Distributed Speaker Synchronization

Semester Thesis

Kevin Luchsinger

`kevinlu@student.ethz.ch`

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Laura Peer, Pascal Bissig
Prof. Dr. Roger Wattenhofer

May 28, 2015

Abstract

In the past years, the quality and volume of built-in loudspeakers of smartphones increased considerably. However, there is a limit of performance due to the size of the devices. In this thesis, we show that it is possible to use multiple modern smartphones to play audio synchronously to boost the sound or increase its range. We thereby create a pseudo hi-fi system. Finally, we develop an application that automatically compensates the differences in audio output latency and achieves synchronous playback of a song using multiple devices.

Contents

Chapter 1

Introduction

1.1 Motivation

Smartphones have replaced regular cell phones and as they offer multimedia features in addition to the ability to communicate. The peripherals, like built-in cameras, GPS, sensors and loudspeakers, improved over the years. However, the built-in loudspeakers have room for improvement and using external loudspeaker boxes has drawbacks like additional cost and weight to carry around. Therefore, we propose to synchronously play music on multiple smartphones in order to improve the quality and range of the sound.

In this thesis, we show the feasibility of connecting multiple phones and tablets and playing audio synchronously on them. Instead of using external loudspeakers for a phone, one can connect to other devices and use the built-in speakers for playback. This allows for instance to cover a room with music by distributing the devices in the room.

To play audio synchronously, one has to find a method to compensate the difference in output latency, which can be in the range of several hundred milliseconds (see Section 2.2). On top of that, the problem is complicated by the non real-time nature of the operating system.

1.2 Related Work

There are apps on Android that implement audio streaming such as *SoundSeeder* [1]. However, synchronization is not automatic, the user has to adjust the offset between devices manually. In contrast, the app developed in our thesis determines the latency difference on its own and the music is not streamed, but rather the song is transferred beforehand and the start of playback is then coordinated automatically.

On iOS, there are several applications that allow synchronous playback, for instance Seedio [2], TuneMob [3] and Whaale [4].

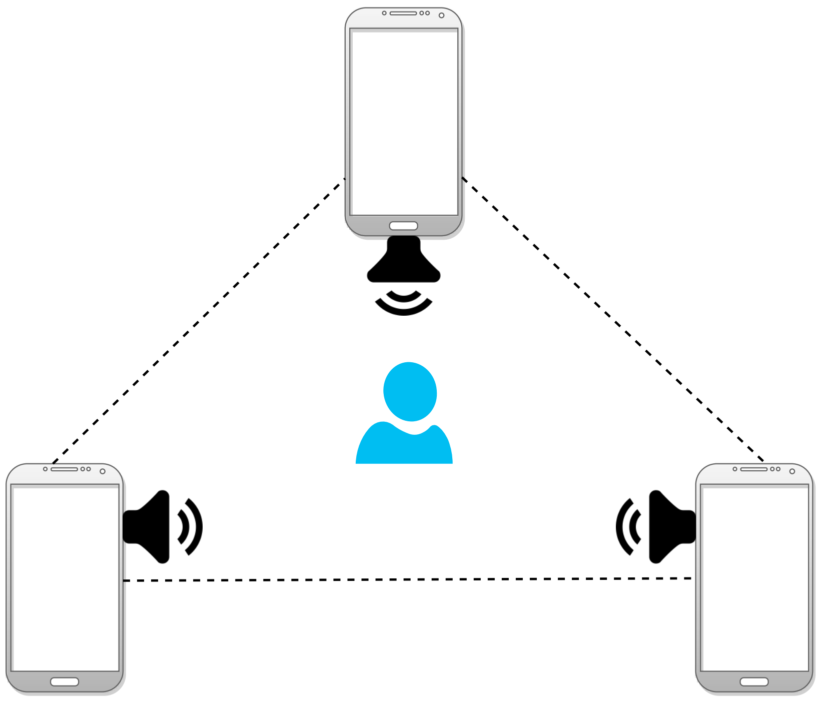

Figure 1.1: Connecting multiple smartphones to play music synchronously.

Google has measured the audio latency on several devices [5]. The sum of output- and input delay (round-trip latency) was determined in a similar way as in our application (see Section 3.3). They used the Larsen test, which works the following way: The phone records audio and plays it immediately, which results in a feedback. By creating a tone externally, it is possible to measure the time between the feedback pulses, which is equal to the round-trip latency.

1.3 Outline

The goal was to develop an Android app (called *SynBa*), that allows us to connect the devices and calibrate the delays automatically to ensure synchronous music playback. A design choice was to make the application independent from the internet. After establishing inter-device communication (see Section 3.1), a way to create a common sense of time was developed (see Section 3.2). Furthermore, a method had to be found to eliminate the differences in audio output latency between devices (see Section 3.3).

Chapter 2

Background

2.1 Operating System

The task of synchronizing actions between devices is complicated by the fact that mobile operating systems like Android are not designed to be real-time. On top of the Linux kernel, the applications run on a virtual machine for java byte code (Dalvik VM or ART). Delays from garbage collection are unpredictable and not in the control of the developer.

Android runs on a heterogeneous set of devices from different manufactures. Hence, the application's desired behavior is not guaranteed, especially when using peripheral functions like communication, audio output and input. While Android provides an interface for the developer to use hardware components without worrying about lower-level implementation, it also creates a black box that makes it hard to predict the behavior of the device. Using hardware like audio output via the API results in a chain of function calls, because the software stack has to be traversed down [6].

2.2 Audio Output on Smartphones

When playing audio files on smartphones, one has to deal with latency. According to the Android documentation [7], the main contributors to audio delay are:

- Application (delay before writing the samples to the buffer)
- Total number of buffers in pipeline and buffer size in frames
- Additional latency after the app processor, such as from a DSP

It is assumed that latency added by the analog circuitry can be neglected. Note that the buffers need to be large enough to provide sufficient time to process the frames, otherwise the audio starts to "stutter" (buffer underrun or overrun).

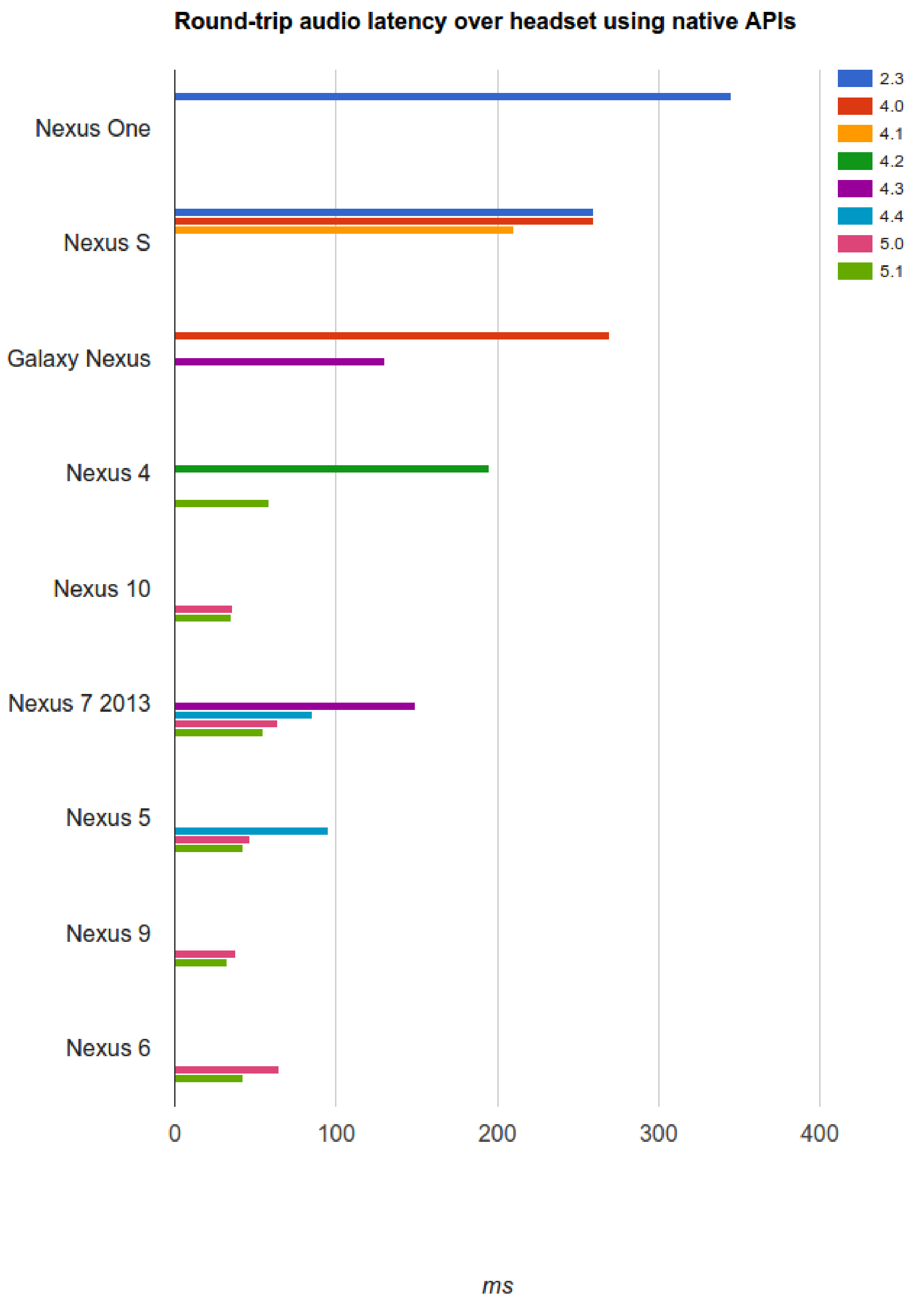

Figure 2.1: Round-trip audio latency, as published by Google [5]: Sum of output and input latency. Different devices have latencies in the range of several hundred milliseconds. Note that newer API versions implement low latency audio.

Chapter 3

Implementation

3.1 Communication

3.1.1 Wi-Fi Direct

We select Wi-Fi Direct as the communication standard, as it allows multiple devices to exchange data without a central access point (router). The group owner device acts as a virtual wireless access point. Furthermore, Wi-Fi Direct provides typical Wi-Fi speed and range, which allows us to quickly transfer the audio files between devices.

In Figure 3.1 one can see the connection setup process. After the application has initialized a broadcast receiver, which allows receiving intents from the operating system, we enter the idle state. When the user presses "*Search Peers*" on the user interface, the program begins discovering other devices. As soon as the list of peers has changed, the app is notified by the Wi-Fi Direct framework. We can now request the list of devices, which activates a callback when the request was processed. This gives us a list of available devices in the network to which we can connect (including the group owner).

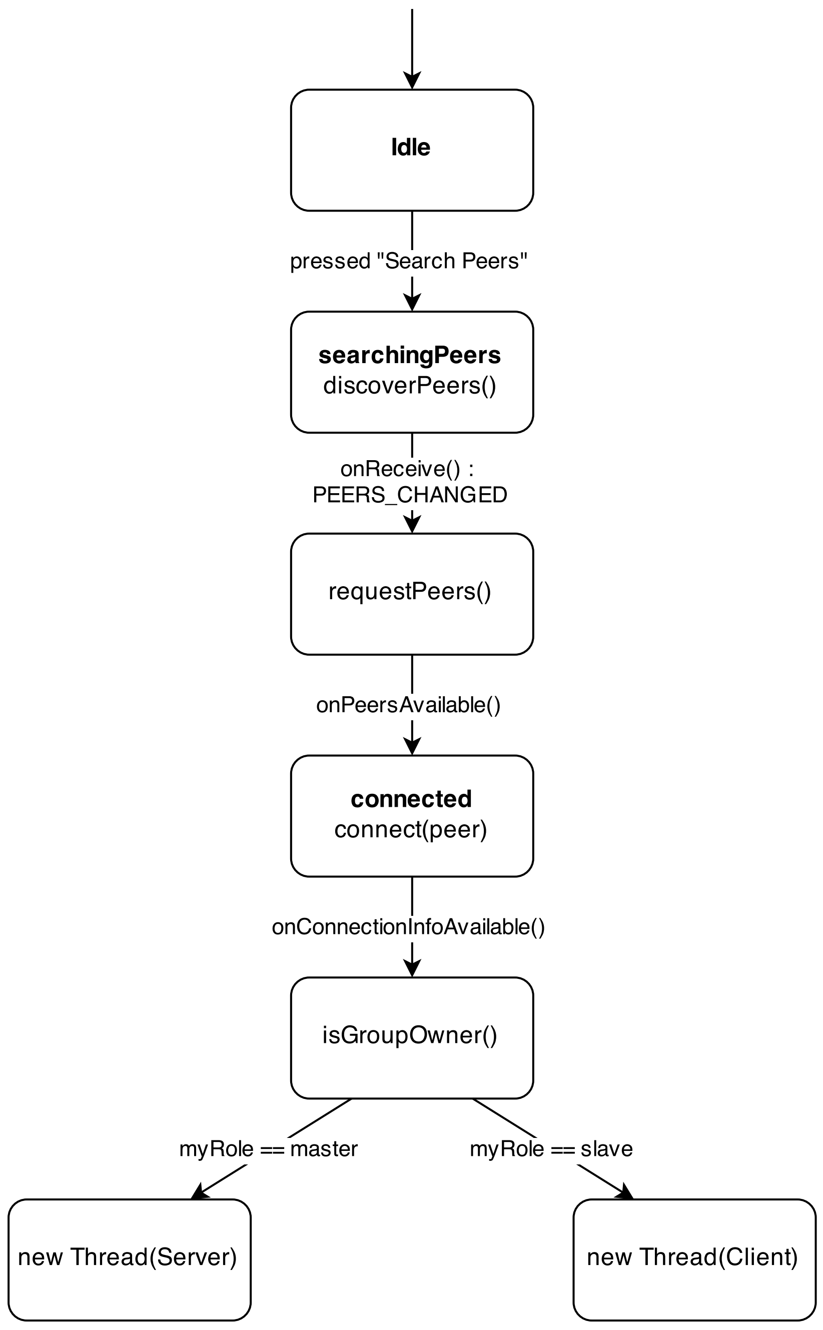

Figure 3.1: Wi-Fi Direct Setup: When the "*Search Peers*" button is pressed, the function *discoverPeer()* is called. When the list of peers has changed, the intent *PEERS_CHANGED* is received. Calling *requestPeers()* allows us to ask for the list of devices, which activates the callback *onPeersAvailable()* when the request was processed. Depending whether the device is the group owner or not, the role of master or slave is assigned.

3.1.2 Communication scheme

The event driven communication scheme of the application is depicted in Figure 3.2. The group owner of the network acts as a master. He runs a server thread, accepting connection requests from slaves via a server socket. Each connection is processed by its own communication thread.

Only a slave can start the communication. For each command that needs to be sent, a client thread is created. After the connection to the server socket has been established, the command and additional arguments are written to the output stream of the socket. The argument of the command is used for sending additional information. On the server side, the responsible thread reads the command from the input stream and replies depending on the type of the command. For example, a slave could be asking for the master's time and the master would reply with a message containing the time as the argument. Before closing the socket, the thread posts a message to the handler of the main thread, informing the application that a command has been received. The same procedure executes on the client side.

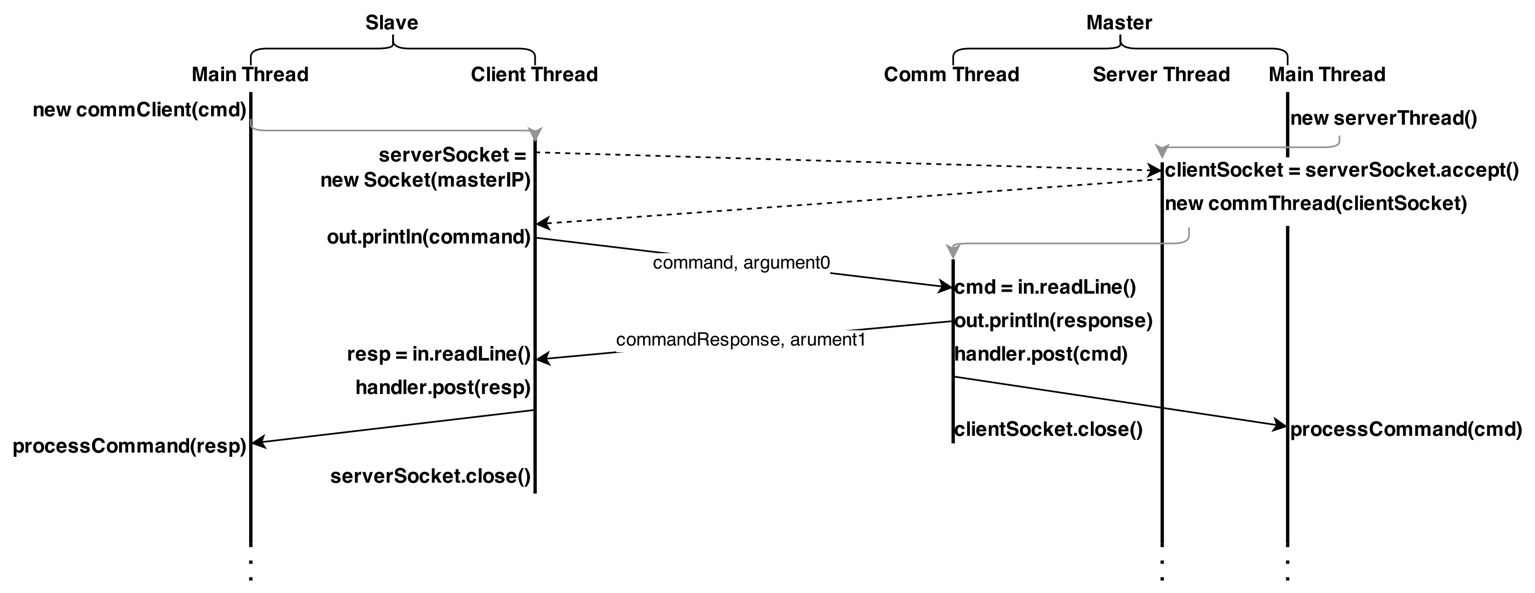

Figure 3.2: Communication Scheme: The master's *server thread* is accepting connection requests from slaves via a server socket. For each established connection to a slave, a thread is created (*Comm Thread*). When a slave wants to send a command, he creates a *client thread*. Connecting to the *server socket* allows him to write commands to the output stream of the socket (*out.println(command)*). An example for a command and argument: *"timeResponse, 123456789"*. The method *processCommand()* is called to notify the application that a command has been received (the main thread is informed by posting a message to its handler).

3.2 Clock Synchronization

In order to synchronize audio playback, the different devices need to have a common notion of time. One approach is to use the network time protocol (NTP) for clock synchronization. However, because NTP uses time servers, the system wouldn't work without an available internet connection. Therefore, we decided to implement an algorithm that only uses the wall clocks of the different devices. The Android operating system allows us to retrieve the nanoseconds since last boot, including time spent in deep sleep.

When a slave is connected, it sends a message to the master and stores the time in the variable t_{out}. The master announces his current time t_m. When the slave receives the message at time t_s, it computes the the round trip time $RTT = t_s - t_{out}$. When we make the assumption that the delay of sending a message is symmetrical,

$$\tilde{t}_m = t_m + \frac{RTT}{2} \tag{3.1}$$

is a good approximation of the master's time that corresponds to t_s (see Figure 3.3). The procedure is repeated several times to accumulate a set of measurements. This way we can determine a tuple

$$(t_s, \tilde{t}_m) \tag{3.2}$$

which allows the slave to have common notion of time with the master.

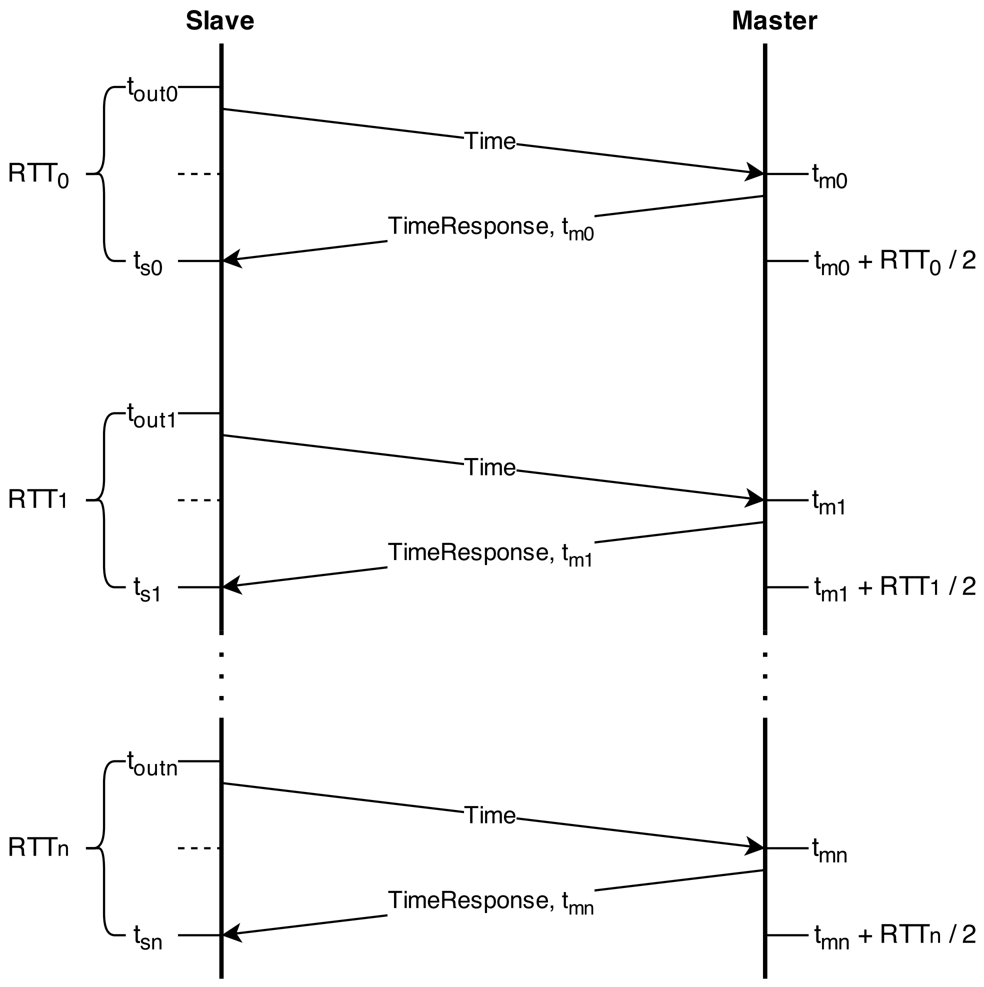

Figure 3.3: Clock Synchronization: The slave requests the time by sending the command *"Time"*. The master responds with *"TimeResponse, t_m"*, where t_m is the master's current time.

3.3 Audio Latency Correction

Determining the absolute audio output latency programmatically is not feasible as it is highly device dependent. We thus develop a method to compensate the difference in output latency between the devices.

On the slave device, the app starts to record audio and then plays a sine sound at a certain point in time $t_{s,start}$. It then detects the sound at time $t_{s,detected}$ in the recordings (see Section 3.3.1) and calculates the so called *audio round-trip latency*

$$T_1 := t_{s,detected} - t_{s,start}. \tag{3.3}$$

In a second phase, which is depicted in Figure 3.5, the slave asks the master to play a sine wave. He will respond, announcing the point in time t_{start} when he will play a sine wave. The slave then records audio and detects the sine at time $t_{detected}$, which allows to compute

$$T_2 := t_{detected} - t_{start}. \tag{3.4}$$

This way, we determined two time periods T_1 and T_2. They consist of the following delays (see Figure 3.4):

$$T_1 = T_{out,1} + T_{in,1} \ (+T_{sound,1})$$
$$T_2 = T_{out,2} + T_{in,1} \ (+T_{sound,2})$$

Where $T_{out,1}$ and $T_{in,1}$ are the audio output- and input latencies of the slave, $T_{out,2}$ is the output latency of the master. We can neglect the propagation delay of the sine waves $T_{sound,1}$ and $T_{sound,2}$. It can easily be shown that:

$$T_D := T_1 - T_2 = T_{out,1} - T_{out,2} \tag{3.5}$$

Therefore, T_D is the difference in audio output latency between the slave and the master. This procedure is run once, when two devices connect for the first time.

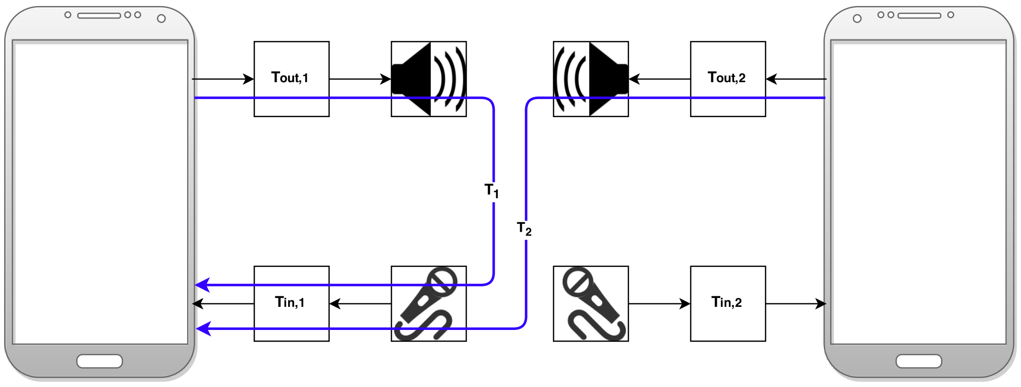

Figure 3.4: Audio Latency Correction

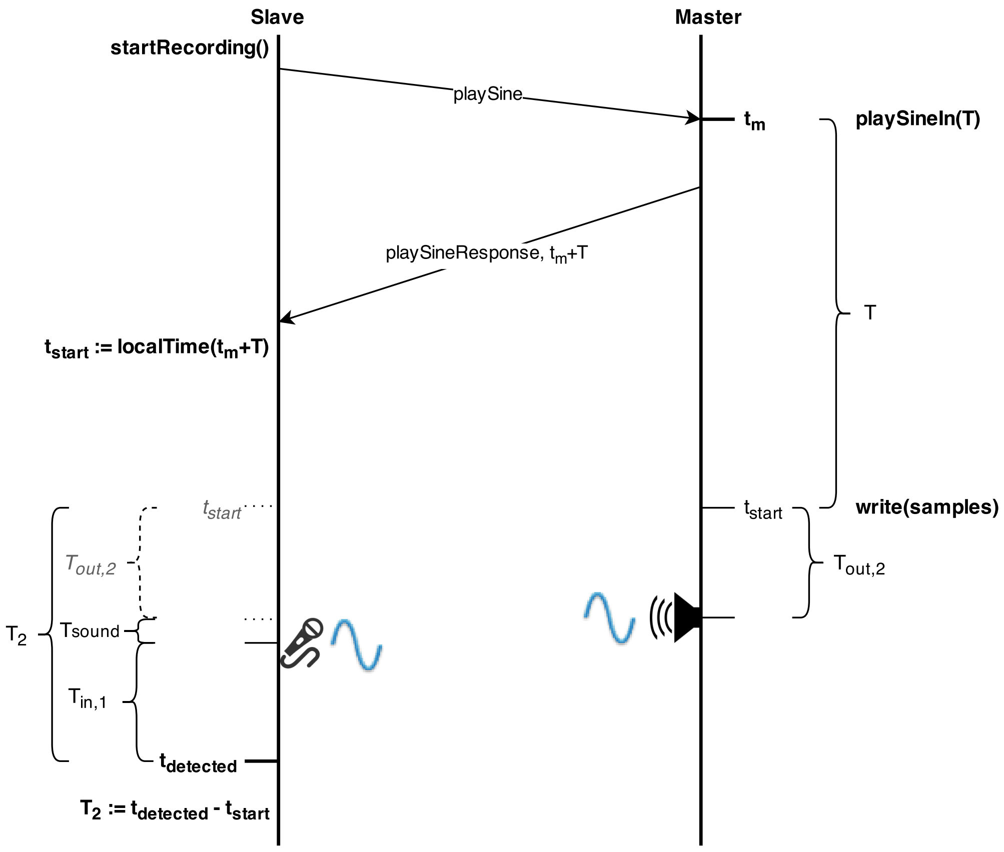

Figure 3.5: Audio Latency Correction: The command "*playSine*" is sent to the master. He will respond with "*playSineResponse*, t_{start}", announcing the point in time t_{start} when he will play a sine wave. This allows the slave to compute T_2.

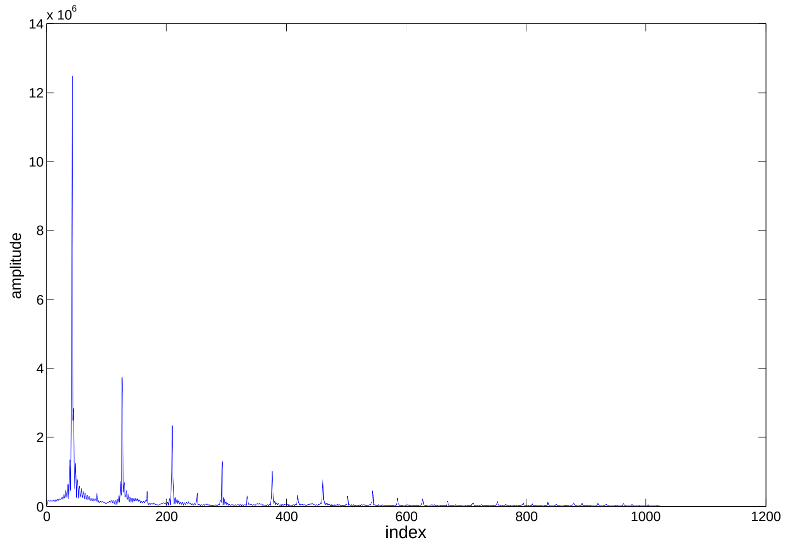

Figure 3.6: FFT of a recorded sine wave

3.3.1 Sine wave detection

As described in Section 3.3, we use sine tones to measure audio latency and thus a method for detecting them is required. Whenever the audio recording buffer is filled, we transform it to the frequency domain using a fast Fourier transform library [8]. Let us denote the transformed buffer as $\hat{b}_k$ with $k = 0 \dots N/2 - 1$. We then find the maximum absolute amplitude $\hat{b}_{k_{max}}$ at position k_{max}. If $\hat{b}_{k_{max}}$ exceeds a certain threshold relative to the mean of $\hat{b}$, we have found a tone. The frequency of the sound is given by

$$f_{max} = \frac{k_{max}}{N} f_s \tag{3.6}$$

with N being the length of the FFT and f_s being the sampling frequency of the recording. In Figure 3.6, one can see the FFT of a buffer of size 1024 with sampling frequency 44100 Hertz. The FFT library only returns half of the transformation, as it is satisfies the symmetry condition for real input data $(\hat{b}_{N-k} = \hat{b}_k^*)$. With the maximum at $k_{max} = 43$, we can compute $f_{max} = \frac{43}{2048} 44100 \approx 926 Hz$. The actual played sound was of frequency 900 Hertz.

When f_{max} is in an interval around the frequency we searched for, we have detected it. In a next step the original buffer is divided into chunks of equal size and the sine detection algorithm is applied on each of them in order to determine the start of the sine within the buffer.

3.3.2 Playback Timing

Using the output latency difference T_D obtained as described in Section 3.3, we can synchronize audio playback on multiple devices. The procedure can be seen in Figure 3.7.

Each connected slave device i has determined its latency correction T_{Di} relative to the master. The slave sends a command to the master, announcing that he is ready for playback. All communication threads on the master device wait until the corresponding slaves are ready. To start playback, all threads are notified and they transmit t_e to the corresponding slave, where t_e is a point in time in the future.

The slave receives this message at time t_s. He plans to start playback in

$$t_e - t_s + T_D \tag{3.7}$$

nanoseconds, so that at time $t_e + T_D$ the first music samples are written into the output buffer. Because T_D approximates the difference in output latency between the master and the slave (positive or negative), the playback starts simultaneously for human ears on all devices at time t_e.

The waiting time is not implemented by delaying the audio player's thread execution. Instead, the thread is started immediately and the number of silent samples needed for the given waiting time is computed. After the required amount of silent samples have been played, the thread starts writing the actual song samples into the buffer. This method is more accurate than posting the runnable delayed into the threads message queue, because accurate timing of the thread scheduler isn't guaranteed.

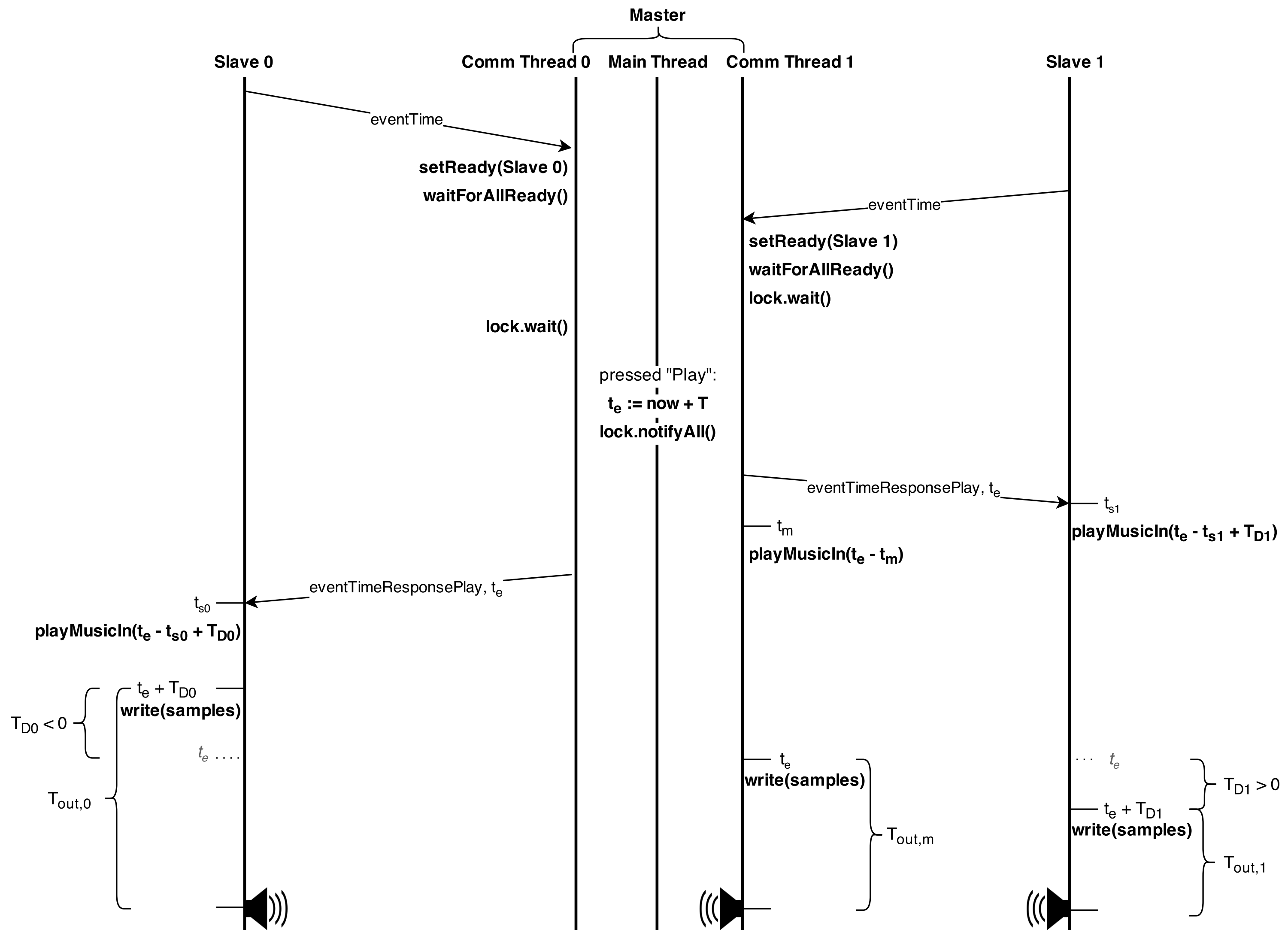

Figure 3.7: Playback Timing: When a slave is ready for playback, he sends "*eventTime*" to the master. On the master device, all communication threads wait until all slaves have sent this command. When the user wants to start playback on the master device, all communication threads are notified and they send "*eventTimeResponsePlay,* t_e" to the slave, where t_e is the point in time when the master will start writing audio samples. T_D allows to compensate output latency difference between devices by starting to play earlier or later than the master.

Chapter 4

Evaluation

4.1 Clock Synchronization

Like described in Section 3.2, our clock synchronization method yields tuples (t_s, t_m). The slave's time t_s corresponds to the master's time t_m. When we repeat this procedure n times, we get a set of tuples $(t_{s,k}, t_{m,k})$ with $k = 1 \dots n$. These allow us to analyze the variance of the method by transforming all measurements to the same time reference by doing the following:

Given a reference tuple $(t_{s,1}, t_{m,1})$, we compute for each tuple a new $t'_{m,k}$:

$$t'_{m,k} = t_{m,k} - (t_{s,k} - t_{s,1}) \tag{4.1}$$

In other words, we subtract the time that has passed since the reference measurement from the current master time.

In Figure 4.1, we can see that this method produces master times with a standard deviation of about 5 milliseconds. Every second a new master time was determined. In the top left plot, we can observe a clock drift between the two devices of about 8 milliseconds per 1000 seconds. To compensate this effect, the app regularly re-synchronizes using the same technique.

In the plots at the bottom of Figure 4.1 we can see the distribution of the round trip times of the master time requests (like described in Section 3.2). The round trip times are likely to be network- and device-dependent. It is interesting to see that for the tested devices, the variance in clock synchronization is orders of magnitude smaller than the difference in audio output delays.

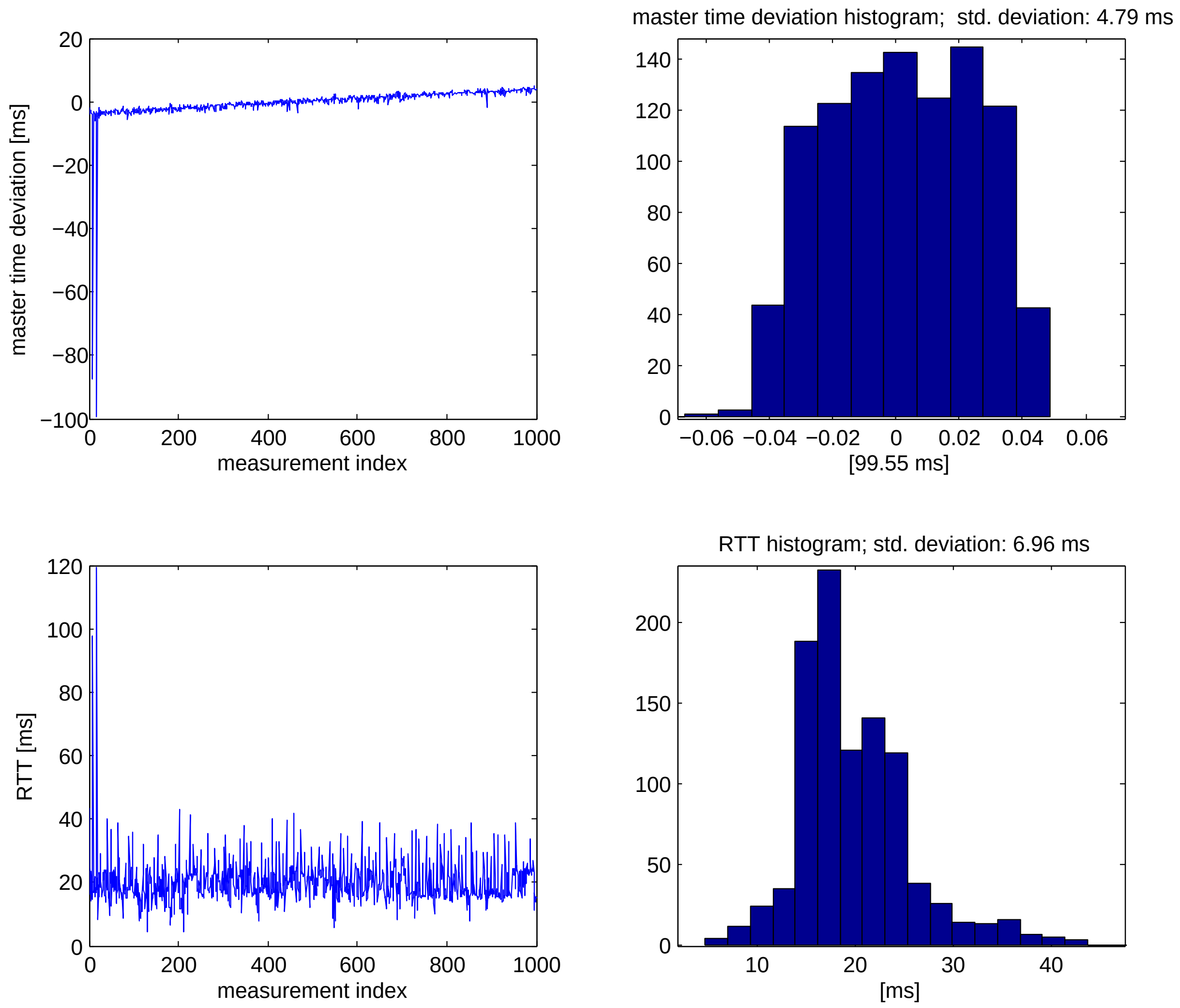

Figure 4.1: Master time deviation and RTT: The distribution of the master times in the top right are not Gaussian, due to the clock drift between the devices (see top left plot). In the bottom, one can see the distribution of the round-trip times, i.e. the duration when the slave requests the time until the response from the master was received.

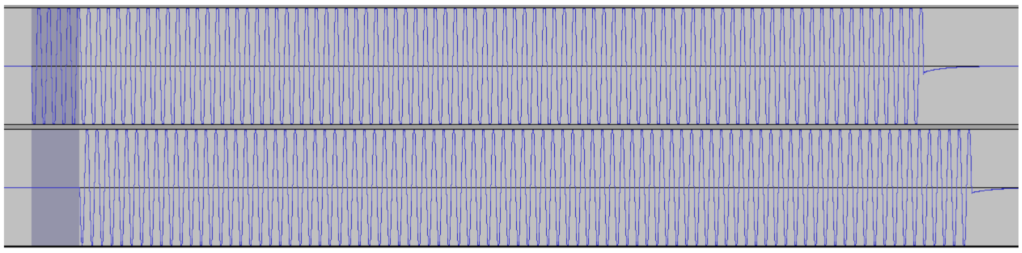

Figure 4.2: Recording of two phones playing a 900 Hertz sine tone. The selection at the start shows the offset of 5 milliseconds.

4.2 Latency Correction

4.2.1 Test setup

To test the quality of synchronization between devices, the following approach was taken: A combination of 3.5 mm to cinch cables allowed to connect one phone to the left channel of the line-in input of a computer and the other phone to the right channel of the same audio input. This way, we could record the audio of both smartphones simultaneously. In Figure 4.2, one can see the recording of two devices playing a 900 Hertz sine tone of 100 milliseconds duration. After a automatic latency correction of 92 milliseconds, the offset is 5 milliseconds (blue selection).

Additionally, we wrote a MATLAB script, that automatically detects the offset between the two channels. The results are documented in the next section.

4.2.2 Results

Tests showed that the approach to coordinate the start of playback, as described in Section 3.3.2, resulted in offsets of very low variance. In Figure 4.3, we can see an example of a series of measurements. A series of sine sounds were played synchronously on two devices and the MATLAB script detected the offset between the two devices.

With a standard deviation of just under one millisecond, the offset can be regarded as almost constant. However, the application determined a latency correction of 98 milliseconds, which didn't completely eliminate the difference in output latency in this case. As one can see in the figure, the offset has a mean of about 24 milliseconds, which was not audible. Measurements between other pairs of devices showed offsets below 10 ms.

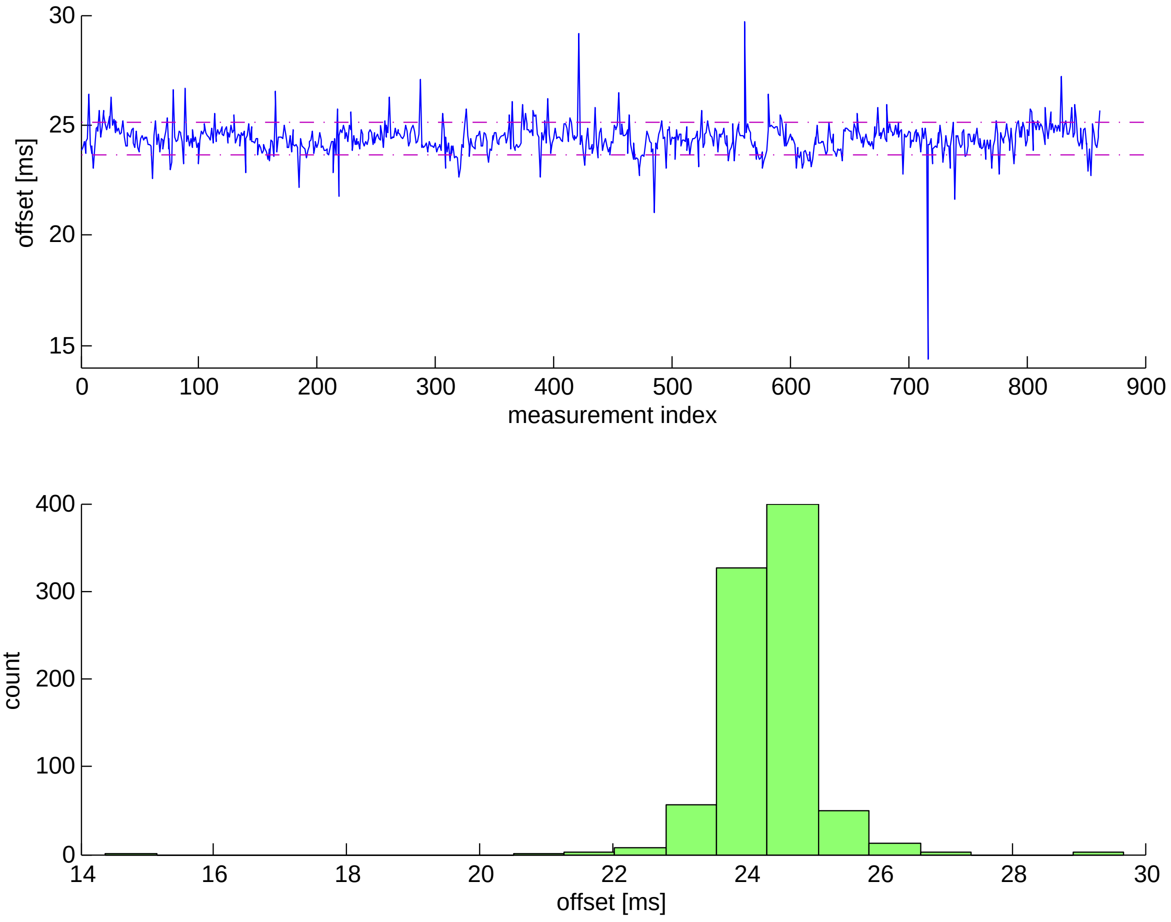

Figure 4.3: Remaining offset between a Galaxy Note 10.1 and a Xperia Z3 after a latency correction of 98 milliseconds (test series of 862 sine sounds).

Chapter 5

Conclusion and Future Work

Our implementation approach produced a good synchronization of audio playback on four devices at once. It has yet to be evaluated what the maximum number of connected devices is and if the app works for the most part in the heterogeneous field of smartphones and tablets. The results of the latency correction procedure (Section 3.3) are saved in the cloud, which eventually can give further insights when enough data was gathered. The values could be used to construct a graph of latency differences between device models (see Figure 5.1). When the sample size is large enough, the application could fetch the latency correction from the cloud. It would also be possible to apply a least squares algorithm on such a graph. For example, in the graph in Figure 5.1, the sum of edge weights on two paths are not equal.

We noticed that using Wi-Fi direct caused problems on some devices. The connection was unstable or the device was not able to connect at all. We successfully connected four devices which played music synchronously. However, we do not know how the app scales with the number of devices. According to the Wi-Fi Alliance [9], the number of devices in a Wi-Fi Direct group is expected to be smaller than with a traditional access point and it is not guaranteed that a device supports multiple connections.

An approach would be to use another way of distributing information about playback timing. Possible options would be using the cell phone towers, GPS or a web server as a central coordinator.

Our approach to determine audio output latency worked well, but it can still be improved. It is also possible to think of other algorithms to determine or compensate the output delay.

Our app allows the master to choose a song and all the devices start playing simultaneously. If the song is not already on all devices, it is automatically distributed to all the slaves. The quality is synchronization is good, offsets between devices were not audible. Nevertheless, there is still room for improvement, including usability and additional features like displaying meta information of songs or selecting entire albums for playback.

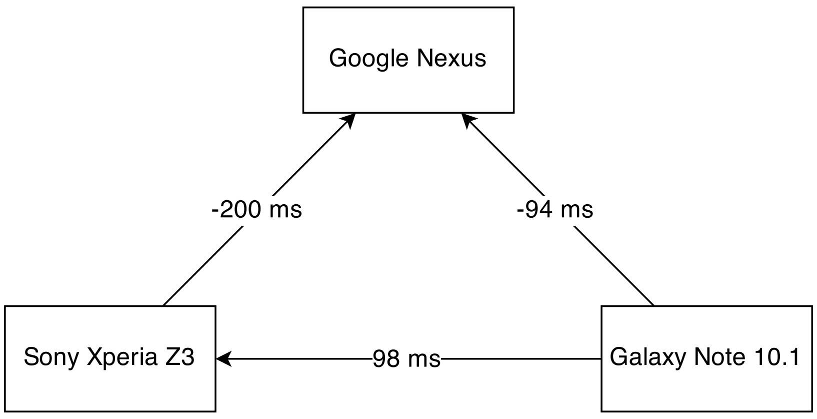

Figure 5.1: Graph containing latency differences: Arrow denotes a slave-master relation with the determined latency correction marked on the edge. Note that $98 - 200 = -102 \neq -94$. One could apply the least squares algorithm on paths with the same start- and end node.

Bibliography

[1] : Soundseeder. (May 2015) http://soundseeder.com.

[2] : Seedio. (May 2015) http://seedio-app.com/.

[3] : Tunemob. (May 2015) http://www.tune-mob.com/.

[4] : Whaale. (May 2015) http://www.whaale.com/.

[5] : Android open source project: Audio latency measurements. (May 2015) https://source.android.com/devices/audio/latency_measurements.html.

[6] : Android open source project: Interfaces: Audio. (May 2015) https://source.android.com/devices/audio/index.html.

[7] : Android open source project: Contributors to audio latency. (May 2015) https://source.android.com/devices/audio/latency_contrib.html.

[8] : JTransforms library. (May 2015) https://sites.google.com/site/piotrwendykier/software/jtransforms.

[9] : Wi-fi alliance: How many devices can connect? (May 2015) http://www.wi-fi.org/knowledge-center/faq/how-many-devices-can-connect.

Appendix A

The User Interface

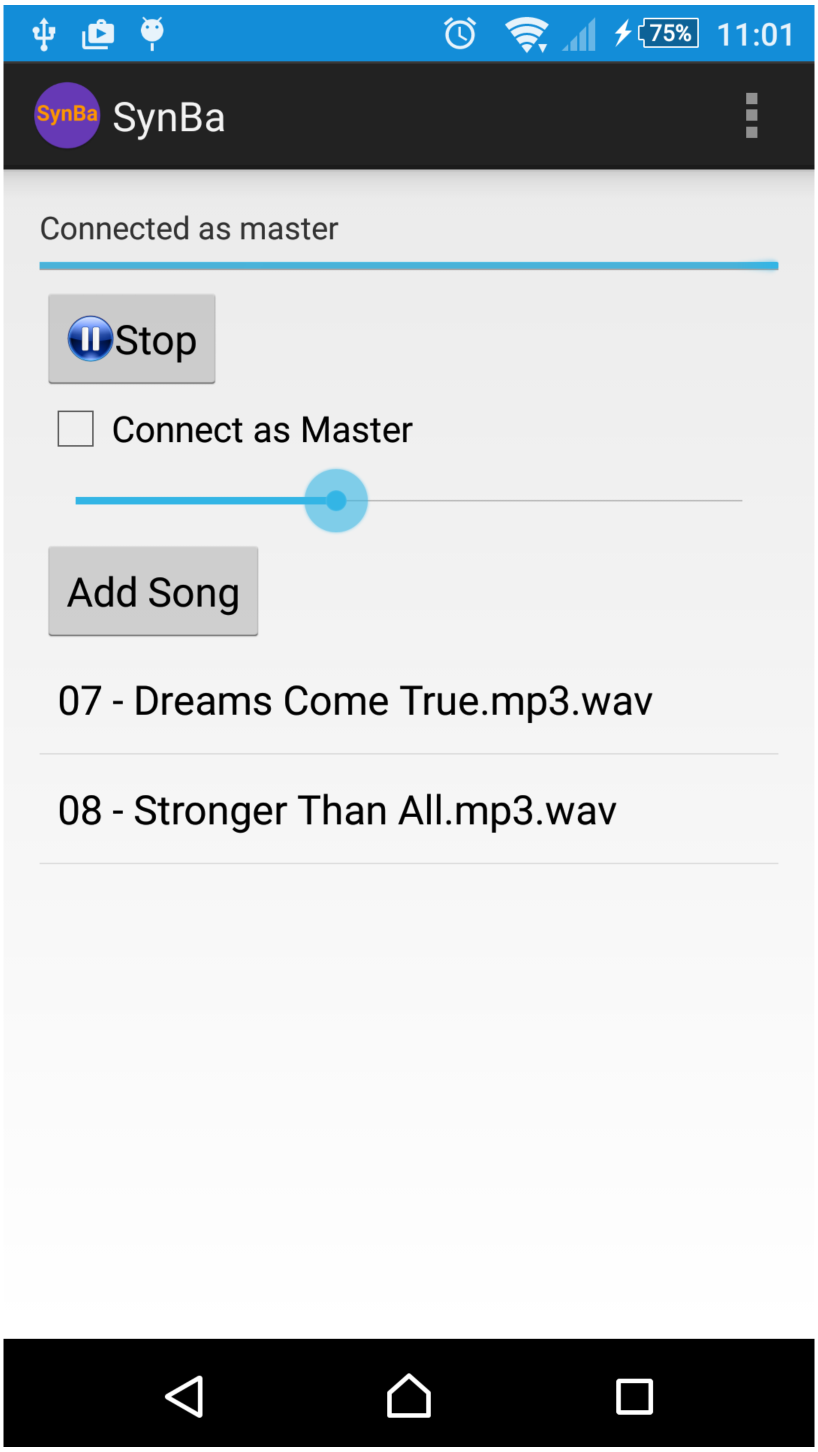

Figure A.1: The app, SynBa