times as large for a small bonus in reliability, which is not very efficient and could be improved.

Moreover, the frequency modulation scheme could be adapted to support more frequencies, instead of only two. This would allow for more information to be sent in a single symbol increasing the transmission-rate. It is also possible to use a different method to transmit data over audio, as opposed to frequency modulation. For example, minimum-shift keying or phase-shift keying.

Additionally, the available frequencies could be divided into separate channels to allow for simultaneous transmissions between devices.

Bibliography

[1] Google: Nearby messages api https://developers.google.com/nearby/messages/overview. (April 2018)

[2] Chirp.io: Chirp https://www.chirp.io/faq/. (March 2018)

[3] Khan, S.G.: Binary frequency shift keying https://ch.mathworks.com/matlabcentral/fileexchange/30581-binary-frequency-shift-keying. (April 2018)

[4] Wendykier, P.: Jtransforms https://github.com/wendykierp/JTransforms. (February 2018)

[5] ZXing: Reed-solomon https://github.com/zxing/zxing. (March 2018)