Universiteit Leiden
The Netherlands

Opleiding Informatica

Assessing the fitness of web-applications within the context of mobile phones, on performing spatially distributed, co-located, collaborative, audio-related activities.

Jeroen van Tubergen

Supervisors:
Edwin van der Heide & Prof.dr.ir. Fons J. Verbeek

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

26/08/2020

Abstract

Mobile devices have advanced capabilities for audio-related activities, such as to record and to play music. Many software applications successfully integrate audio into their software, and a multitude of applications exist in which audio plays a central role, for example audio-creation or audio-listening software.

However, these applications focus on individual audio performance, while when considering music, one of the most prevalent uses would be listening and playing music together.

Even though each one of us carries along their mobile phone around, and is using it for listening, there isn't a scalable solution available for any such audio activities to be done together. In other words, there are no audio applications available to successfully draw either beginners or professionals into the digital landscape of interactive, co-located audio-related activities with the aid of their mobile phone.

In this thesis we research whether a typical modern mobile phone could perform sufficiently enough, to allow it's user to participate in a collaborative audio activity, in real-time.

We research available technological solutions, their use-cases, and afterwards seek how they could or cannot be applied in the context of a web-application.

We target a web-application, since this we think would provide the most available software solution. We reason about the inherent latencies that could influence an audio-application in it's performance, and we develop a solution with the aid of a signal detection approach, which consists of cross-correlating recorded audio material against target audio signals, to afterwards resolve with these measure results relevant, device-related latencies.

We test our approach in a proof of concept and show that a web-application is able to successfully calibrate. However, it fails to meet accuracy requirements to achieve acceptable synchronized audio performance, due to the inherently available inaccurate of web-application audio scheduling.

We define a benchmarking approach to assess an audio activity for it's fitness for implementation within a web-application.

We conclude that mobile devices are capable to participate in audio activities in real time. Web-applications however, could offer a more complete solution: Both on improving the development platform to simplify and improve the development process, as well as the performing qualities.

Contents

List of Tables

List of Figures

1 Introduction

Mobile devices are everywhere, and so is music. Yet, it is hard to do a jam session together by just using a mobile phone. Whether this is due technological barriers, considering such a thing to be redundant, or just never the thought of using a phone in such a way, there seems to be a focus on individuality in the current state of audio applications. Which seems to be out of line with what we see in the gaming industry. Even though audio applications could and should highly interactive and collaborative interactive aspects, as with gaming, the gaming industry made real-time collaborative interaction an integrative part to it's application for many years already. I wonder if there is a way to somehow create the magical and unexpected experiences audio and social interaction could create, in either every day life, or on specific occasions, with a mobile phone. Therefore we will consider in this thesis audio-related activities where musical interaction and playing together plays a central role.

A common and straightforward audio-related activity, is playing the exact same music by different participating musicians, in this case the musicians are mobile phones. Even though the activity itself is simple, achieving proper synchronized behavior requires a lot of collaboration to play on time all together. Testing how well a mobile web-application could provision this activity would therefore seem to be a proper test to assess how fit mobile web-applications are to perform spatially distributed, co-located, collaborative, audio-related activities.

Therefore, the research question of this thesis is ***"Can we achieve automatically, synchronized audio performance in a web-application, focused on the context of mobile phones?"***. With synchronized we mean song playback differences no larger than 10 milliseconds, with automatic we mean that there isn't user interaction required to synchronize, and with a web-application we aim at up-to-date web-browsers (interpreting JavaScript, HTML and CSS).

After considering related technology to achieve this, we will explore the seemingly most viable approach by synchronizing based on signal detection with the aid of microphones and speakers, optionally with the integrated speaker and microphone of the mobile phone itself. The challenge is to tackle the inconsistent and highly variable audio output latencies different mobile phones in combination with their audio output target can have, however there isn't a software approach to obtain knowledge on this latency without physically measuring it. In addition to synchronizing, it allows for devices to recognize where other devices are located in space.

As we elaborate on how performance requirements are crucial to determine how well a web-application would be able to provision an collaborative audio-related activity, we can directly state that using the microphone and speaker in the frequency domain we can hear, carries along certain fundamental qualities that the alternatives Bluetooth and WiFi aren't able to offer. These quality can basically be derived from the given that both microphones, and our ears, are able to sense the exact same signal frequency range. And besides perceiving, both speakers and our vocal/instrumental tools are able to produce these signals. This allows for a dialogue between devices and humans in the domain of audio.

Far more audio-related activities may exist besides this specific activity we choose for a fitness assessment. In particular we are interested in what audio-related activities could be performed with mobile web-applications. Besides assessing the performance capabilities of the current technological stack of a web-application running on a mobile phone, we research the different types and classes of spatial and collaborative activities. However, as we consider different types of classifications for activities, we note how we aren't able to relate a certain classification of audio-related activity to certain performance requirements, and the most suitable approach would be to inventorize the performance requirements per give activity to conduct whether an mobile web-application could provision it's usage.

1.1 Overview

In the first section 2, we explore existing technological solutions that attempt to tackle the challenge of synchronized audio playback, which focuses particularly on sound reinforcement systems. We focus on technology for audio sound-reinforcement, since playing the same piece of music synchronously is a straightforward action related to audio, yet still requires adequate performance and coordination to be performed well. Both on large, specialized, commercial scale, with a study of the proprietary solution Q-LAN 2.1.2.1, as well as consumer-oriented non-live playback systems, with a study on Volumio, build on Shairport Sync 2.1.3. As last we consider software for Android and iOS in 2.1.4, for which the available mobile apps for both platforms don't offer a performant solution. Next up, we research how it can be that there isn't a performant solution on the market yet for mobile phones, by considering the mobile platform on it's audio-related functionality in 2.2. In this subsection, we first consider the audio latency path in Android and the latency of Bluetooth data communication latency, and afterwards we consider how in theory precise audio output latency feedback could be provided in 2.2.3.

With this background information, and in particular the knowledge obtained about the audio path latency in Android phones, we reason about how to build software to synchronize mobile devices for audio playback in section 3. First, we come up with a procedure on how devices could recognize themselves and others with the aid of signal detection 3.1. Secondly, we build a model on how audio latency could be derived with these measure results; in other words how to deduce the calibration performance 3.2. And finally, we describe a topological model to calculate the calibration performance once the system has incomplete information due restrictive detection capabilities in 3.3.

With the theory ready about how to synchronize two or more mobile devices, we develop a proof of concept in section 4, in which two devices first calibrate with each other and afterwards attempt to play back audio synchronously together. Here we in addition do experiments with the proof of concept in 4.5 and elaborate on it's results in 4.6.

We do an inventorization on relevant performance metrics to benchmark audio-activities, and consider the variety of categories of spatially, co-located, audio-related activities in 5. Additionally, we attempt to relate performance metrics to categories, with an aim to conclude what type of audio-related activities the current state of web-applications could support.

In 6 we discuss our findings and choices per section. In 7, we describe future research for both

further development in the technique in a front-end manner, in a more underlying manner of research to web-browsers and low-level implementations, and in consideration of audio activities. In 8 we conclude, in particular about the audio latency path, the bottlenecks in web development and the overall web-application performance.

This bachelor thesis is written at the LIACS under supervision of E. v.d. Heide, to who I am thankful for the patience and collaboration in the development process and the writing of this thesis.

2 Background

We explore the overall technology available to achieve an audio reinforcement system, and provide some products within certain technology categories with certain interesting functionality/features. Afterwards, we explore Android, Firefox, and Bluetooth in the context of audio, and focus on how each component of the technology stack can introduce latency to the audio output latency of our web-application.

2.1 Sound reinforcement systems

First we mention the default wiring of speakers to an audio system. Afterwards we look at audio networks to improve on scalability, and last we look at software to convert computer and mobile phones, respectively, into an audio node of a sound-reinforcement system.

2.1.1 Wired audio systems

The classic approach to achieve an audio system is wiring multiple speakers to a single audio station. Audio data is streamed from the audio station to each speaker separately as an analogue signal (with a phone connector), or as a digital signal (with e.g. a S/PDIF connector). Digital streaming has the advantage that the original audio signal can be send clean without any noise or spectral changes introduced, nor requires additional amplification in case of large traveling signals. However, it has the disadvantage that each speaker requires a digital-to-analog converter (DAC) to convert the digital signal into analogue audio, which makes the speaker both more complex and introduces additional latency in outputting the audio.

2.1.2 Audio network systems

The default home system could be scaled by a large extend without introducing any novel audio signal transportation technique, yet rather increasing only the amount of wires. For certain situations wiring is sufficient, but for certain situations cabling might not be the most convenient approach to scale up the system. Obviously, cables require to be physically connect from the audio workstation to each separate output. In case the speakers of a single system are distributed over larger areas, wiring becomes a burden to quickly set-up, and any reduction in wiring can be a reduction in e.g. installation costs or maintenance. Therefore, for large-scale audio installations, such as in theme parks and stadiums, the need rose for audio networking, since it offers a more flexible, optimized audio signal distribution system [1].

Audio networks introduce another level of complexity on top of the digital-to-analog conversion in digitally transported cabling, since we have to interpret and respond to network signals. As the network choices are relevant for the audio networks we will discuss, and to better grasp the variety of choices, we have outlined the default approach on how to categorize network functionality as defined by the ISO (International Organization for Standardization) OSI (Open Systems Interconnection) model, in table 1.
As mentioned in [6], lower level functionality could be implement in a higher level abstraction layer, and therefore the separation of concerns per abstraction layer in this model shouldn't be

Level	Name	Function	Description
1	Physical	Binary Transmission	Defines the electrical, mechanical, procedural, and functional specifications for activating, maintaining, and deactivating the physical link. Also, transmission and reception of raw bit streams over a physical medium.
2	Data Link	Access to Media	Defines how data is formatted for transmission and how access to the network is controlled, with the aim to result in reliable transmission of data frames between two nodes connected by a physical layer.
3	Network/Internet	Data Delivery	Structuring and managing a multi-node network, including addressing, routing and traffic control, which consists of: • Provides connectivity and path selection between two host systems. • Routes data packets. • Data prioritization (QoS). • Selects best path to deliver data.
4	Transport	End To End Connections	Provisions transportation between hosts, to obtain reliable transmission of data segments between points on a network: • Segmentation, acknowledgment and multiplexing. • Establishes, maintains and terminates virtual circuits. • Provides reliability through fault detection and recovery. • Information flow control - managing the rate of data transmission between nodes.
5	Session	Managing session communication	Defines the procedure to achieve successfully communication between end-nodes that spans longer time duration. And consists of session establishment, maintenance, termination, authorization, logging.
6	Presentation	Data abstraction	Data abstraction of objects that may require system-specific evaluation, yet aim to provide system-independent interpretation. Such as images, audio, files, etc. Simplifies data sharing. The web describes it consists of data conversion, character code translation, compression, encryption and decryption.
7	Application	Functionality abstraction	High-level API's that are specifically designed for a certain application. They aim to simplify and generalize the development of communication between devices, by interacting with abstracted functionality rather than the underlying system. Such as HTTP for the web-browser.

Table 1: OSI network abstraction layers. Lower four levels description from [2], level 5 description from [3], level 6 and 7 description combined and self-interpreted from [2][4][5]

considered to be an absolute truth.

We will only focus on audio networks which function on top the (level 3) IP (Internet Protocol) network layer or higher, since these networks can already guarantee to be very performant, as we will see during the examination of Q-LAN in 2.1.2.1. Afterwards, we examine consumer-oriented audio reinforcement systems, which guarantee less to no performance guarantees.

2.1.2.1 Q-LAN

Q-LAN is a proprietary audio network solution build by Q-Sys, providing low-latency and high quality audio integration with the aid of an AoE network, and is developed with high-performance, large-scale audio setups in mind. Q-Sys has developed their own mechanisms for device discovery, audio delivery, fault tolerance and clock distribution. The following information is obtained from their Q-LAN white paper [7].

The entire system is guaranteed to function under 2.5 milliseconds latency, which consists of

- Analog-to-digital conversion of captured audio input.
- Transmitting the data to the CPU.
- Processing the audio data in the CPU.
- Transmitting the resulting audio data to audio output nodes.
- Digital-to-analog conversion at audio output nodes.

Q-Sys proprietary CPU can process up of 512 input channels and 512 output high-resolution audio channels concurrently, and guarantees an upper limit in the audio distribution latency to 1 millisecond. They introduce and support multiple fault tolerance strategies, such as a fully parallel deployment of network cabling, switches and/or a second CPU, to switch over to this redundant network without interrupting audio in case any component within the main system fails.

Since Q-LAN functions on top of IP, it can coexist/mingle among st other types of traffic. However, this requires the switches in the network to meet certain Quality of Service (QoS) features. The QoS functionality enables Q-LAN to prioritize time-critical data communication over other network activities.

Audio data is transmit digitally to each end-node in the Q-LAN network. The audio data carries along timestamps of when they are supposed to be output over the system speakers. In order to synchronize the internal time of each end-node with the CPU, the system deploys the Precise Time Protocol (PTP). With PTP, each switch can address how much time it has takes to send data forward to the next network node, and with this information the end-nodes can achieve synchronized clocks with microsecond precision, even if the network has to bridge large distances (multiple kilometers) and/or multiple hops over intermediary switches. However, as noted in their white-paper, more switches/hops imply more latency. Even though internal clocks can be synchronized accurately, the amount of hops should be minimized in order to minimize the network latency.

2.1.3 Consumer-oriented audio reinforcement operating systems

For the consumer at home who wants to listen to music, a high-end business solution such as Q-LAN highly exceeds their requirements. A performant sound reinforcement system, which only has to support audio playback, has to support only a fraction of audio streams, doesn't require the low audio input processing latency, nor low audio output latency, nor the extensive fault tolerance mechanics as with Q-LAN. The only strong requirement is to support accurate audio output coordination between various speakers.

2.1.3.1 Volumio

We will take a look at Volumio[8], an operating system which can be installed on any computer. To describe the software in our own words, it converts the computer into a participating audio node of a music streaming installation on a local area network. This conveniently allows connecting multiple computers either over a network cable or wireless to this network and collaborate in audio playback to thereby create a reinforcement system. Since it runs on existing, default consumer hardware, it is a scalable, cheap, and an accessible solution.

Volumio mentions Raspberry Pi's - small, low-end computers which are considerably cheaper than laptops or PCs - to install their operating system on and connect speakers to the device. The consumer only has to make sure each Raspberry Pi has access to the Local Area Network (LAN), whether that is over Ethernet or WiFi, and it automatically becomes an audio node in the reinforcement system. The devices ensure that one of them is hosting a web server, which enables the entire audio reinforcement system to be accessible through a website, and in such the entire audio system can be controlled through any tablet, phone or computer connected to the local area network with the use of a web-browser.

In contrast to a default approach for which all speakers are wired to a single audio station, with optionally speakers present in different rooms, or even different buildings, this approach removes the burden of wiring through walls, under the ground, etcetera, and from this perspective it can therefore be considered a more scalable and convenient solution.

2.1.3.2 Shairport Sync

Volumio approaches audio calibration in an equal matter as the application Shairport-Sync [9], which is an audio software player that runs on Linux, FreeBSD and OpenBSD. Each computer that participates in the reinforcement system requires to have Shairport Sync installed. One device will be the source, and streams the audio to it's Shairport Sync application. The audio data is transmitted over the local internet, all of the other devices receive the streamed audio data in their Shairport Sync application, and each device prepares the audio to be output over the connected speakers.

The Readme-file defines the amount of wander of the playback synchronization - the time difference of audio emission of speakers of different devices - as "audio drift". The default audio drift tolerance is set to 2 milliseconds ([9], section Tolerance). Also, for any two output speakers the application claims to emit scheduled audio within less than 4 milliseconds deviation (devices

can both deviate by 2 milliseconds from the source). We elaborate on how Shairport Sync works and thus how the audio drift is measured.

The process of communicating audio over the local internet and afterwards preparing it for output takes a non-trivial amount of time. Steps consist of:

- Pack the audio data at the source computer.
- Transmit audio data over the local network.
- Retrieve and unpack the audio data at participating computers.
- Send audio to the respective audio outputs of the computer.

Each step may include buffering, compression, encoding, encryption etcetera, and these network delays and processing times may vary over time.

If a latency peak occurs, such as due network congestion, a device may receive audio data too late and miss out on participating in the audio playback. Therefore, a time margin is introduced in such that each computer has sufficient time to prepare the audio. The system introduces an agreed waiting time, for instance 2 seconds.

For a device to know when exactly the waiting time has passed, it has to know two things: what the local internal clock time is in relation to the source (computer), and the timestamp of the audio sample it has to play. The latter can be transferred along with the audio data itself, the former is resolved by synchronizing the internal clock of the local time with the source time, using a variant of the Network Time Protocol (NTP). The Readme-file mentions the local time, on a Local Area Network (LAN), is synchronized to the source clock to within a fraction of a millisecond with the aid of this NTP variant.

Timestamp and time synchronization itself provides the required knowledge when audio has to be played, yet it is not sufficient to ensure the playback is synchronized. Namely, the underlying operating system may introduce additional delay for audio to be processed *after* the audio leaves the application. If the operating system can provide accurate feedback to the application on how much delay it has introduced, the synchronization is considered to be "full". In such a sense, that Shairport Sync can consider (and anticipate towards) this additional delay to adhere even better to the scheduled time for the audio playback. For instance, the Advanced Linux Sound Architecture (ALSA) is an audio framework which provides an API to user applications to communicate with the audio card drivers. It enables Shairport Sync to obtain accurate feedback about the delay for writing audio data to the audio output node.

We note, that beyond the control of the operating system, the output nodes themselves - the speakers - also take time to process the audio data. For instance a Bluetooth-speaker can have a significant latency to output the audio, which we will further discuss in section 2.2.3.1. Therefore, Shairport Sync offers the functionality to manually compensate for the speaker output latency in case this is necessary.

We began with describing the operating system Volumio, and explained it's approach to sound reinforcement with the aid of the Shairport Sync application. Shairport Sync is a modular solution, and may provide the expected performance if run on any kind of Linux device. Also, if Shairport Sync already performs sufficiently, what is the use-case for an entire operating system dedicated to

do exactly that? The main reason/benefit for the existence of an operating system such as Volumio, is the simplicity of not having to install software yourself, less data requirements due selective, minimal binaries installing, and the possibilities to include a real-time kernel within the product, to give a higher degree of guarantee about the audio data handling speed/process.

2.1.4 Mobile phone audio reinforcement applications

Shairport Sync doesn't run on a mobile device, nor provides an extending software product that can run on iOS or Android. We therefore seek unrelated applications that aim to provide an audio reinforcement system for these mobile platforms. AmpMe [10] attempts to offer a platform agnostic solution by supporting macOS, Windows, Android and iOS, and therefore supports consumer devices both computers and hand-held to participate in sound-reinforcement. We couldn't find any information about the functionality of their product. Therefore, we in addition have taken a look at SoundSeeder [11]. Soundseeder doesn't offer any information either.

From both their FAQs (SoundSeeder FAQ - "How does it work?"[12])(AmpMe FAQ - "Can I use AmpMe without an internet connection?"[13]), SoundSeeder explicitly mentions to only sync over internet or WiFi, and AmpMe mentions it supports offline support with a local LAN, which implies it only syncs over internet as well.

Both software products allow to output audio to Bluetooth-speakers, yet Soundseeder mentions *manual* adjustments might have to be taken. They advise to first make steps of 100 milliseconds to get the synchronization correct roughly, and to further pin-point the synchronization more accurately with steps of 10 milliseconds [14].

AmpMe doesn't mention anything about manual synchronization to potentially compensate for any deviations in output latency, however on the Google Play Store multiple comments about out-of-sync speakers is present [15] [16] [17]. Each comment has 20 likes or more, we therefore assume other users to agree on their statement, and therefore we assume these claims to be true and persistent. We outline in specific the second complaint, who mentions a lag of around half a second. We assume the reviewer is talking about a large perceived audio drift in the synchronization, rather than a half second of latency for audio playback to occur (which should considered expected behavior). This comment, in addition with the third comment mentioning Android and iOS devices not syncing properly (for which iOS devices is known to provide much lower output latencies than Android) we conduct from these reviews that AmpMe doesn't support "full" synchronization as Shairport Sync does, as otherwise these deviating latencies would have be compensated for.

2.1.5 Recapping sound reinforcement systems

The reinforcement systems we have mentioned support different use-cases. Each approach offers a scalable solution, yet differ in the requirements for software and/or hardware. Q-LAN offers calibration which guarantees audio drift of a few microseconds. Volumio guarantees an audio drift of a few milliseconds. SoundSeeder offers manual intervention to reduce audio drift in steps of 10 milliseconds. Furthermore, if using Bluetooth-speakers, none of the consumer-oriented solutions seem to mention (or guarantee) a minimal or maximum audio drift at all.

2.2 Platform functionality and performance

In the previous subsection 2.1 we have explored software and hardware available to achieve a sound reinforcement system. Ideally, we could extend an open-source project - such as Shairport Sync - to extend support for synchronized web-browser media playback. In the form of a web-application, the user would navigate to a web-page, synchronize their local clock with the source, and afterwards play along by correctly anticipating towards it's audio output latency. With a web-application, there isn't the need for mobile platform-specific implementations.

2.2.1 Output latency

Yet to target the mobile phone, we have to anticipate towards the output latencies that occur due the processing of audio in mobile phones. The audio output latency can vary greatly from approximately 5 millisecond up to at least 100 milliseconds [18]. However, in combination with Bluetooth, both the latency and the variation of latency may increase significantly. Values of an online test case show latencies that range from 200 up to 600 milliseconds. We show their resulting graph in figure 1. In practice, with different devices, the differences in latencies between devices and Bluetooth-speakers may be even much larger.

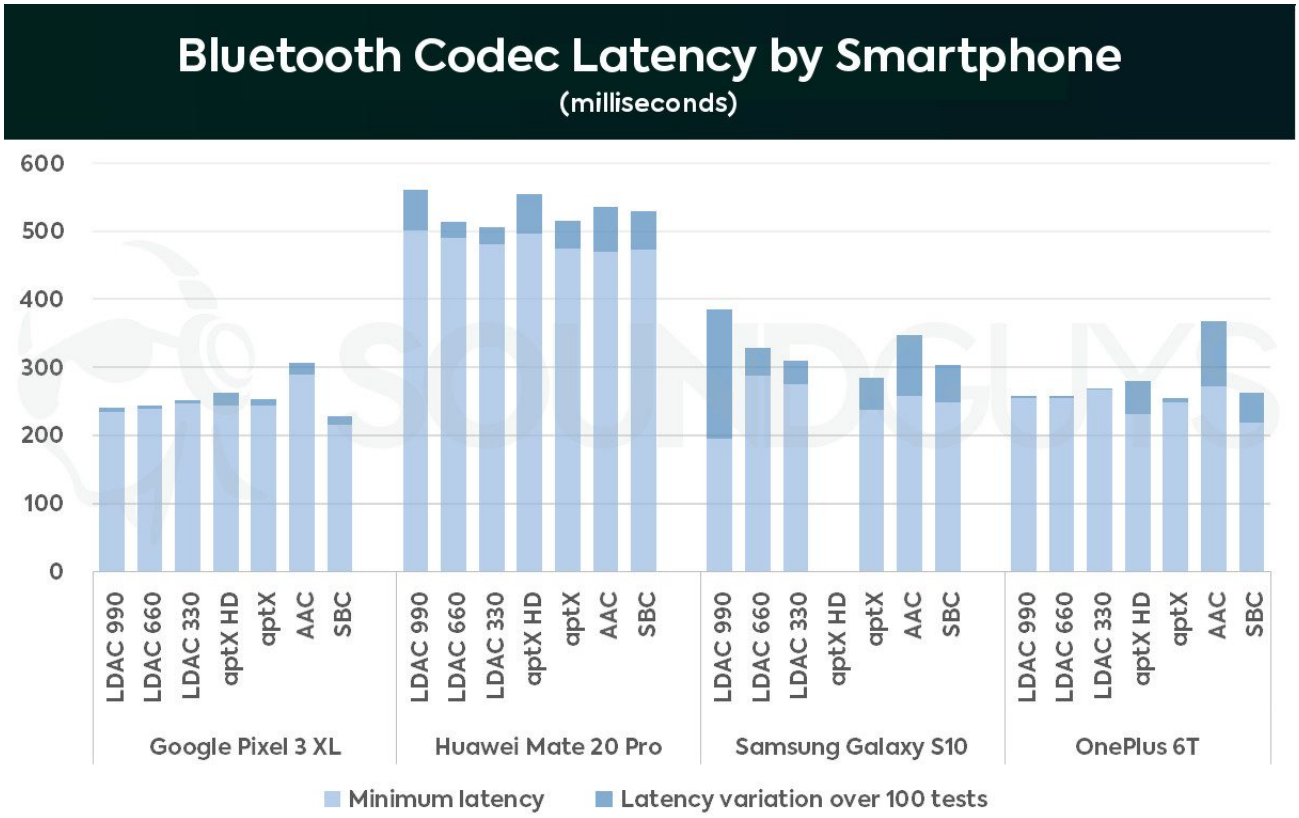


Figure 1: Audio output latencies for different mobile phones over a Bluetooth-speaker, used in combination with different Bluetooth codecs ([19], Section "Smartphone Bluetooth latency test results"). These results are conducted from 100 tests on each handset for each Bluetooth codec.

The cause for this increase in latency with Bluetooth usage, is the audio data to be delivered in batches to the speaker: packaging, fault-tolerant encoding, modulating, transmitting over the ether, demodulating, decoding, and as last unpacking. The article itself mentions that encoding and decoding is fast. Also, the Bluetooth encoder/decoder hardware doesn't influence this latency much. Differences in Bluetooth encoding therefore have mostly to do with the required buffering and/or efficiency of data transfer.

We are fine with large audio latencies in our sound reinforcement system, and thus accept variable or large audio output latencies. However, we have to know the latency very precisely. We are interested in how the feedback is provided or could be provided with the current software for mobile, web-browser and Bluetooth.

2.2.2 The audio path

In order to understand whether we can retrieve audio output latency from a web-application, we analyze the path that audio data travels from it's starting point - the web-application - all the way to it's final destination - the DAC of the speaker. We call this the audio path.

We elaborate on how audio output latency manifests itself in the audio path for the web-browser Firefox, running on Android, playing audio over a Bluetooth-speaker. Additionally, we analyze the audio processing system to find the way for a web-application to obtain accurate knowledge about the audio output latency.

2.2.2.1 Android

We begin with examining Android for the audio path. Superpowered has already analyzed the audio path within Android[20]. We have added an explanation or changed the explanation at each step where we have seen fit. Audio data is passed from the first item all the way through to the last item.

- User Application, the web-browser.
- AudioTrack, Android's abstraction to audio functionality within user-space (also usable by a User Application).
- Binder, shared memory where audio is stored by AudioTrack, and retrieved by AudioFlinger.
- AudioFlinger, audio stream handler in the Android media server.
- Android audio Hardware Abstraction Layer (HAL), mobile device agnostic approach to audio functionality.
- Audio (kernel) driver (such as ALSA), the driver that communicates with the audio card present in the mobile phone.
- Bus (such as USB, PCI, or Bluetooth), the medium over which the audio data is transferred to the DAC of the speaker.

Binder itself isn't adding any latency, it provisions shared-memory and acts like thereby as a communication medium over which AudioTrack and AudioFlinger share their data. The AudioTrack writes audio data into a First-In-First-Out (FIFO) buffer, and once the buffer is sufficiently filled, the AudioTrack stops writing and AudioFlinger begins reading from the first audio sample onward.

Similarly, the HAL acts like a communication medium between the audio stream of the Android media server (AudioFlinger) and the audio framework. Here as well, a FIFO buffer is used for sharing audio data, and a certain buffer size is required to be filled before the audio framework will read out the audio data.

As last, the bus that transfers audio data to the speaker may also communicate data in certain quantities. In specific Bluetooth, transports data packets, each containing a block of audio data.

A root cause for audio latency, is the choice of the buffer size per read/write between each of the two. Namely, a larger buffer implies AudioFlinger will have to wait longer before the buffer is ready to be read. Therefore a smaller buffer size results in a lower audio latency. Yet, the drawback of decreasing the buffer size, is risking insufficient audio data to be read out by the receiver due some writing latency or writing loss at the sender's end. Also, it increases the chance of audio glitching [19]. Another cause of latency may occur if the application audio processing is preempted for other activities running on the phone, therefore the application should be given priority over others by using a higher-priority thread ([21], "Using a high priority callback").

2.2.3 Output latency information feedback

There are multiple places in the audio path that introduce audio latency. In order to deduce what happens, each node in this path should provide feedback on how much time further communication has taken. We consider each node. We start at the end of the audio path, at the Bluetooth-speaker.

- The Bluetooth driver in the speaker has to provide feedback to the Bluetooth driver in the mobile device.
- Then the operating system has to read out this latency from the Bluetooth/audio-driver.
- The user application has to be be able to retrieve this information from the operating system.
- The web-browser requires an web API call to inject the information from the user space into the browser-sandbox.

We elaborate on each point.

2.2.3.1 Bluetooth

The Bluetooth Audio/Video Distribution Transport Protocol (AVDTP) [22] provides audio and video streaming with latency management. The Bluetooth endpoints synchronize their clocks ([22] chapter 16, transport and streaming considerations) and during communication provide delay information by comparing send and receive timestamps ([22] chapter 8.19, delay report signaling). It should suffice to give accurate latency of the bus during run-time.

2.2.3.2 Android Audio NDK

As we have mentioned for Shairport Sync, an audio framework like ALSA can provide feedback about the audio samples it has written to it's output node. Even though we send audio data over Bluetooth, and it therefore seems logic to write audio data instead to the Bluetooth-driver, audio data is still processed by the audio framework. And indeed, audio data is afterwards forwarded to

the Bluetooth-driver, such as BlueZ. Android provides audio functionality through the Native Development Kit (NDK) it's audio Application Programming Interface (API): AAudio and OpenSL ES [23].

The audio NDK interfaces with the audio framework, also we should be able to address audio latency information from within an Android application. AAudio for example, supports feedback on the amount of written samples [24]

2.2.3.3 Web-browser

The last step that remains is obtaining this information within the web-browser. In Firefox, JavaScript code is evaluated by the SpiderMonkey interpreter (in the form of Just-In-Time (JIT) compilation). When a certain audio-related function is called, as defined in the audio Web API, the Java-Native-Interface (JNI) handles the request, and handles it with the underlying implementation of the web-browser. Also, in the case of Android, web audio API commands are translated into mobile-native audio API requests.

How a web-browser should function is standardized by the World Wide Web Consortium (W3C), and we both reference to their documents and adhere to their standards that are considered stable. Such as the Web Audio API standard [25], that defines how to use and interact with audio within a web-application, and how the web-browser should respond to API function calls.

An audio media element in the browser supports reporting current time of audio playback of a song [26]. It therefore could allow the web-browser implementation to precisely deduce the audio output latency. In addition, the audio context (which we elaborate on in 4.2.2) provides a read-only attribute `outputLatency`, that estimates the audio output latency, and depends on the platform and the connected hardware audio output device.[27]. However, this does not change with the audio context it's lifetime ([25], "DOM audio-context output latency"), also it is only evaluated once. Evaluating again during run-time would require to build a new audio context, which is computative expensive and a bad practice: re-evaluation without re-creating a audio context should have been provided by the API instead. Furthermore, `outputLatency` is currently *only* supported by Firefox version 70 or higher (this does *not* include Firefox for Android) ([27], section "Browser Compatibility"), and therefore, even though the functionality is specified by the web audio standard, it currently isn't widely implemented. And even if implemented, the standard doesn't guarantee any accuracy, it is only an estimation.

2.2.3.4 Combining output latency

We note, that this information can only be passed back in the accumulated latency by traveling each node in the audio path, and thus the feedback isn't provided instantly. However, this shouldn't be a problem, since the output latency will be very accurately derived as statistical information can be gathered over time.

2.2.3.5 Recapping audio output latency handling

We have analyzed the audio path for audio from a web-application all the way to the DAC of a Bluetooth-speaker. Each component through which the audio data travels offers - or is capable to - provide accurate feedback on the amount of latency it introduces. We have seen from other research that mobile devices have non-trivial latencies for outputting audio, and that current mobile web-browsers don't have the functionality implemented to provide this information to a web-application. Therefore, we don't see the current landscape fit to approach audio calibration in an equal manner as Shairport sync. In the following section 3 we propose a different approach to calibration, by physically measuring the audio output latency rather than attempting to do so by API requests.

3 Calibration

In section 2.1 we have explored the available sound reinforcement-systems, afterwards we mentioned the fitness of a web-application to meet the audio calibration approach as Shairport Sync in 2.2.2 by analyzing the audio path. We concluded that, under the current technological landscape, the audio output latency is too large to neglect, and the popular web-browser provide insufficient functionality to provide accurate audio output latency feedback to anticipate towards the non-trivial audio output latency. Therefore, we approach the calibration differently than Shairport Sync.

3.1 Signal detection

In order to successfully record the speaker output and afterwards detect the signal the speaker has output, we need an approach to signal detection. Instead of relying on audio output latency feedback by the web-browser implementation, we will conduct audio experiments with the microphone and speaker present in the mobile device to deduce output latency.

We consider two techniques for output latency deduction: the Larsen test, and through signal detection with the aid of cross-correlation.

3.1.1 The Larsen test

Android used the Larsen test to measure the audio round-trip latency [28]. The device outputs everything it records in real-time. An impulse - a short-duration, loud sound - is output over the speaker, recorded by the microphone, and directly output again over the speaker. A successful Larsen test should have a resulting audio recording with clearly loud peaks at a constant time interval. The time interval between each audio peak indicates the audio round trip latency, because this is the time for the device to record the impulse and emit the sound again. The Larsen test name is derived from the Larsen effect - feedback singing - which occurs when we perceive to hear a certain frequency at the interval speed at which the impulse is emit.

3.1.2 Cross-correlation

Instead of detecting a loud audio peak, more preferably we could detect a certain sound pattern which isn't necessarily more loud than the other sounds in the environment. This is possible with cross-correlation: The device records itself and concurrently emits a certain sound, and instead of seeking an audio peak, the audio is analyzed and searched for the specific sound pattern. A successful cross-correlation results in a graph with a single clear peak at the point in time at which the recorded material fully matches the target signal pattern output by the speaker.

We consider cross-correlating to be more convenient to latency measuring than the Larsen test, because it allows for a larger range of sounds to test against, in environments with loud background noises. Potentially, it might be possible to calibrate against music of an active sound reinforcement system we want to participate in, removing the need for artificial - or other

disturbing sounds - to calibrate against in this system. Furthermore, it is more robust, since external sounds can more easily influence a loudness peak detector than a sound pattern detector.

3.1.2.1 Algorithm

We begin with approaching the cross-correlation algorithm in respect to the time domain in 3.1.2.1, and deduct tests for the effectivity of this signal detection algorithm. Cross-correlation defines the relation between two arrays of integers f and g, both having a size of N real values, and an integer τ that relates the offset of f in respect to g with the amount of similarity between the two arrays. The mathematical formula for cross-correlation is defined as:

$$(f \star g)(\tau) = \sum_{t=0}^{N-1} [f(t) \cdot g(t+\tau)] \tag{1}$$

$$\tau \in \mathbb{N}, (f \star g)(\tau) \to \mathbb{R} \tag{2}$$

We interpret f to be the target audio signal we seek in the audio source signal g. We note how our target signal is only a fraction of a second, while we will be searching for this signal in a second up to multiple seconds of recorded audio. Therefore, we can consider τ as the start index of g for a correlation with f. A single cross-correlation value describes the amount of similarity between the two signals that have been compared with each other. Higher positive indicates more similarity, while higher negative indicated more anti-similarity.

In a real scenario, in which we record a full second - 44100 samples - of audio into g and seek a target f consisting of 4096 samples, we expect to obtain 40004 $(44100 - 4096)$ results. If we set the offsets τ against the correlation results and draw this a new graph, we obtain the the cross-correlation graph of f and g.

A height of the value of $(f \star g)(\tau)$ indicates the amount of similarity between f and the partial samples of g starting at τ. The highest value in the resulting graph indicates the amount of shifting to obtain the most similarity between the two functions. In case the result is negative, the signals are anti-similar, also certain to many wave-patterns are present in both the target and source signal, yet have an opposite phase.

3.1.2.2 Time-domain implementation

We implement the cross-correlation function in JavaScript. Note the usage of `Float32Array` as we require float precision. Using something like e.g. `Uint8Array` magically results in zero-valued arrays due to implicit float to integer conversion.

```
1 function correlate( target, source ) {
2   let result = new Float32Array( source.length - target.length + 1 );
3   for ( let tau = 0; tau < source.length - target.length + 1; tau++ ) {
4     for ( let i = 0; i < target.length; i ++ ) {
5       result[ tau ] += f[ i ] * g[ i + tau ];
6     }
7   }
8   return result;
```

```
9 }
```

Listing 1: Cross-correlation JavaScript implementation

There is another implementation that considers the source signal as a circular buffer. We however do not want to use that variant, since we seek a pattern in consecutive samples, which the circularity doesn't support when splitting and gluing the last with beginning samples onto each other.

3.1.3 Testing effectivity and deriving a test validation technique

Conceptually, we convert the participating devices into an active radar system. The speakers act as transmitters, the signal the speakers emit is generated Gaussian white noise. The microphones act as receivers, and the cross-correlation function - with the generated Gaussian white noise as the target signal, recorded audio as the source signal - acts as the matched filter.

Research into the effectiveness for cross-correlation in respect to passive radar applications is carried out, such as in [29]. Extensive knowledge available, such as statistical signal analysis - which is relevant in relation to the auto-/cross-correlation process of Gaussian white noise that we apply - such as in [30]. Techniques for signal detection choices exist, for instance SNR improvements as mentioned in [31]. However, these sources are too complex to understand and apply in the scope in this bachelor's thesis. We experiment with an simplified approach and rough indication on the effectiveness/accuracy of the customized signal detection/radar system we implement, and thus neglect the vast amount of potential points of improvement.

We do some testing with the performance of signal detection of Gaussian white noise.

3.1.4 Virtual signal detection test case

We start with a virtual test on how well the cross-correlation function functions in signal detection. We relate the peak-to-peak Signal-To-Noise ratio to the accuracy of the cross-correlation algorithm. We outline the procedure for this test.

1. Pick a target sample size.
2. Generate a target sample size amount of white noise samples, which represents the target sound of the audio recording.
3. Generate a full second of white noise, which represents the background sound of the audio recording.
4. Merge target signal into the background signal to create the source signal.
   (a) Apply loudness ratio to the signals: Multiply each sample in each recording appropriately. This allows testing under different Signal-To-Noise Ratio's (SNRs).
   (b) Generate a random index at which to inject the target signal in the background signal, which
   (c) Inject the target signal into the background at the randomly generated sample index.
   (d) Store the index of injection for later reference to compare the estimation of the algorithm against.

5. Apply the cross-correlation algorithm to the target signal with the source signal.
6. Store the execution time, correlation result, target signal and source signal.
7. Deduce other statistically relevant information:
   (a) Seek the index within the cross-correlation that has the highest value - the correlation peak - which represents the highest similarity with the the actual injection sample.
   (b) Compare the found index against the index of injection. If these two values match, the algorithm has successfully found the target signal.

Note: The cross-correlation peak must be absolutely valued the highest. It can therefore as well be negative. In that case, there is the most anti-similarity between the target and source, which in the signal context implies there is a phase offset.

We test with target signals ranging from 32 samples up to 8192 samples. Additionally, we compare with different SNR's, ranging from 0.1 to 10. Each setting we perform a hundred tests, in such the result is more statistically significant.

The testing is based on exploratory picking ratios. Therefore, there is an inconsistency between the chosen intervals per target sample size. Each data point in the results, displayed in 2, is the average of the hundred Monte Carlo tests performed in that setting.

3.1.4.1 Accuracy versus loudness

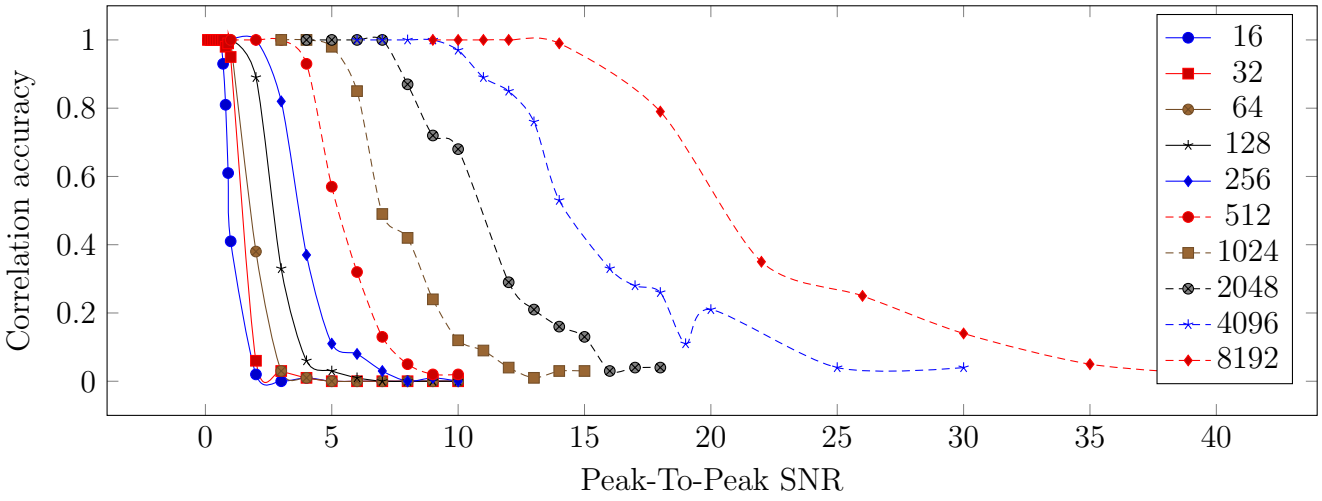


Figure 2: Accuracy of cross-correlating a target signal against one second of audio recording under different peak-to-peak signal-to-noise ratios, for different target sample sizes.

We can clearly see that a larger target sample size results in a more accurate cross-correlation. Additionally, it seems like the length of the target pattern can outnumber it's reduced signal strength in the resulting cross-correlation graph completely.

Without supplying/knowing/having the actual mathematical proof, I think this behavior can be addressed to the significance of the accumulated contribution of each target sample. White noise - a fully random signal - doesn't have similarity with other signals. Also, in theory, an unrelated

signal should result in a zero-valued cross-correlation. Since the target signal is present at each sample index - no matter it's actual strength - if a sufficiently large sample size is used, each sample contributes to accumulate into a correlation peak. Also, the target size has to be sufficiently large, and unique, to become statistically significant and therefore addressable for it's given SNR.

Note how the cross-correlation algorithm can detect weak signals, which would be impossible to detect with the Larsen test. Namely, once the background sound becomes more loud than the target signal, the Larsen test doesn't provide any mechanism to discriminate between the target signal and the background sound. Also, the test doesn't provide a mechanism it sufficiently invalidate different samples, as the cross-correlation manages to do achieve with statistical significance.

3.1.5 Signal detection result validation

In practice we aim to detect the signal without prior knowledge of the signal occurrence. However, in case a certain SNR is surpassed, as we have seen in 2, the peak doesn't have to be the target value. Unfortunately, we don't know during measuring the SNR, also we cannot invalidate the measure beforehand. As we seek a peak, and a peak will almost guaranteed to exist, we can be certain to detect, and thus we require a measure to decide on whether it is probable a test result is valid.

We could consider the loudness of the highest peak in contrast to it's second correlation peak and consider this as an heuristic value on how strong the correlation peak is. However, nearly adjacent points could be part of the same signal response and therefore this heuristic value could result in a very low value - even though the cross-correlation may have a very strong and accurate peak - and therefore incorrectly invalidated. Also, this naive approach is incomplete: how many peak values exactly should be considered to be part of the same correlation peak?

We could consider a peak detection algorithm, and compare the two highest peaks found by this peak detection algorithm to compare the ratio between the two. This may improve the heuristic correctness significantly, yet requires a peak-detection algorithm that relies on parameters to fine-tune it for usage. Yet still, even with a proper peak indicator, when exactly should we consider that peak to be relatable to the target signal?

Instead, we compare the highest peak value of the cross-correlation result with the the average cross-correlation value of the graph (peak-vs-average ratio). We outline this comparison in the graph below with the result data from the virtual test.

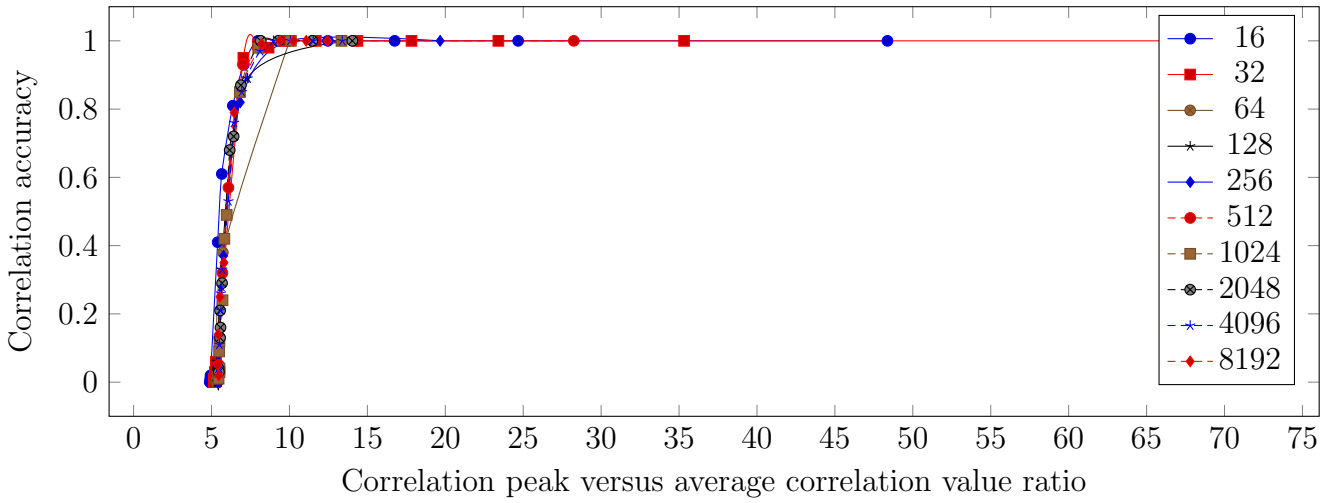


Figure 3: Accuracy versus cross-correlation peak strength to average cross-correlation ratio.

We can see, for each target sample size we have test with, that the signal detection algorithm guarantees to succeed once the peak is 10 times stronger than the average correlation value. Thus once the peak-vs-average ratio of 10 or higher, the peak guarantees to be the target peak in this virtual test environment.

3.1.6 Signal detection speed

As seen in 3 we would clearly prefer a very large target sample size for an increase in accuracy. However, the drawback of picking a larger target frame size is the increase in computation cost. We have test the speed of different target sample sizes. In the previous test, we have in addition kept track of the computation time required. The average computation time per target sample size is outlined here.

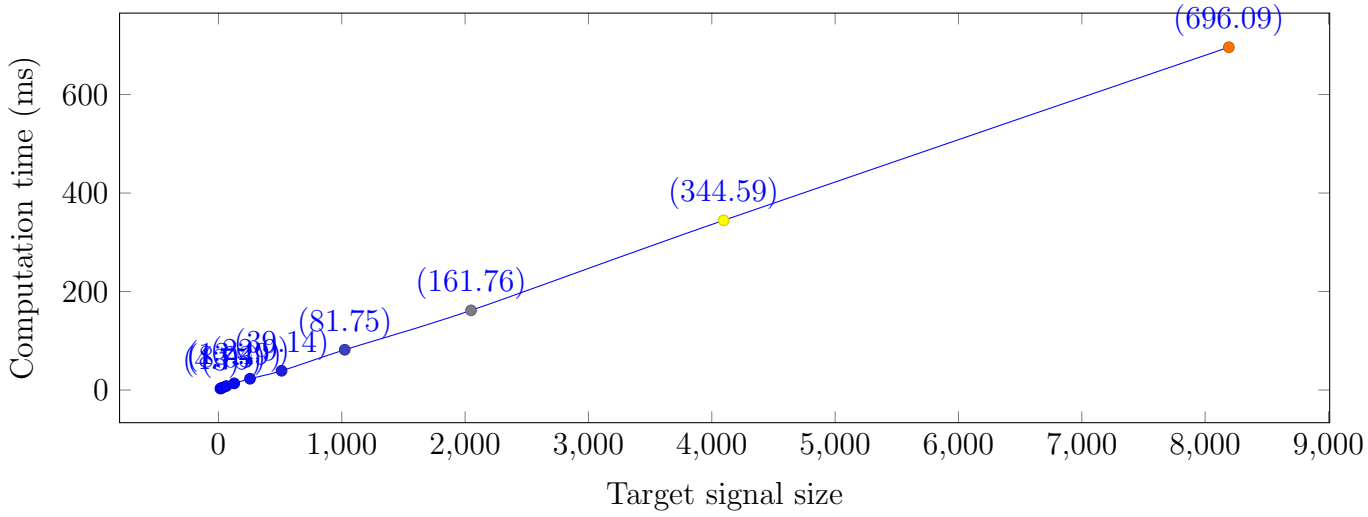


Figure 4: Speed of computation for cross-correlating with different target signal sample sizes against one second of an audio recording.

There is a linear growth in computation time, which can be explained as followed: A doubling of the target signal sample size doubles the amount of required comparisons for a single cross-correlation result. I would have expect a fitting linear function would have a slope slightly lower than 2, (the computation time to grow a little less than 2), since one additional sample of target size lowers the amount of correlations to be computed by one. Maybe, the allocation of larger sized Float32Array's introduce the additional latency. Yet, we consider it irrelevant to further investigate this specific algorithmic computation time.

3.1.7 Real signal detection test case

From the results obtained in 3.1.2.1, we conclude that in theory, if the background sound consists solely of white noise and even if it is ten times louder than the recorded target signal, we can guarantee to find the target signal by a target sample size of 4096 or larger.

We test the relative cross-correlation peak strength in relation to the peak-to-peak SNR when using the speaker and microphone for obtaining the source signal, rather than virtually composing the source signal we did in 3.1.2.1.

3.1.7.1 Ideal environment

We test with a physical measure environment with a high SNR to check whether the algorithm succeeds in such an ideal situation. We say to have successfully found the target peak, if the peak-vs-average correlation ratio is 10 or higher. And thus rely on the results of the graph of the previous section 3.

The setup environment consists of a single speaker functioning at 44100 Hz and a microphone functioning at 44100 Hz. We use a single speaker - mono channel - to prevent multiple speaker outputs interfering with each other for the microphone input. We repeat the test a hundred times. Since we very clearly can capture the target signal with the microphone, we expect each test to succeed. The resulting peak-vs-average correlation ratio is displayed. Each result 2 had a ratio of

Minimum	Average	Maximum
37.25	73.78	95.87

Table 2: Strength of the highest cross-correlation peak in relation to the average cross-correlation value of a measure under ideal circumstances, deducted from a hundred tests performed under the same setting.

37.25 or higher, which greatly surpasses the ratio of 10, and thus we expect to have perfect signal detection accuracy.

To be certain the signal detection algorithm has actually succeed, we take a closer look at the results to be sure. Here below the target signal, source signal, and their cross-correlation result of the first test is visualized. This particular has a peak/average ratio of 67.94.

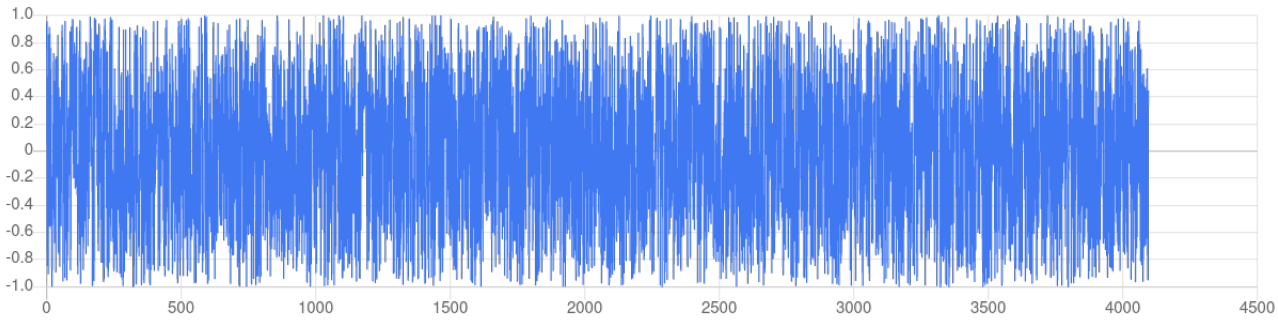


Figure 5: The target signal.

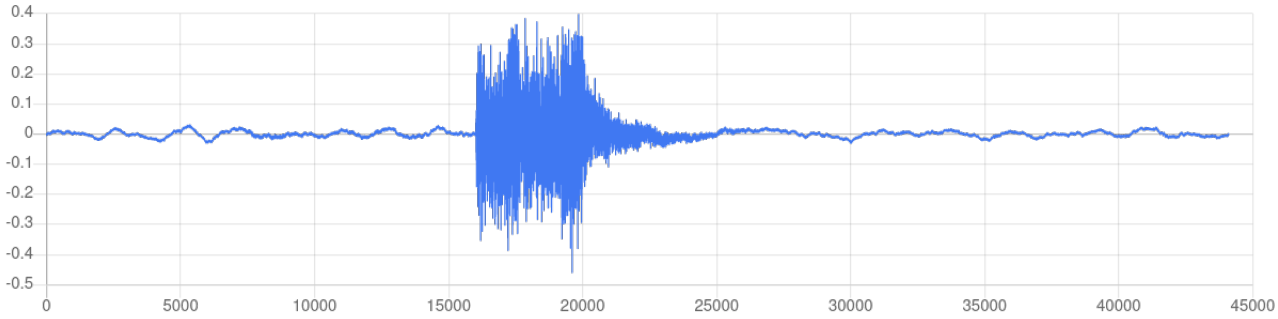


Figure 6: The source signal.

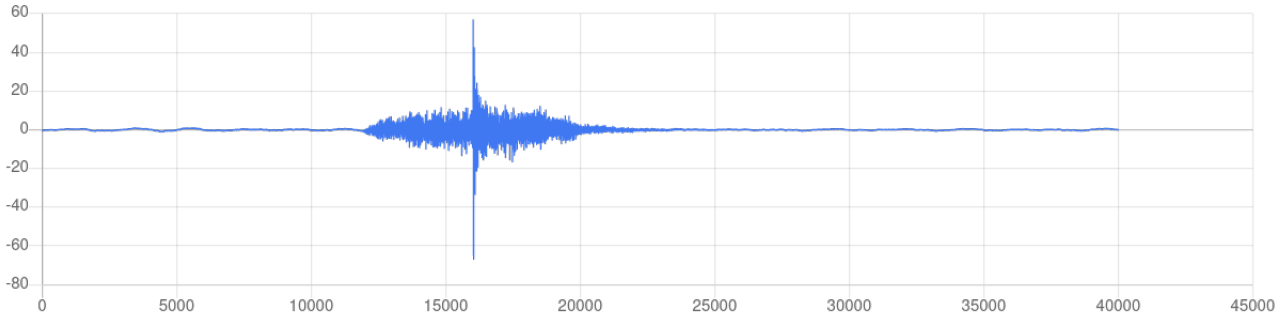


Figure 7: The cross-correlation of the target signal with the source signal.

From the source recording in figure 6 we can see the target signal is loud and clearly received by the microphone. In addition, the cross-correlation graph displayed figure 7 has a single large spike, exactly at this point in time, and thus the signal is detected successfully.

In the source signal we see an sudden volume increase caused by the speaker output. The speaker turns silent equally abrupt after outputting the target signal, yet the dissipation occurs rather slowly in contrast to an abrupt audio silence. This could be ascribed to echo.

Around the point 12100 we see a rise in correlation. From this point in time the correlation includes summations with the beginning of the loud (target) audio in the source around 15500,

resulting in increased (yet still random) values. Similar behavior occurs at the the tail of the correlation graph at the point 20000.

Remarkable is the negative value at 16037, which is both negative *and* stronger than the earlier found positive peak at 16021. Cross-correlation is known to provide a strong, negative result if the source signal has a half phase offset to the target: the signals are then multiplied sample wise with their sample values on the exact opposites sides of the y-axis, and thus each sample-wise multiplication is either zero or negative.

We aren't dealing with a single frequency sinusoidal wave for which a clear phase exists. As there is a 16 samples difference between the positive and negative peak, then with the above reasoning this implies the wave to have shift by half a phase with 16 samples. Also, the wave phase completes in 32 samples, which would be a wave of 1380 Hz under a sample rate of 44100 that we have record with. We have displayed the spectral analysis of the source signal 10, where we further elaborate on this peak.

3.1.7.2 Spectral analysis

We display the frequency spectrum of the 4096 target samples in figure 8, the frequency spectrum of 4096 samples of the source signal of background noise, taken around 5000 samples before the cross-correlation peak occurs in 9, and the frequency spectrum of 4096 samples of the source signal starting at the sample index on which the cross-correlation peak occurred in figure 10.

As we take a Fourier Transform of 4096 samples, we obtain 4096 frequency bins. The second half of these bins has the exact same values as the first half, yet mirrored, due to the Nyquist rate. Therefore, we select only the first 2048 frequency bins without information loss. Since we have recorded audio with a sample rate of 44100, the half of this frequency range, 22100 Hz, is divided over each frequency bin. Thus, each frequency bin covers a frequency range of approximately 10.8 Hz. The source signal isn't normalized before calculating the frequency spectrum, we neglect it as we focus on the frequency distribution.

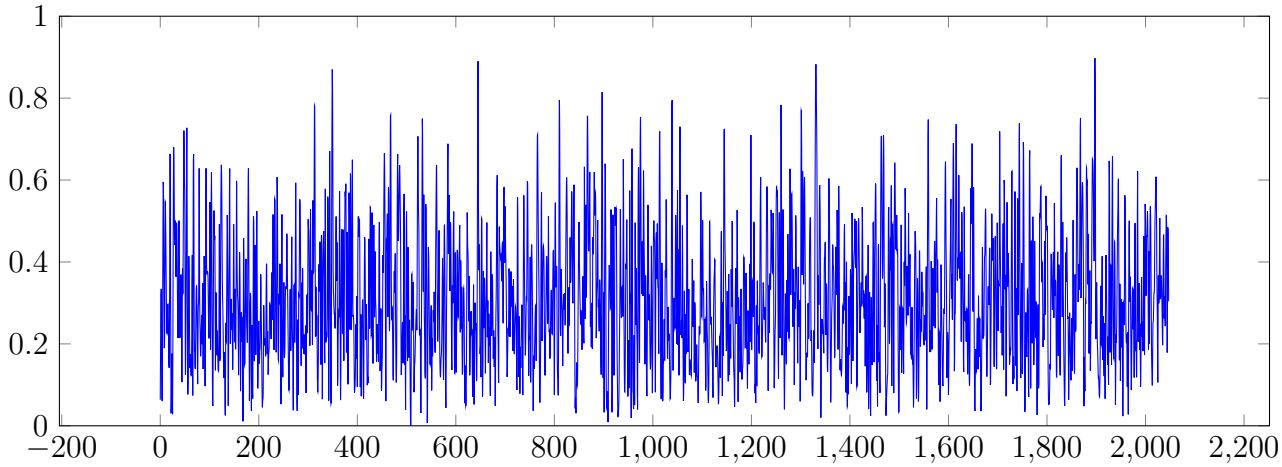


Figure 8: Frequency spectrum of the target signal.

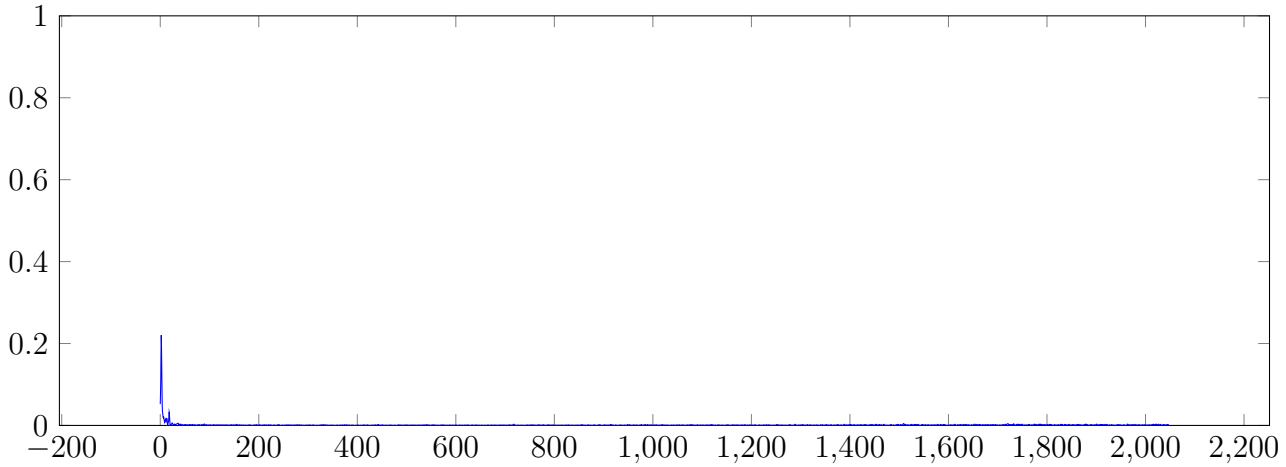


Figure 9: Frequency spectrum of the source signal at the occurrence of silence - the background noise.

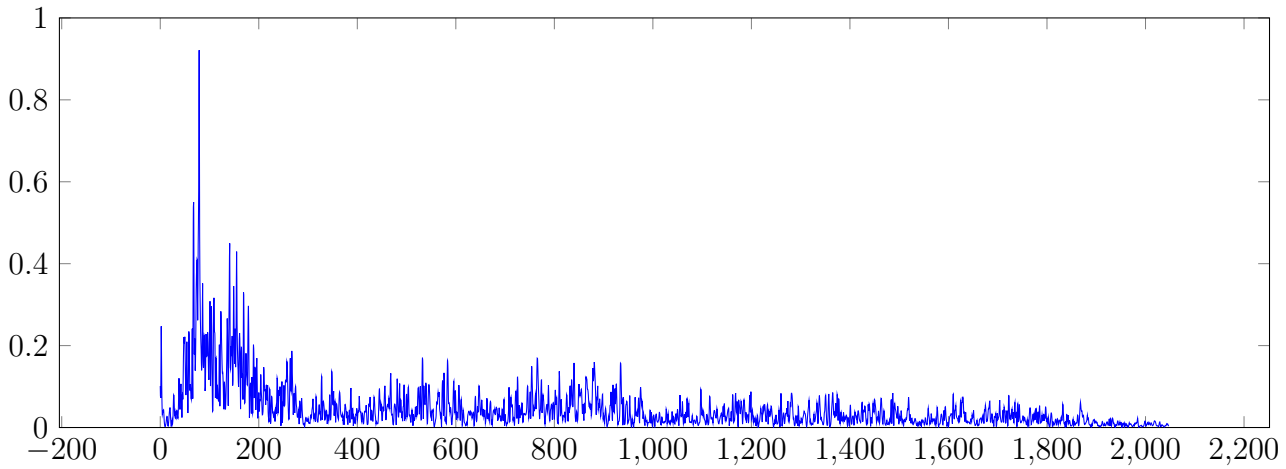


Figure 10: Frequency spectrum of the source signal at the occurrence of the target signal.

The background noise has a peak value at the third frequency bin, which is around 30 Hz. This peak remains visible in the frequency spectrum in the source signal when the target occurs. The frequency bin values of the target signal seem completely randomized, and once measured in the source signal a lot of spectral filtering seem to have occurred.

Part spectral filtering can be ascribed to attenuation of the source signal in either the speaker output or microphone input. In practice, either the microphone, speaker, or both, may fail to emit or to receive the full range of frequencies. Additionally, microphones may be more sensitive to certain frequencies, speakers may output certain frequencies more loudly.

Besides attenuation introduced by the hardware, another cause might be that higher frequencies dissipate more quickly as sound is traveling through air. However, this is unlikely to have occurred this presently in this specific measure result, since the microphone was place directly next to the speaker.

The spectral filtering that has occurred to the target, has two strong peaks. The first peak around frequency bin 79, and the second peak around frequency bin 130. The second peak could very well explain the negative correlation spike: If the target signal got filtered in such a way that in specific frequencies around 1400 Hz would be the loudest, the resulting signal - even though still white noise - would have a stronger sinusoidal wave and thus such a phase of that.

We haven't got an explanation for the first peak. We zoom in to better analyze the signal. We list the samples 15900 to 16400 of the source signal - where the cross-correlation peaks occur - in figure 11 and the first 300 hundred samples of the target found in the source signal in figure 12.

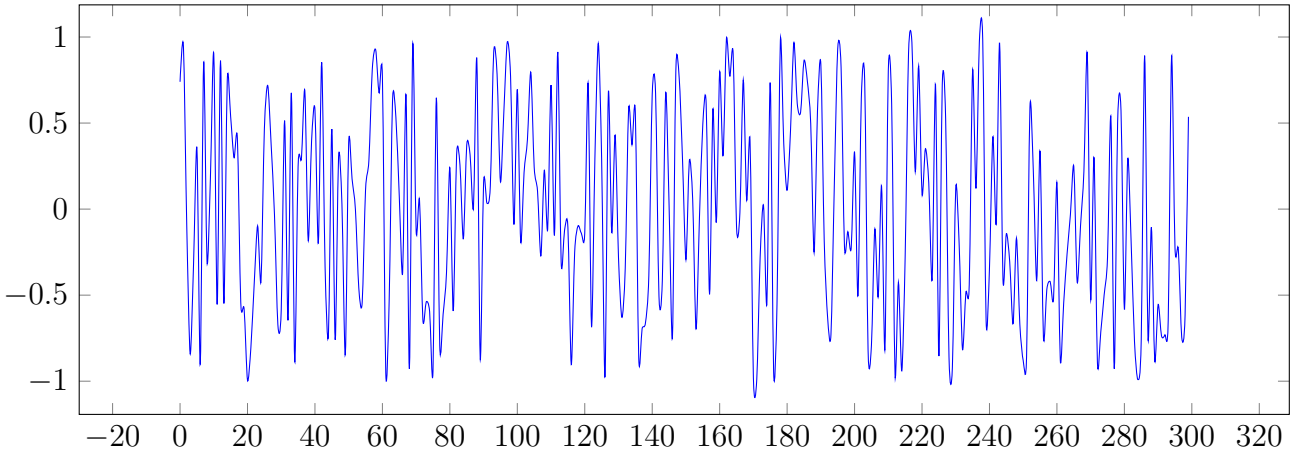


Figure 11: Target signal, samples 0 to 300.

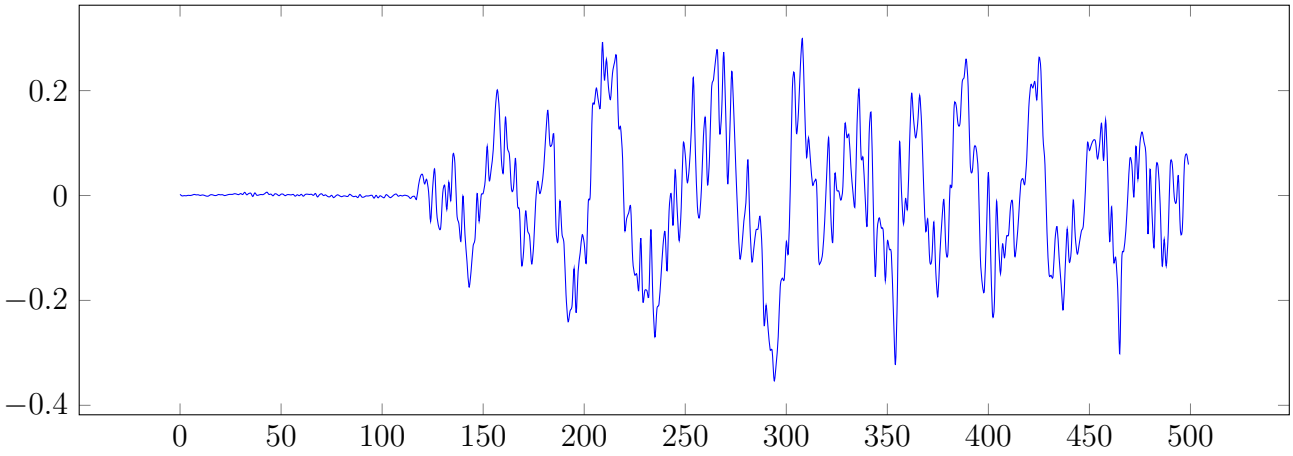


Figure 12: Source signal, samples 15900 to 16400.

If we zoom in, it remains hard to see any clear recurrence of the target signal in the source. Yet what can be clearly seen, is that it seems like a sinusoidal wave is introduced to the source signal. This pattern continues for the entire 4000+ samples. These waves aren't caused by the background noise, since we can clearly see there is barely any background sound present in the first hundred highlighted samples here, and these first hundred samples are background sound only.

Therefore, either the audio speaker has introduced this pattern during playback, or the microphone introduced it with recording, or some external object that began to resonate due the speaker has become part of the recording. Since the laptop got placed on a table during the test, it is plausible either the laptop itself, or the table it stood on, resonated along.

3.1.8 Error sources

The resulting table 2 of the physical tests show a large deviation between the minimal and maximal correlation peak/average ratio. By analyzing the audio, we already have attributed potential causes for a reduction in, or alteration of, signal detection strength due a variety of sources, such as echo, background noise, resonating objects, signal attenuation.

However, they may be many more. Audio clipping, malformations/impurities in the speaker output or microphone input, microphone SPL measuring clock cycles slightly missing the exact moment of the signal arrival and thereby introducing a kind of signal interpolation, random audio spikes and other type of changes of the background noise, signal interference once the target signal is dispatched from multiple speakers simultaneously, even things possible such as minor frequency shifting due to the Doppler-effect if someone throws it's phone through the room during calibration.

There is insufficient time available to address the influence of each potential error source. Therefore, we won't further elaborate on adjusting/understand/improving/testing the signal detection algorithm in relation to these aspects.

We do elaborate further on the importance to prevent audio clipping and up-/down-sampling, since either of these two are common to occur, and are fatal to the signal detection.

3.1.8.1 Audio clipping

Audio input data from the microphone is feed to the web-browser within the range of values between -1 and 1. Any measured sample value too loud - that ends up outside of this range - is therefore mapped to the boundary of this range. This is called audio clipping. Audio clipping causes a loss of information, since it is unclear what exactly the value of a clipped sample is - we only know it was too loud - and lied somewhere beyond this range of -1 to 1.

Audio clipping - in the context of audio recording - occurs once either the microphone is too sensitive, the speaker outputs audio too loud, the speaker and microphone are too close to one another, and/or all of the above.

Decreasing the microphone sensitivity in order to prevent audio clipping might resolve in insufficient sensitivity to detect a target signal of a device from far away. Additionally, decreasing the loudness of the speaker may prevent audio clipping.

Audio clipping thereby introduces a constraint on the range in which devices can communicate, and the level of background noise under which a device can operate. The point made about the statistical significance of the target sample size to outnumber the reduced influence due a low SNR in 3.1.4.1, doesn't hold up once sample values are being clipped: The minor, yet relevant - information contribution of each target signal sample is lost completely.

The web audio API or media stream API is unable to adjust the microphone sensitivity, and audio data streams in clipped between -1 and 1. Also, a software component in the audio path strictly clips the input signal. Maybe the user can adjust the sensitivity manually, if such an option is available. Either way, we are unable to automatically adjust/anticipate to audio input receival *without* user interaction. There is a certain convenience built-in feature, which is the `autoGain` parameter to define automatic gain control, maybe this is the way to go. We however skip this due time limitation.

3.1.8.2 Input and output sample rates

We briefly mention the importance of having a matching output and input sample rate. If they aren't, signal interpolation will occur and the target signal is rendered differently. A web-browser may up-sample or down-sample audio for *both* audio output and/or audio input where it sees fit, without warning or notification. Even worse, the web API doesn't mention how to retrieve up-/down-sampling information.

We ran into this problem during the physical cross-correlation tests. Even though we explicitly constructed the audio context to function with a sample rate of 48000, and requesting the microphone through the user media API with a sample-rate 48000 samples, the system silently down-sampled output to 44100, and up-sampled input of 44100 to 48000, all hidden behind the scene.

We managed to find this problem by using the audio framework (ALSA) on the laptop - to request the hardware devices present in the audio system. ALSA gave the feedback that both speaker and microphone function on a sample-rate of 44100 Hz.

Luckily, both audio hardware components function at the same sample rate of 44100 Hz, and it was therefore easy to resolve it by updating all audio-related functionality to a sample rate of 44100 Hz. However, if the speaker or microphone functioned on a sample-rate of 48000, it may have become more problematic.

I haven't looked in further into how this could be solved. Maybe it must be fixed with two audio contexts. Or maybe it cannot be circumvented, and in such the speaker and/or microphone have to anticipate to the interpolation that will occur.

Similarly, the only manner I can think of to detect the hidden up-/down-sampling, is by performing additional tests. For instance, applying the cross-correlation signal detection, yet instead of only testing against the original target signal, in addition test against all the possible interpolated variants that might occur.

3.2 Calibration performance model

In the previous subsection 3.1 we have outlined a technique to signal detection. In this subsection we elaborate on how to use this functionality to deduce relevant audio latency information. We aim for mobile devices to output audio at the same time. For each device, we assume to not know beforehand the exact server time offset nor audio output latency. And thus we use their microphones for signal detection.

With a signal detection algorithm available, we want to develop a model to figure out how we can define measurement steps and convert their results into relevant latency information. To simplify the model, we assume the signal detection functions accurately and provides correct information. Additionally we assume the latencies in the following paragraphs to be static, and thus neglect the potential of variance or drift over time.

3.2.1 Model variables

In figure 13 we define the latencies we recognize in our model for *each* device *separately*.

- e : Audio **e**mission/output latency for the web-browser to speaker
- r : Audio **r**eceival/input latency for the microphone to web-browser
- d : Audio signal travel time due **d**istance between devices
- $\emptyset$: Clock synchronization difference to the server clock

We do not know what the values of these variables are beforehand, also we have to deduce them with the aid of time synchronization and signal detection.

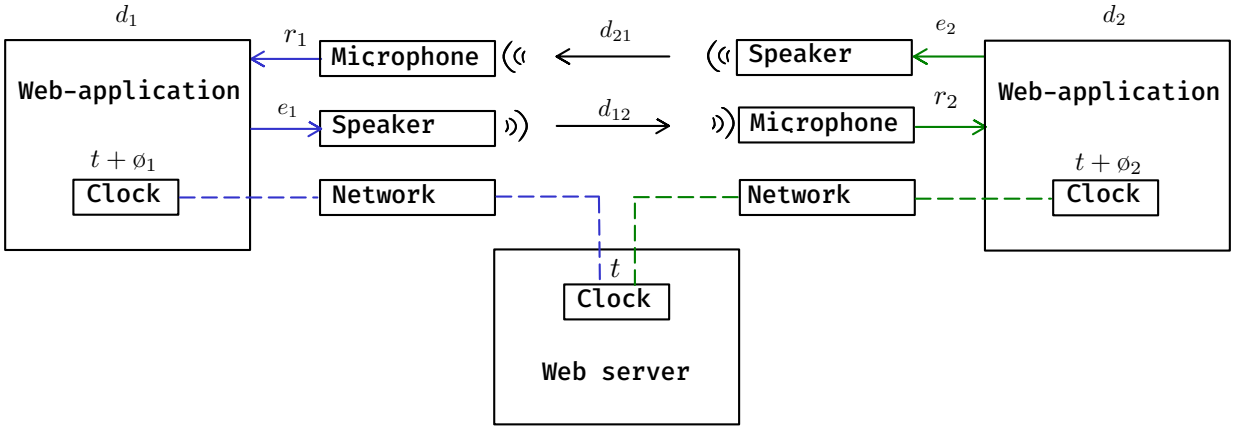


Figure 13: Device latency variables

3.2.2 Model measurement tests

We define the measurements in the scope of two devices. We can define four separate tests: x_{11}, x_{12}, x_{21}, x_{22}.

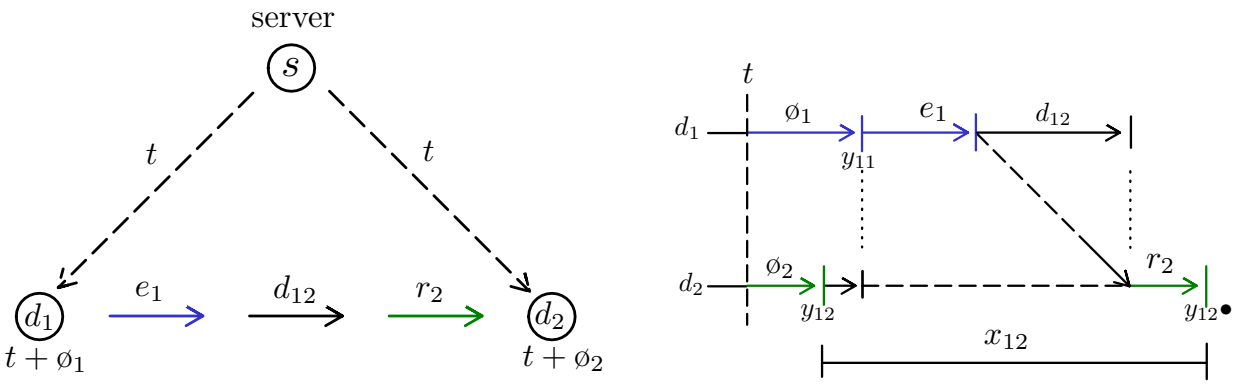
- x_{11} : Audio round-trip latency measurement test case for device **1** .
- x_{12} : Audio latency measurement test case, from device **1** to device **2** .
- x_{21} : Audio latency measurement test case, from device **2** to device **1** .
- x_{22} : Audio round-trip latency measurement test case for device **2** .

Each test measures the time it takes for the web-browser to output audio up to the moment the web-browser received that audio in it's input.

Since we deal with two different devices in the test cases x_{12} and x_{21}, we in addition require clock synchronization to communicate the start time for the test. Both devices **1** and **2** separately negotiate their local clock time offsets $\emptyset_1$ and $\emptyset_2$, respectively, in relation to the server time s. (We elaborate on this negotiation process in the proof of concept at 4.4.1.1.)

Let us consider the test case x_{12}, to measure the latency for device **2** to receive a signal from device **1** . A physical representation is given in figure 14a, and a time-graph of the test is drawn in figure 14b.

The server initiates the test, by sending the starting time t to both device **1** and device **2** . Device **1** emits an audio signal at time t, and concurrently device **2** begins recording at time t. As both devices have a (minimal) time-difference in the clock synchronization to the server clock, we note that the *actual* physical moment in time the devices begin with the test differs from the server time with $\emptyset_1$ and $\emptyset_2$ for device **1** and device **2** respectively. We name the physical moment in time at which the devices *begin* with the test y_{11} and y_{12}, and the physical time at which device **2** detects the signal with $y_{12}^{\bullet}$.



(a) Abstraction of measure x_{12}

(b) Time-graph of test case x_{12}

3.2.3 Decomposing test results into model variables

In this paragraph we provide the mathematical formula's to relate the variables defined in paragraph 3.2.1 with the test case results defined in the paragraph 3.2.2. As we will further elaborate on in paragraph 3.2.7, it is impossible to find the *exact* value for *every* unknown variable in our model with the tests we can perform. However, we don't need this information to obtain sample-precise audio output that we want for perfect perceived performance.

First, we derive how the result of x_{12} is composited of the known variable s, and unknown variables $\emptyset_1$, $\emptyset_2$, e_1, e_2, r_1 and r_2 present in our model. Note, x_{21} can be deduced in the exact same

manner as x_{12}.

$$\begin{aligned}
y_{11} &= s + \emptyset_1 \\
y_{12} &= s + \emptyset_2 \\
y_{12}^{\bullet} &= y_{11} + e_1 + d_{12} + r_2 \\
&= (s + \emptyset_1) + e_1 + d_{12} + r_2 \\
x_{12} &= y_{12}^{\bullet} - y_{12} \\
&= (s + \emptyset_1 + e_1 + d_{12} + r_2) - (s + \emptyset_2) \\
&= e_1 + r_2 + d_{12} + \emptyset_1 - \emptyset_2
\end{aligned}$$

The formula of x_{11} is different than x_{12}, since there isn't a time synchronization difference present, because it is a self-measure. Equally, x_{22} is deduced in the exact same manner as x_{11}.

$$\begin{aligned}
y_{11} &= s + \emptyset_1 \\
y_{11}^{\bullet} &= s + \emptyset_1 \\
x_{11} &= y_{11}^{\bullet} - y_{11} \\
&= (s + \emptyset_1 + e_1 + r_1) - (s + \emptyset_1) \\
&= e_1 + r_1
\end{aligned}$$

We list the four formula's that describe how the results of the tests we perform can be decomposed into the variables of our calibration model.

$$\begin{aligned}
x_{11} &= e_1 + r_1 \\
x_{22} &= e_2 + r_2 \\
x_{12} &= e_1 + r_2 + d_{12} + \emptyset_1 - \emptyset_2 \\
x_{21} &= e_2 + r_1 + d_{21} + \emptyset_2 - \emptyset_1
\end{aligned}$$

3.2.4 Deducing the distance between two devices

We assume devices to remain at a fixed position during the calibration process. Also, we assume the audio signal traveling time d_{12} in the measure x_{12} to equal the audio signal traveling time d_{21} in x_{21}. With this assumption, we can calculate the distance between device **1** and device **2** as follows:

$$\begin{aligned}
x_{12} + x_{21} &= e_1 + r_1 + e_2 + r_2 + 2 \cdot d \\
d &= \frac{(x_{12} + x_{21}) - (x_{11} + x_{22})}{2}
\end{aligned}$$

3.2.5 Deducing the calibration performance

The perceived calibration performance for a listener consists of the difference in arrival time of the audio output of device **1** in relation to the audio output of device **2** . We describe the calibration performance as the difference in arrival times for a given time sample.

A lower difference in arrival time implies a better performance. The performance is optimal, once the the arrival times are simultaneous. Given a audio sample is played at time t, device **1** will play the sample at time $t + \emptyset_1$, output the audio with an emission latency of e_1, and it is travels to the listener through the air over a distance with a latency of d_1. Equally, device **2** plays the sample at time $t + \emptyset_2$, emits the sample with a latency of e_2, and afterwards travels with latency d_2 to the listener. Also, an audio sample of device, scheduled at the time t by the server, arrives at the listener from device **1** at the time point $t + \emptyset_1 + e_1 + d_1$, and from device **2** at the time point $t + \emptyset_2 + e_2 + d_2$.

We don't know, nor can measure, where the user is positioned in relation to the speakers. Therefore, we simply assume the user is located at a central place in relation to both speakers, and neglect the difference in distance between the devices and the listener. This simplifies the requirement for optimal calibration, which now occur for device **1** and **2** if $t + \emptyset_1 + e_1$ equals $t + \emptyset_2 + e_2$.

Since we don't know the values of the model variables $\emptyset$ and e, we derive the calibration performance with the test results. We apply the the variable decomposition equations of the previous paragraph 3.2.3. The calibration performance p (the difference of the arrival for an audio sample of device **1** and device **2**) can then be described as

$$\begin{aligned} p &= |e_1 + \emptyset_1 - (e_2 + \emptyset_2)| \\ &= |(e_1 + r_1) - (e_2 + r_1 + d_{21} + \emptyset_2 - \emptyset_1) + d_{21}| \\ &= |x_{11} - x_{21} + d_{21}| \\ &= \left| x_{11} - x_{21} + \frac{(x_{12} + x_{21}) - (x_{11} + x_{22})}{2} \right| \end{aligned}$$

Note that, due the absolute value, we can invert the numbers to obtain the same result

$$\begin{aligned} p &= |e_1 + \emptyset_1 - (e_2 + \emptyset_2)| \\ &= |(e_2 + \emptyset_2) - (e_1 + \emptyset_1)| \\ &= \left| x_{22} - x_{12} + \frac{(x_{21} + x_{12}) - (x_{22} + x_{11})}{2} \right| \end{aligned}$$

3.2.5.1 Optionally choose which microphone to use

In practice, we are most probably dealing with short-range audio reinforcement only. Also, the distances are negligibly small, and in such the actual formula's we require to measure to the calibration performance is either $|x_{11} - x_{21}|$ and/or $|x_{22} - x_{12}|$. This implies that - under the assumption the distance is negligibly small - we can make the system functional with only one microphone present, even if we have more than two sources.

As an example, we extend the model to three devices (and consider distance to be negligibly small). Consider device **1** to have a microphone. Define p_{12} the calibration performance considering device **1** and device **2** , and the calibration performance between device **1** and **3** to be p_{13}. Then

$$\begin{aligned} p_{12} &= |x_{11} - x_{21}| \\ p_{13} &= |x_{11} - x_{31}| \end{aligned}$$

If we have p_{12} and p_{13} to equal zero, then certainly we have p_{23} to be zero as well. And this way the system is optimally calibrated by using a single microphone.

3.2.6 Optimizing the calibration performance

After performing the tests x_{11}, x_{21} and/or x_{22}, x_{12}, we deduce the calibration performance p. The performance however, may not yet be optimal, also p to be greater than zero. Recall from the previous subsection that we have $p = |(e_1 + \emptyset_1) - (e_2 + \emptyset_2)|$.

We cannot increase nor decrease the output latencies e, as this occurs outside of the reach of the web-browser. We could adjust the local clock, in such the calibration becomes optimal. However, this we consider bad practice, since we may be de-syncing perfectly synced clocks that have a bad calibration performance due to very large deviating output latencies.

Therefore, instead of changing the local clock, we introduce the variable τ: the audio scheduling latency. The calibration performance can now be optimized by adjusting the variable τ, and in such the new calibration performance formula becomes:

$$p = |(e_1 + \emptyset_1 + \tau_1) - (e_2 + \emptyset_2 + \tau_2)|$$

We add some restrictions to the values that τ may take. First, we may not address a negative value to τ as this may result into scheduling audio into the past for that device. Secondly, we aim to keep the scheduling at a minimum to prevent excessively large playback latencies. Unconditionally increasing audio scheduling latencies could result in scheduling unnecessarily far ahead in the future, which thereby will result in a unnecessarily slow and unresponsive application. Therefore, to minimize the latency we introduce, the slowest performing device (for which $e + \emptyset$ is the highest value) should maintain an audio scheduling latency of zero.

Here we outline an example on how device-related audio scheduling latencies of τ can optimize the calibration performance.

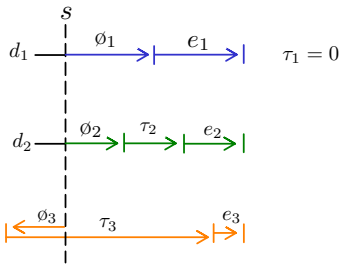


Figure 15: An example of three devices that have different audio output latencies and clock synchronization offsets. The calibration performance is optimized by applying device-related audio scheduling latencies.

In this example, device **1** is the slowest in participation due to the relatively largest accumulated audio output time and clock synchronization offset. We therefore set it's scheduling latency to zero and deduce τ_2 and τ_3 appropriately.

We deduce the required values for τ_2 and τ_3 with the measure results. Note, there are multiple ways to derive the calibration performance, as mentioned in at the definition of the calibration performance in sub-subsection 3.2.5. We outline each of them below.

$$
\begin{aligned}
p_{12} &= (e_1 + \text{ø}_1) - (e_2 + \text{ø}_2) &&= \left(x_{11} - x_{21} + \frac{(x_{12} + x_{21}) - (x_{11} + x_{22})}{2} \right) \\
p_{12} &= -((e_2 + \text{ø}_2) - (e_1 + \text{ø}_1)) &&= -\left(x_{22} - x_{12} + \frac{(x_{21} + x_{12}) - (x_{22} + x_{11})}{2} \right) \\
p_{13} &= (e_1 + \text{ø}_1) - (e_3 + \text{ø}_3) &&= \left(x_{11} - x_{31} + \frac{(x_{13} + x_{31}) - (x_{11} + x_{33})}{2} \right) \\
p_{13} &= -((e_3 + \text{ø}_3) - (e_1 + \text{ø}_1)) &&= -\left(x_{33} - x_{13} + \frac{(x_{31} + x_{13}) - (x_{33} + x_{11})}{2} \right)
\end{aligned}
$$

3.2.6.1 Algorithm performance optimization algorithm

With the math outlined, we can describe the algorithm for implementation. Note that for any two given devices a and b, we got p_{ab} equals $-p_{ba}$. We find the slowest device x, for which every other device y the value of p_{xy} is negative. The device x takes on a scheduling latency of zero, and each other device y takes on the the scheduling latency $-p_{xy}$.

The algorithm can be made more robust in the absence of microphones. Consider for example only device **1** to have a microphone. Under this assumption, we must neglect the distance measure (as mentioned in 3.2.5.1). Compare each device y with device **1** to obtain the highest value of τ_{1y}. The slowest device (which could be device **1**), we name x, and each device y takes on the scheduling latency $\tau_y = \tau_{1x} - \tau_{1y}$.

3.2.7 Bonus: No alternative approach with given measures

Before I approached the problem to solely deduce the calibration performance, I attempted on obtaining knowledge about e_1, ø_1 and r_1 separately. As we outlined in the previous paragraphs, we don't need their respective information, as we only care about the combined latency of e_1 and ø_1. However, let me emphasize why I think it isn't possible to deduce the calibration-variables values under our defined model and test cases.

After calculating the distance with 3.2.4, we remain with six unknown variables, namely the emission, receival, and server offset values. Therefore, to simplify further calculations, we neglect the distance variable from our equations.

$$
\begin{aligned}
x_{11} &= e_1 + r_1 \\
x_{12} &= e_1 + r_2 + \text{ø}_1 - \text{ø}_2 \\
x_{22} &= e_2 + r_2 \\
x_{21} &= e_2 + r_1 + \text{ø}_2 - \text{ø}_1
\end{aligned}
$$

We can translate these equations into a matrix to solve.

e_1	e_2	r_1	r_2	$\emptyset_1$	$\emptyset_2$	x_{11}	x_{12}	x_{21}	x_{22}
1	0	1	0	0	0	1	0	0	0
1	0	0	1	1	−1	0	1	0	0
0	1	1	0	−1	1	0	0	1	0
0	1	0	1	0	0	0	0	0	1

Preferably we could solve this matrix and find all or some of the unknown variables of the left matrix. However, since we have more unknown variables than equations, nor can we cancel out due the singularity of this matrix, we aren't able to find a unique solution. And since we already have an identity matrix on the right side, we won't be able to find more linearly independent equations.

We assume emission and receival (e_1, e_2, r_1, r_2) to be static variables, also the only dynamic variables are $\emptyset_1$, $\emptyset_2$. As x_{11} and x_{22} depend solely on the static variables, this implies x_{11} and x_{22} won't change if we do additional measurements. Any change we would apply in either $\emptyset_1$ and/or $\emptyset_2$ will result in linear change in both x_{12} and x_{21}, which we can also conduct theoretically without having to do an additional measure.

Also, from this theoretical perspective, additional measurements won't result in more information about our unknown variables. We therefore conclude we aren't able to find the exact values for these unknown variables from the measures we have defined this far.

We could attempt to solve the above defined linear system by finding the most-likely values each variable could take. Even though we could make assumptions, or have knowledge about latencies of certain devices, it's questionable how this knowledge can be obtained from a mathematical point of view. Namely, the problem isn't statistical, nor can we assume any relation to exist between e_1, e_2, r_1 or r_2. Therefore, using standard linear approximation techniques, such as searching for the minimal euclidean distance, we therefore consider not to be applicable to our problem.

3.3 Calibration topology

The calibration model outlined in the previous subsection 3.2 has focused on how calibration can be computed between two devices both having a speaker and a microphone. In this subsection we elaborate on the calibration under more diverse spatial circumstances.

In practice, devices may not be connected with a microphone, or devices have a microphone but not a speaker, or a device may have multiple speakers and/or microphones. Besides the available hardware, certain devices may be in range of one another, while others are not. We therefore could end up with different scenarios under which different topologies may function or outperform others.

We assume to be operating on a sensor network with fixed sensor nodes, assume accurate measures, and no drifting/variance in measure results. This simplifies in such we aren't in need for continuous measuring and analyzing, and additionally do not have to introduce a separate statistical model to support error reduction.

3.3.1 Neglecting distance latency

The most important differentiation is whether distance latency is trivial, such as in a small-sized room.

3.3.1.1 Trivial case

In case a single microphone can oversee the entire system, as in figure 16, we have a trivial case to compute the latency output latency. To rephrase the trivial case, it is when all transmitters are detectable by a single receiver. The formula defined in 3.2.6.1 can be directly applied and we are set.

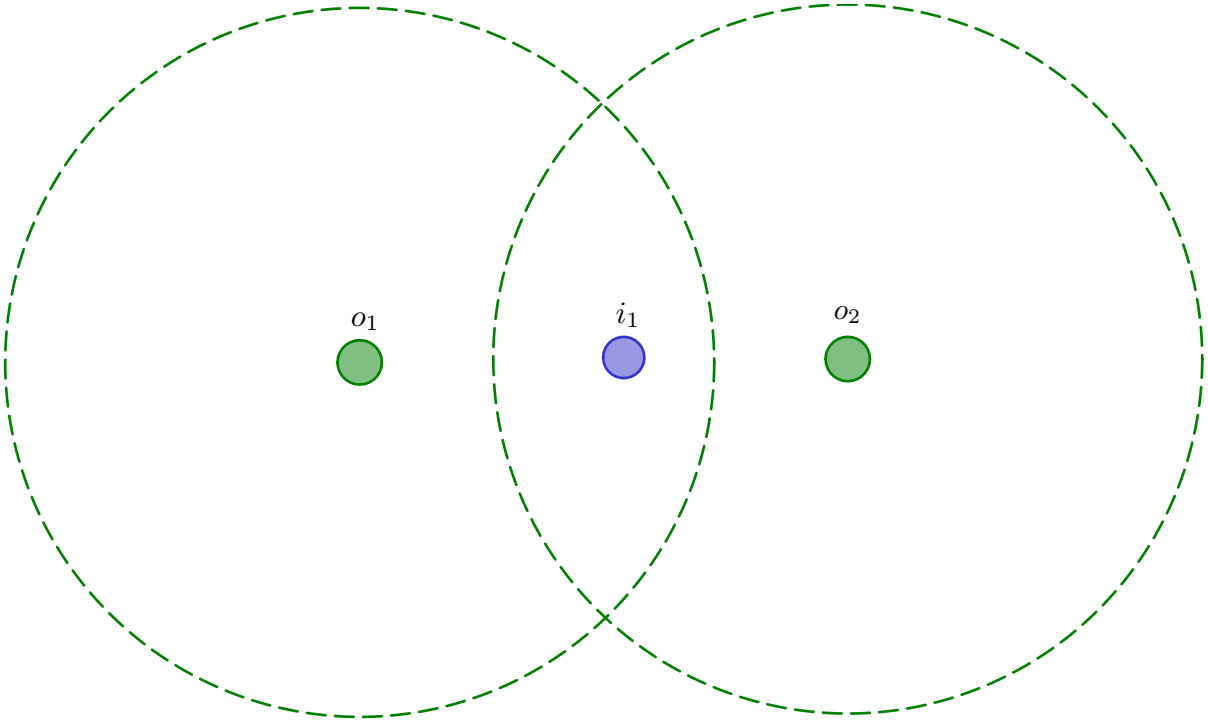


Figure 16: Trivial example of a scenario neglecting distance

3.3.1.2 Non-trivial case

A minimal case occurs when multiple microphones are involved, and at least one speaker isn't in range for each participating microphone, as in figure 17.

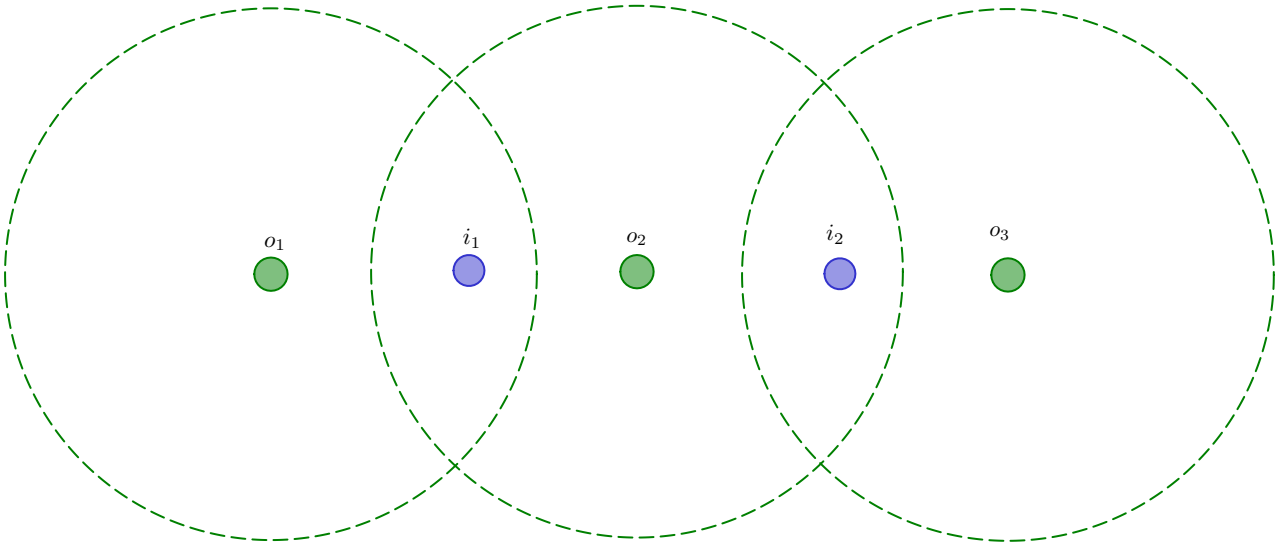


Figure 17: Non-trivial, yet minimal example of a scenario in need of a topology.

The speaker o_3 cannot be detected by microphone i_1, and the speaker o_1 cannot be detected by microphone i_2. There isn't knowledge available on the input latency of both microphones, therefore in order to relate the latency of i_1 with i_3, we require the microphone input latency difference between i_1 and i_2. In other words, we want to relate all output devices to a single input device: in such that the input latency differences aren't influencing the calibration performance.

As both i_1 and i_2 detect the signal emitted by o_2, we can use their results x_{21} and x_{22} respectively to deduce difference of input latency of i_1 and i_2. We define the input latency difference between the two inputs with

$$\delta_{i_1 i_2} = x_{21} - x_{22}$$
$$\delta_{i_2 i_1} = x_{22} - x_{21}$$

Then we are able to relate all three output speakers with both input 1 or input 2.

$$x_{31} = x_{32} + \delta i_1 i_2 = x_{32} + x_{21} - x_{22}$$
$$x_{12} = x_{11} + \delta i_2 i_1 = x_{22} + x_{22} - x_{21}$$

And from this point on we can calibrate the results again with the calibration formula described in 3.2.6.1.

To describe the formula more abstract, for any input i_a and input i_b, we can transfer any measure from one input device to the other, if there exists a speaker o_o in such the measure x_{ca} and x_{cb} exists.

There seems to be an advantage if nodes are controlled by the same device: they are time synchronized. Yet, time synchronization is a part of the latency and therefore doesn't conceal actual useful information, and therefore we consider it insufficiently to integrate into the model. Therefore, it seems applicable to consider the whole as a virtual input array and a virtual output

array, such as done in [32], which combines any kind of microphone to add it to the bundle make it contribute to analyzing what is being said in a conversation/meeting. This clearly adds a centralized component in the calibration process.

To give a example of clusters, consider the following spatial setup that consists of three separate clusters.

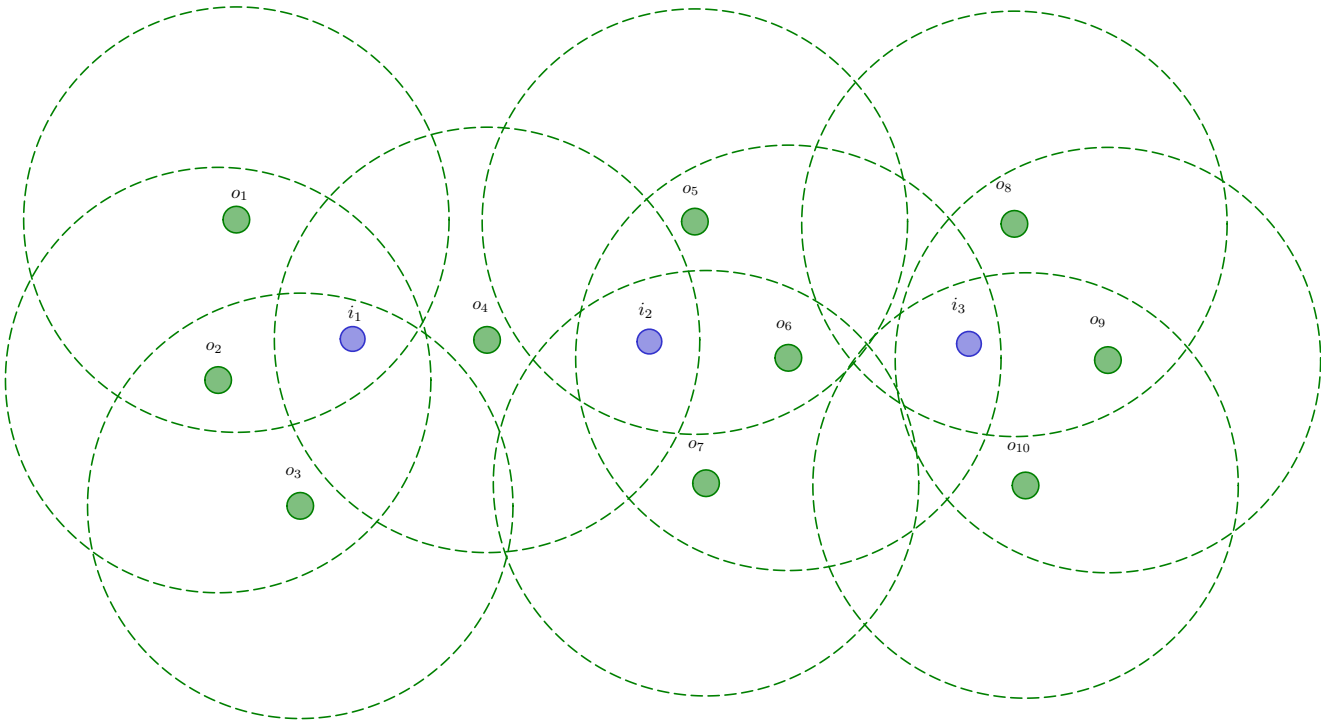


Figure 18: Multiple clusters

Placing them in a matrix

	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8	o_9	o_{10}
i_1	a	b	c	d	0	0	0	0	0	0
i_2	0	0	0	e	f	g	h	0	0	0
i_3	0	0	0	0	0	i	0	j	k	l

Non-zero values in columns connects clusters. First, we could obtain output latency variables for the three separate clusters, which relate to each microphone. Due o_4, the cluster that results of i_1 and i_2 can be merged into a single cluster. And due o_6, the results of i_2 and i_3 become accessible as well, merging the whole into a single cluster.

3.3.2 Considering distance latency

Even if we are neglecting distance in our model, we could still benefit from considering distance latency to be a part of the network. Namely, in the last example in figure 18, if there would have been more clusters, devices could have large distance differences which would strongly impact

perceived playback synchronization. However, one could argue these distance latencies won't impact the listening experience, since the outputting speakers are seemingly this far and thus heard softly, the microphones didn't manage to detect their emitted signal, nor will the users.

Another argument against considering distance latencies to be relevant, is, which point in space should the devices be calibrated to? Consider a person A standing next to speaker o_1 and another person B standing next to speaker o_{10} from figure 18. If we would decide to calibrate in order to optimize the calibration performance for person A, that would imply further lying outputs, such as o_{10}, will have to compensate their distance traveling time by decreasing their relative output latency in contrast to other, more nearby speakers. Person B however, would require the exact opposite output latency compensation. The more the spatial calibration point is moved towards the left, the worse the perceived calibration performance becomes as we move to the right. Therefore, we decide to fully neglect the presence of distance latency.

Under certain scenario's there may still be a preference to consider the spatial calibration point. If there is an interest to consider distance, and the connectivity information is available as we have described above, a promising approach that claims to cover exactly that would be multidimensional scaling (MDS): *"position estimation using mere connectivity information. Additional distance estimation may contribute, and the algorithm works even if no beacon nodes are available in which case relative coordinates are generated for the dumb nodes."*([33], Chapter 5.4 over MDS, page 36).

3.4 Location estimation

An interesting property of calibrating devices could be to locate devices in space. If we manage, for instance, in the browser a calibration precision of 1 milliseconds, in theory that would induce a devices can relate it's position to other devices with an precision of 0.3 meters.

We explore whether a web-application could provide a solution to obtain location estimation. The challenge to deduce the positions of devices is extensively researched in radar positioning networks, therefore we use this domain knowledge.

3.4.1 Location estimation specifications

Multiple aspects have to be considered, outlined in [33]. To begin with, is the goal to obtain range-based or range-free/proximity-based results? If clusters have to behave in a distinct manner, such as distinguishing different rooms to apply different sound effects or playing different songs, a course-grained solution is required. If a device has to be localized more precisely in space, a fine-grained solution is required.

Afterwards, will devices function fully autonomously - thus we end up with a distributed network - or do they address directly a server - and thus the network is centralized. Autonomy is as well interesting from the perspective of the end-user. Functionality choices could decide on what a participant is allowed to do in a session: who decides in a collaboration effort what song will be played? Does the entire system wait for devices to be ready, or should uncalibrated devices mute themselves?

The network may function in a indoor constrained environment or an outdoor unconstrained environment. Additionally, nodes in the network could be statically positioned, yet most likely will be mobile as users walk around with their device.

3.4.2 Solution directions

Without providing solutions, we consider a few directions to location estimations. Which is, considering Bluetooth for position estimation, the approaches RSS, ToA and TDoA, An obvious alternative to speakers and microphones as the receivers/transmitters in the position network, would be considering Bluetooth to complete this task. However, under the current standards it is questionable how scalable this approach is: communication initialization is cumbersome due standardized security policies. Additional, the bandwidth in which Bluetooth functions limits 16 separate channels. This makes the approach inaccessible.

Currently we have considered a centralized approach. However, once many devices could participate in a single session, such centralized network communication could become a bottleneck. When such a turning point is reached - that network communication becomes the factor for performance degradation - autonomous/decentralized/distributed communication becomes more interesting to consider as a solution.

Besides signal detection, more approaches to distance measuring could be considered.

- Received Signal Strength Indicator (RSS)
- Time of Arrival (ToA)
- Time Difference of Arrival (TDoA)

Relying on Received Signal Strength (RSS) is very hard to do in our context, since speaker loudness is too variable: users could change the volume on the fly. Additionally, there is no standard available on the loudness of mobile device speakers in the context of a web-browser. Therefore, we fully neglect this as a viable parameter for location estimation.

We use the Time of Arrival in the calibration process outlined above. However, time of arrival is only useful once input nodes are synchronized. Under the hood, the time of arrival is depending on the time differences of arrival in relation to a certain output node. Therefore, ToA is not applicable either.

The only approach which remain is using the time difference of arrival. TDoA can be separated into

- Multi-node TDoA
- Multi-signal TDoA

In the multi-signal TDoA, instead of a single signal, multiple signals - on different frequencies - are emitted by a speaker. As signals travel at different speeds depending on their frequency, the time difference of arrival can be used to deduce the distance of the node. However, as we work with default consumer speakers, the speed difference between low and high frequencies in the bandwidth of 20 Hz to 20 kHz is too small for accurate results.

3.4.3 Multi-node TDoA

So the only option to consider as a location estimation approach is multi-node TDoA. Unfortunately, we haven't got knowledge beforehand what the exact audio input latencies are. Without this information, we won't be able to apply multi-node TDoA. Therefore, we need speakers that are detected by multiple microphones in order to gain awareness of relative input latencies for microphones. However, we aren't certain where exactly such a speaker is placed in relation to these microphones, also there still isn't sufficient information available to decide on how much transfer latency is involved.

3.4.3.1 Node types

At this point it becomes usefully to name different type of nodes in the network.

- **full** network nodes: contains both a microphone and a speaker
- **silent** network nodes: contains only microphone
- **deaf** network nodes: contains only speaker

Full nodes provide more information, as there is no transfer distance in a self-measure. We have already shown that two full nodes can obtain the distance to each other with the formula outlined in 3.2.4. Due to the incomplete knowledge on input latencies, output latencies, and positioning, both beacon nodes and full nodes could greatly contribute to filling in the gaps. Due time limitations we won't elaborate further on location estimation.

4 Proof of concept

We build a reference prototype as a proof of concept to test whether automatic, synchronous audio playback collaboration is possible within a web browser. We apply the signal detection algorithm outlined in 3.1 and the calibration model outlined in 3.2.

First we describe some interesting things on important audio components in our prototype. Afterwards, we describe how the implementation functions. We test the proof of concept, and as last conclude on the functioning of the prototype.

4.1 Time

In the web-browser we have three concepts to time. The first is `Date`, which tracks the system time. The second is `performance`, which tracks the lifetime of a thread. The third is the `currentTime` property of the audio context, which tracks the lifetime of the audio context.

Audio scheduling depends on the time of the audio context. Since this time is ever-increasing, scheduling has to be done in the future: at a later point in time that it currently is at the audio context. Therefore we must use the audio context time.

Additionally, communication handled between devices can rely on their system time or thread lifetime. Both should be sufficiently accurate. `performance` should be faster, since it doesn't require a API call to the system. Additionally, the API specification mentions it could have microsecond precision. However, most web-browsers have this precision purposefully lowered to prevent both fingerprinting and security threats [34].

4.2 Recording

In order to calibrate we have to retrieve microphone input and process it accordingly.

4.2.1 User media stream

The first step is to retrieve user-media.

```
let mediastream = null;
export async function get_mediastream () {
  if ( window.isSecureContext == false ) {
    throw Error( "Insecure window context, microphone inaccessible." );
  }
  let mediastream = await navigator.mediaDevices.getUserMedia({
    audio: {
      noiseSuppression: false,
      echoCancellation: false,
      autoGainControl: false,
      sampleRate: 44100,
      sampleSize: 128,
      channelCount: 1,
    },
    video: false
```

```
  });
  return mediastream;
}
```

Listing 2: Retrieve user-media audio stream JavaScript implementation

First ensure the web-browser considers itself to be processing within a secure environment, otherwise functionality such as microphone input will be disabled (as is the case in most browsers).

Disable the noise suppression, echo cancellation, autoGain control, and provide sample rate with sample size and channel count. These parameters are provided as a request, the actual parameters applied therefore may differ.

Cache the result and re-use the mediastream on subsequent calls. To close down a mediastream in our audio case, close each `mediaTrack` in the stream with a code line such as `mediastream.getAudioTracks().forEach(function(track)  track.stop());`.

4.2.2 Audio context

We need an audio context to process audio.

```
let context = null;
export default function get_context () {
  if ( context ) { return context }
  context = new AudioContext({
    sampleRate: 44100
  });
  return context;
}
```

Listing 3: Retrieve audio context

Specify the sample rate to 44100. In the application only a single audio context is created, cache the result and re-use the same audio context on subsequent `get_context()` calls. To close down a context, call `context.close()` anywhere in the code.

4.2.3 Audio node

We convert the media-stream into an audio node that we can process, do so with the `let node = await context.createMediaStreamSource( stream )`. Now we can connect our node to an audio processing node with `node.connect( processor)`.

4.2.4 Processing node

4.2.4.1 AudioWorkletNode and ScriptNode

The approach to writing customized audio nodes is with the AudioWorkletNode, which runs in a separate thread. Older web-browsers don't support this feature, and instead require to define a ScriptProcessorNode that runs on the main thread.

Two poly-fills are available in the wild to substitute the audioWorkletNode functionality on older browsers that do not support that new web API. One poly-fill approaches it by emulating a

AudioWorkletNode by forwarding all processing of a ScriptProcessorNode to a Web Worker [35]. Even though limited, this has the advantage to support multi-threading.

Another poly-fill approaches it by, in addition to emulating the AudioWorkletNode, as well emulating a Web Worker on the main thread [36]. This solution provides zero performance gain, and is only meant to not having to rewrite logic to support deprecated web-browser versions.

We stick with the AudioWorkletNode. This node itself doesn't do any processing, instead it directly forwards all incoming audio data (from the microphone) over it's port to the main thread.

```
class MonoAudioStream extends AudioWorkletProcessor {
  process (inputs, outputs) {
    this.port.postMessage( { frame: inputs[0][0], time: Date.now() } );
    return true;
  }
}
registerProcessor('mono-audio-stream', MonoAudioStream);
```

Listing 4: AudioWorkletNode script

Note how we add a self-generated timestamp to preserve the time receival information. We will explain this choice in 4.2.4.2.

In the main thread we listen for incoming events transmitted over the AudioWorkletNode-port, and process the data accordingly.

```
let timestamps= [];
let recording = [];
let processor = new AudioWorkletNode( context, "mono-audio-stream" );
    processor.port.onmessage = ( evt ) => {
      let { data: { frame, time } } = evt;;
      timestamps.push( time );
      recording.push ( evt.frame );
    }
```

Listing 5: Processor node

4.2.4.2 Input latency due event handling

The process of sending data from the AudioWorkletNode to the main thread takes time. If this time is fixed it wouldn't deteriorate the accuracy, it just only adds up to the total audio input latency. However, if we take a close look at the flame-chart of the event-handling of the autoworker communication, we see a non-negligible deviation. The behavior is shown in figure 20.

Audio frames in the AudioWorkletNode are processed with the exact size of 128 of samples, and each frame is encapsulated and forward over an port message event. Ideally, each event would be forward directly, and the flame chart would only show behavior of exactly one message every 2.9 milliseconds (under a sample rate of 44100 Hz). However, instead events are are sometimes stalled and afterwards send together in batches.

Figure 19: Flame chart of audioWorkletNode port handling with 20 milliseconds of latency for two subsequent events to arrive and get handled on the main thread.

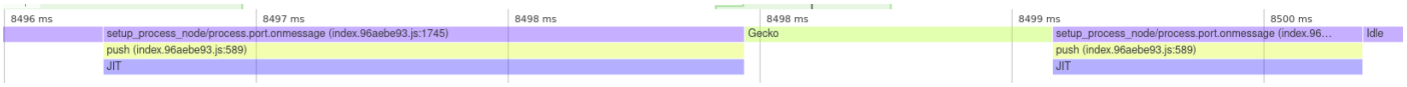


Figure 20: Flame chart of audioWorkletNode port handling with 3 milliseconds of latency for two subsequent events to arrive and get handled on the main thread.

4.2.4.3 Preserving time receival information when forwarding data to the main thread

In order to circumvent the inconsistency in time of arrival, instead of relying on the time of arrival, we can rely on the timestamp of the event creation. Every event in the JavaScript environment carries along a timestamp property of it's time of creation. This timestamp is generated with `performance.now()` of the running thread. This enables us to circumvent the speculative latency introduced by the port-message forwarding and we can ensure to have accurate knowledge of the original time of arrival.

Due convenience we rather use `Date.now()` than `performance.now()`, even though this API call may be a little slower/accurate. As an additional excuse besides convenience, converting `performance.now()` to it's counterpart in `Date.now()` would as well take time and may end up to be just as inaccurate. To realize this, a self-generated timestamp is send along the frame data in the event message.

An additional measure we could take to reconstruct the time of arrival, is by calculating the average time per sample and applying this to deduce which sample arrived when. In the calibration procedure, we use this to improve

4.3 Playing

Playback of audio is scheduled on the audio context in relation to it's time. Schedule a AudioSourceNode to playback the target signal at a specified time. This is done by writing first audio data into a BufferSourceNode.

```
1 export default async function signal ( sound, time ) {
2   let context     = get_context();
3   let buffer      = context.createBuffer( 1, sound.length, 44100 );
4       buffer.copyToChannel( sound, 0 );
5   let node        = context.createBufferSource();
6       node.buffer = buffer;
7       node.start  ( time );
8       node.connect( context.destination );
9 };
```

Listing 6: Playback a signal

Time is handled as seconds, and actual time to wait to play depends on the readable property `currentTime` of the audio context. Any audio scheduled *before* the current time of the audio

context will fail and throw an error.

4.4 Calibrating

The proof of concept supports to calibrate two devices for which we assume to have both a microphone and a speaker, close in range in such we can neglect their distance.

4.4.1 Server time offset

Before we can continue with signal detection we have to know at what time each device is supposed to act. In our implementation we have chosen the server time to be the reference time point on which moments of action should be related to.

4.4.1.1 NTP

We synchronize the internal time of each device with the server with an overly simplified variant of NTP. We skip all procedures and variables of the NTP-specification [37], and just copy their naming for time-stamping, round-trip latency, and clock offset.

```
let socket = io ( address );
let t1 = Date.now();
socket.emit( "NTP", t1, function( t2 ) {
  let t3 = Date.now();
   delta = t3 - t1; // round-trip latency
   theta = 0.5 * ((t2 - t1) + (t2 - t3)); // time offset
} )
```

Listing 7: NTP client code

```
    import socketio from "socket.io";
const io = socketio( server )
      io.on( "connection", socket => {
        socket.on( "NTP", async ( cb ) => {
          cb ( Date.now() );
        });
      });
```

Listing 8: NTP server code

We expect a round-trip latency below 100 milliseconds, and we are set. Otherwise we fail the time measure, and thus must retry before we can continue with the calibration procedure. We add 200 milliseconds safety-time to the preparation time-frame in the measure phase, to be certain there won't be a signal detection miss due overly large clock differences: e.g. to prevent the scenario that device 1 emits this far up front that device 2 isn't even recording yet.

4.4.2 Target signal

Providing the target signal is done with fetching a WAVE-file (a song of 4096 noise samples) from the web-server and decoding it. We store it under a 441000 Hz sample rate, mono channel.

4.4.3 Deducing latency

The recorder records 1700 milliseconds of audio, to cover for a variety of latencies in order to detect the target signal:

- 200 milliseconds due potential clock differences.
- 800 milliseconds of audio output latency.
- 200 milliseconds of signal traveling.
- 500 milliseconds of audio input latency.

We have fixated these values, to simplify prototyping the proof of concept. We have chosen these values to be quite large, to increase the likeliness that both devices used in the proof of concept are fast enough to reach time deadline at each step in the process.

4.4.3.1 Microphone preparation

On a calibration request, we initiate the recorder as quickly as possible, and then test some things before we consider the incoming audio input to be valid:

1. Is audio input processed by the audioWorkletNode? If not, the data stream is undefined.
2. Is the audio input unmuted and receiving audio? If not, the data stream consists of only zero-valued signal samples.
3. Is the audio input stabilized? If not, audio input is glitched: sample values fluctuate enormously, and the signal samples don't correctly reflect the actual audio environment.

In such, audio input is discarded as long as it is either streaming only sample values of undefined or zero. Once non-zero audio data starts streaming in, we enter a "stabilization" period, in which we wait 200 milliseconds to settle on the glitches. After this point forward we just assume the audio to be stable and correct.

The task of recording is initiated with the same time as the playback for the target signal. Due the stabilization period, the recorder must have started earlier than the signal scheduler. In case the recorder is prepared earlier, we drop all incoming audio with a timestamp of a 100 milliseconds or earlier than the start time. Afterwards we record (in addition of these first approximate 100 milliseconds) another 1700 milliseconds.

Any incoming audio data with a time-stamp that surpasses the end time, is discarded. Additionally, once this happens, the microphone input node is disconnected from the processor node. We deduce when the first sample of this 1700 ms recording occurred, then we slice from that index the 74970 samples to obtain 1700 milliseconds of recording.

4.4.3.2 Cross-correlating

Consider this slice of 1700 milliseconds to be the source, and cross-correlate it against the target signal. The index of the cross-correlation peak can be directly translated into the audio receival latency.

4.4.3.3 Validating the calibration step

The calibration process should be invalidated if:

- The microphone isn't receiving non-zero audio data 400 milliseconds before the time of signal emission. Namely, we require 200 milliseconds of stabilization and 200 milliseconds of potential time clock difference. This step could be left out, namely if it did miss the signal detection it wouldn't correlate in the first place, which brings us to the second reason for invalidation:
- The cross-correlation peak/average ratio is below 10. As we have found in 3.1.5, this ratio was sufficient to correctly classify a cross-correlation peak to relate to the target signal.

4.4.4 Additional implementation details

The goal of the full calibration process is to deduce the following three variables:

- `theta` : Time offset to server.
- `tau` : Time difference to compensate for audio latency and server time offset (theta).
- `delta` : Time difference to the audio context time.

With this information, we should theoretically be able to schedule audio and successfully play back audio synchronously over multiple devices.

The implementation is aimed at realizing the experiment that we further describe in 4.5. We consider two devices to participate, both with a microphone and speaker connected.

We obtain `theta` with the time request outlined in 4.4.1.1. Afterwards, the calibration measure is initiated to emit the signal from device 1 or from device 2. Whether only detecting, or additionally emitting the signal, both devices receive the same server time that mentions when the calibration test starts. Therefore, both devices require to have knowledge of `theta`, otherwise they haven't got sufficient knowledge to be certain in what time frame the other device may start emitting the audio signal. The administrator of the session can decide which device emits the target signal, and has the option to repeat it, for example in case a device fails to detect the emitted signal.

When both devices have a successful measure of their own and the other device their target signal, the server requests their times, calculates τ_1 and τ_2, sends it back to the users, and the users are able to schedule their audio synchronously with the formula `node.start( servertime + delta + theta + tau )`.

We will further elaborate on `delta` in the subsection about audio context time drift in 4.9.

4.5 Experiments

With each component described in the previous subsection, we can build the proof of concept. We do four separate tests, each involves both a laptop and a phone that play either play together calibrated or uncalibrated The phone is a Motorola Moto G5, running Chrome for Android version 84.The laptop runs on an Intel i5-1035G1, 8 cores, at 3.6GHz, using PulseAudio with ALSA as the sound system, Firefox version 78 (64-bit)

The experiment didn't work with Firefox for Android, some issue occurred with the preparation phase. Therefore instead Chrome was picked as the web-browser.

Both devices start outputting clicks, scheduled under 500 millisecond intervals. The level of calibration is thereby clearly hearable unveiled by the difference of the time of output between the clicks of both devices. This activity is recorded and these results are outlined below in here in subsection 4.6.

Audio output loudness, microphone input sensitivity, silent surrounding, stable internet connection, were are manually set to create an ideal environment.

4.6 Results

The four tests performed are:

1. The phone over Bluetooth-speakers and the laptop over internal speaker.
   (a) Uncalibrated playback, in figure 21
   (b) Calibrated playback, in figure 22
2. The phone over internal speaker and the laptop over internal speaker.
   (a) Uncalibrated playback, in figure 23
   (b) Calibrated playback, in figure 24

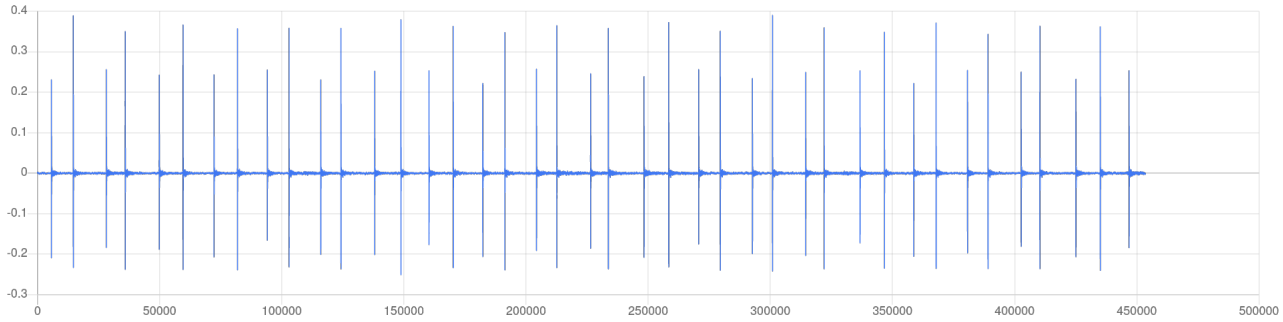


Figure 21: Uncalibrated audio playback with the internal speaker of a phone and the internal speaker of a laptop.

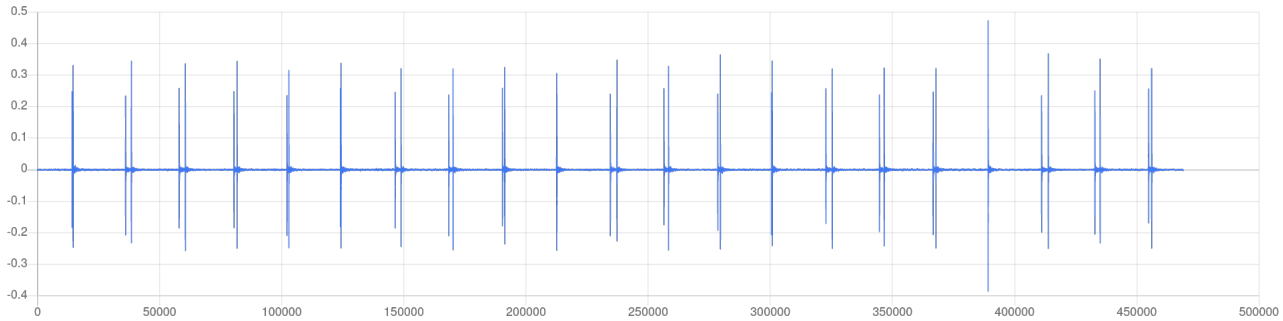


Figure 22: Calibrated audio playback with internal speaker of a phone and internal speaker of a laptop.

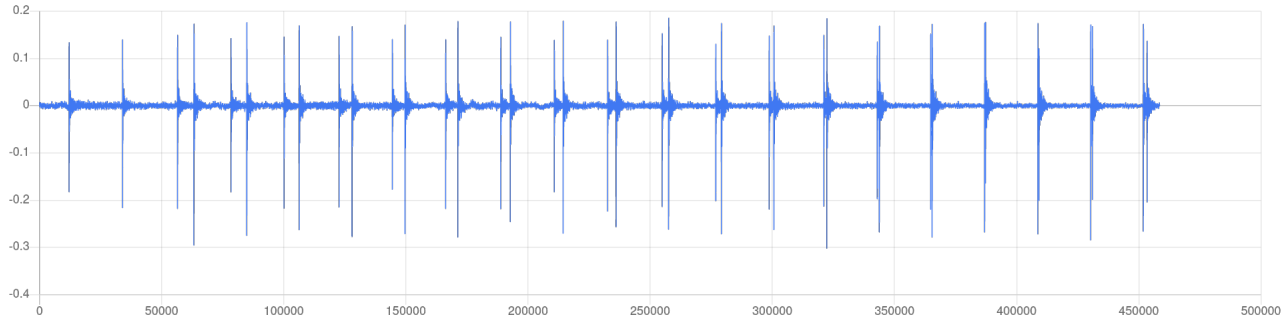


Figure 23: Uncalibrated audio playback over Bluetooth-speakers connected to a phone and internal speaker of a laptop.

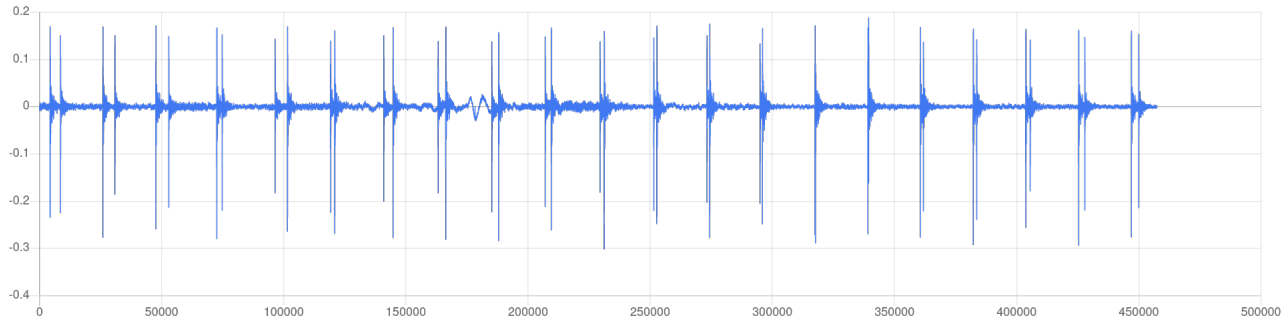


Figure 24: Calibrated audio playback over Bluetooth-speakers connected to a phone and internal speaker of a laptop.

4.7 Comparing synchronization performance of calibrated- against uncalibrated audio scheduling

If the phone and laptop both use their internal speaker, the offsets between the peaks are clearly smaller once calibrated, and quite regularly, clicks are emitted by both speakers fully synchronously. In contrast, in uncalibrated audio scheduling, the difference of playback is approximately 200 milliseconds, and additionally, clicks are never emit synchronously.

If we use Bluetooth-speakers with the phone, the uncalibrated playback results are far worse in consideration of the synchronization performance. In figure 23 there are three clicks visible of the internal laptop speaker, once the first click is output over the Bluetooth-speaker. Also, the difference of output between the two devices is over 1500 milliseconds. In contrast, if we apply our calibration technique and schedule the clicks calibrated, the clicks are sometimes output fully synchronously, and otherwise stay below a difference of approximately 150 milliseconds.

4.8 Variance in click offset differences

During playback, whether calibrated or not, we only require static variables. Yet, in each result we see a variation in the click offset differences. The test-results of the experiment with the Bluetooth-speaker involved, clearly shows a larger amounts of deviation between the

synchronization performance. Since this behavior cannot be declared with the simplified model, and each signal is scheduled beforehand on a 500 millisecond interval, the only point that could introduce this variation is the note scheduling line of 6.

4.9 Audio context time drift

In order to see whether the audio context drifts in relation to the thread lifetime, we have compared the drift between `performance.now()` and `context.currentTime` over a time span of a thousand seconds. The test code is listed here.

```
1 let results = [];
2 let context = new AudioContext();
3 let reference  = performance.now() - ( context.currentTime * 1000 );
4 for ( let i = 0; i < 1000; i ++ ) {
5    results.push( performance.now() - ( context.currentTime * 1000 ) - reference
      );
6    await sleep( 1000 - ( performance.now() % 1000 ) ); // [1][2][3]
7 }
```

Listing 9: Drift test `performance.now()` with `context.currentTime`

1. This is run in an asynchronous function.
2. `sleep` is a simple sleeping function:

```
1 export default function sleep ( ms ) {
2   return new Promise( ( resolve, reject ) => {
3     setTimeout( resolve, ms );
4   });
5 };
```

Listing 10: Sleep milliseconds

3. Pick `await sleep( 1000 - performance.now() % 1000 )` to start a new test on the beginning of the second. `await sleep( 1000 )` results in a small additional delay that accumulates over time, and thus won't necessarily start at the beginning of each second.

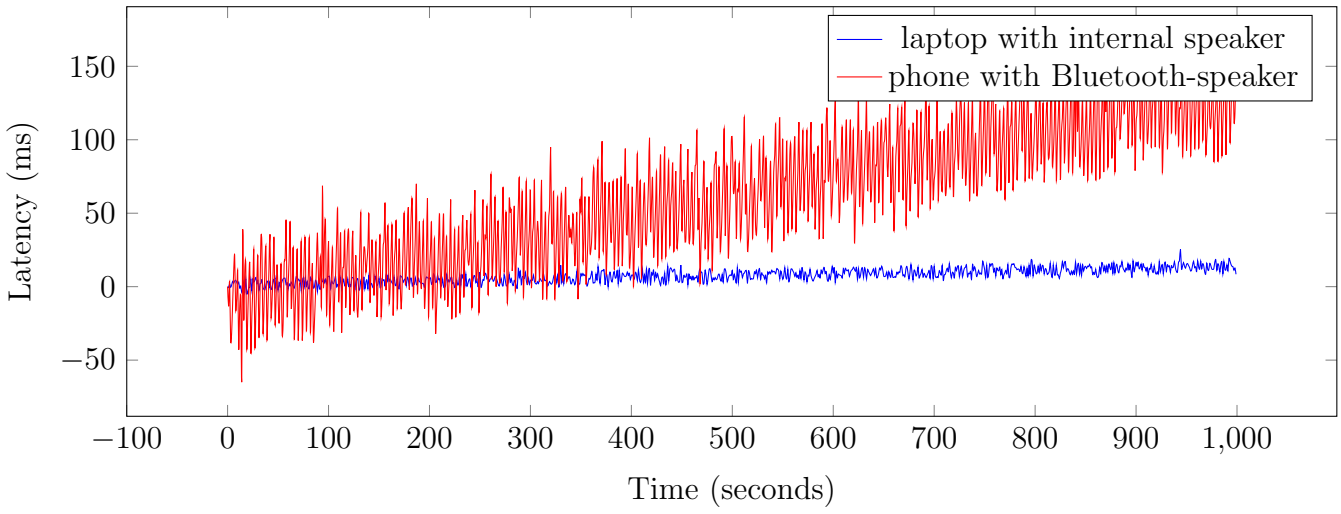


Figure 25: Drift results of `performance.now()` with `context.currentTime` for both a laptop with it's audio output connected to it's internal speakers, and a phone with it's audio connected to a Bluetooth-speaker.

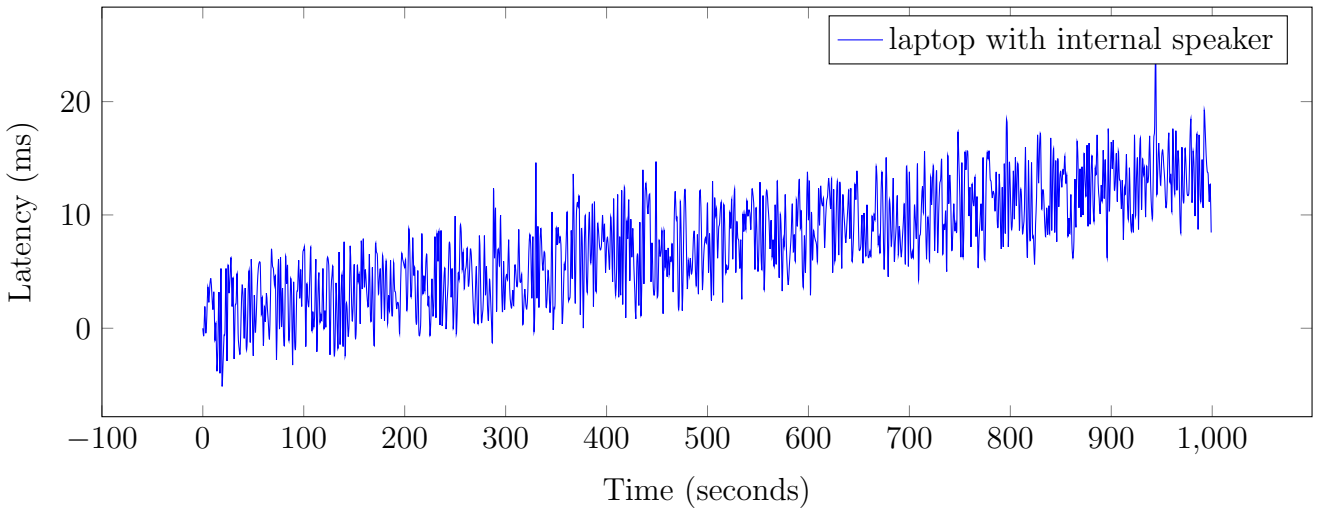


Figure 26: Zoomed in drift results of `performance.now()` with `context.currentTime` of the laptop, outputting it's audio to it's internal speakers.

In addition the test is repeated for a phone without using Bluetooth-speaker output.

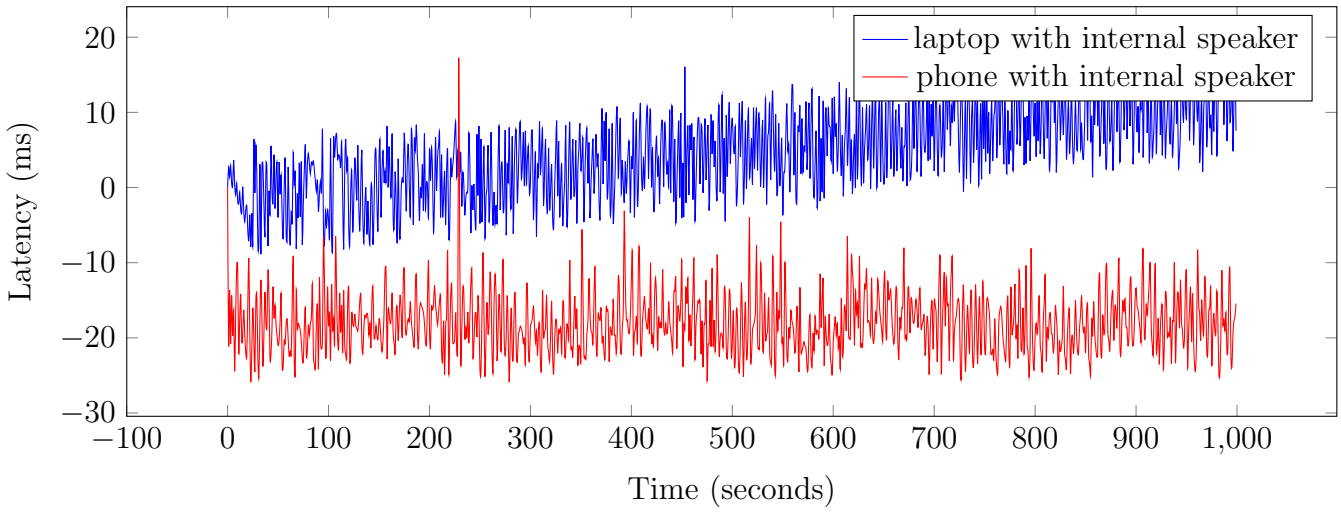


Figure 27: Drift results of `performance.now()` with it's audio output connected to it's internal speakers, and a phone with it's audio connected to it's internal speakers.

This drift in latency seems to be quite problematic as the `context.currentTime` cannot be neglected as audio playback scheduling directly depends on it. Even if we attempt to remove the minimize the use of this variable, for instance by to once calculate an offset in relation to `performance.now()`, and to further on only depend on `performance.now()`, devices will still deviate: As we can clearly see in the graph 25, after a thousand seconds the phone will have got an additional approximate hundred milliseconds audio output latency in comparison to the laptop.

If we are able to measure the drift continuously, we might be able to stabilize the time difference to the audio context time (the `delta` variable) by statistically deducing the context audio drift at a certain point in time. The drift seems to change linearly over time with quite uniformly distributed variation. This is convenient, as possibly a least square fitting algorithm may suffice to tackle this problem. We won't implement and elaborate on this due time constraints.

4.10 Neglecting context drift

Thus, if we neglect the presence of the audio context time drift, will the calibration result in successful synchronization? Yes, in both test-cases, in the stream of clicks, at least one click was output in both speakers fully synchronous. In specific, if we consider an output latency difference of over 1500 milliseconds if no calibration is applied, it emphasizes the use for this technique to synchronize devices on their audio playback.

5 Provisioning audio-related activities

With a proof of concept available, and a more clear picture of the capabilities of audio handling under the current state of web-applications, the question arises what audio-activities could be provisioned with a web-application. Therefore we have to know whether a web-application could provision the performance requirements.

First we define the performance requirements of synchronous, co-located, concurrent audio-related activities. Afterwards, we consider classifying the range of activities that could be performed, in such we are able to relate classification to performance requirements. however, due to the incompleteness of the available classifications, and due to not being able to provide a solid relation of any such categorization to the our performance requirements, we define a benchmark for an activity to be benchmarked on it's fitness to be implemented by a web-application.

5.1 Performance requirements for activities

Note, we won't define here what user-interaction is expected, nor the behavior of the application, nor the behavior of collaborative efforts. Instead, we only consider what the specifics of the performance requirements are for the whole (and thus each separate device) to function successfully.

Additionally, note how requirements could be asymmetrical, as certain devices may have to be more performant than other devices. As well note how activities may support degradation, by offering an alternative solution if certain requirements cannot be met.

Considering the performing requirements, at least three different types of performance requirements could be considered. First, is time-criticality, which consists of response speed and scheduling accuracy. Secondly, is the scalability, which consists of inter-device communication, information load, and group size. Thirdly, is the computative load, which consists of processing power and memory size.

5.1.1 Time-criticality

We could consider time-criticality for multiple aspects, yet as we focus on audio-activities we only mention the time-criticality of auditory feedback.

5.1.1.1 Response speed

With responsiveness, we aim at the time it takes for generating auditory feedback to performed actions by the user or the device. This is obviously related to the audio output latency. However, additionally audio output could be dependent on inter-device communication, also the communication speed could as well be a relevant factor on the response speed.

We give two examples on response speed. Playing an instrument would require a very low response speed, as direct auditory feedback is preferred. In contrast, a playback system as the proof of concept allows a high response speed.

In a web-application, the activity may not demand a low response speed performance. Namely, due to high audio output latency, and slow internet communication.

5.1.1.2 Scheduling accuracy

With scheduling accuracy we directly aim at both users and devices to accurately define when audio actions should occur. From the perspective of the device, it is only relevant to consider the accuracy - the drift - between several devices. However, if an action - that requires scheduling - depends on user-interaction, the accuracy of this user-to-device communication becomes relevant to.

In a web-application, due to the drifting audio-context timer, only activities which require a low scheduling accuracy could be provisioned.

5.1.2 Scalability

With scalability we aim at the amount of devices which attend to an activity. It are three separate properties that define how congested the communication channels will become, and thus define the performance requirements in order to scale.

5.1.2.1 Inter-device communication

The amount of inter-device communication, is the amount of devices which have to directly, or indirectly, have to exchange information with one another. As the group size grows, the inter-device communication doesn't necessarily have to. For this, consider for instance an application that functions on communication clusters, or an application in which each device communicates with exactly two other devices, to create something like a linked-list of the participating nodes to forming a chain of communication. In a web-application, as in the proof of concept, all communication flows centrally through the web-server. This restricts on the inter-connectivity of devices. Additionally, it is

5.1.2.2 Information load

The information load describes the amount of information that is being communicated between two devices. Exchanging relevant information doesn't necessarily require a high load. For example, a device could interpret raw data itself and communicate only relevant, abstract information.

In a web-application, as in the proof of concept, the information load is quite restricted. Raw information transfers, for instance that of sensor data, can be extreme costly and congest the network instantly. Instead, an efficient communication strategy must be provided for the network to be able to handle it.

5.1.2.3 Group size

The group size describes the amount of devices which can, or have to, participate in the activity.

In a web-application, as in the proof of concept, the group size isn't a restrictive requirement. The centralized solution with a web-server, only suffers from network congestion and server load, which can be compensated by a decreased information load, as well with packing separate communication packages together under a high inter-device connectivity. In contrast, a distributed communication approach, such as Bluetooth, the maximum group size is heavily limited due to the low amount of concurrent connections a device can have.

5.1.3 Resource intensity

The device itself has several resources available. There will be computations and data involved, thus we can talk about computative and memory load.

5.1.3.1 Computative load

If we consider the computative load, we cannot directly address the computative complexity. Namely, if we consider a web-application, algorithms could be optimized quite well with the aid of for instance Web-Assembly, multi-threading, GPU computing. However, older browsers may not support certain of these optimizing functionalities, and therefore may have a stronger boundary of the algorithmic complexity.

Additionally, besides the performance of the software, additionally there can be changes in the hardware. Older phones have a slower CPU, or less cores, than the newest models. Nor can we assume older devices to have a viable GPU for computative tasks to be outsourced to.

5.1.3.2 Memory load

If we consider the memory load, we can both address the memory usage of the application, as well the total memory size of downloadable assets. In a web-application, there is quite a strong limitation on the memory load. Not only may the internal memory be restricted, as background processes and the web-browser itself have a lot of memory allocated. Additionally, fetching data over mobile networks could be expensive, and fetching over bad internet connections could result into the fetching to take a very long time, therefore the sizes of assets should remain small.

5.1.4 Influence of the the properties

The properties within the categories we have defined above can influence one another.

- Response speed and scheduling accuracy rely on each others, as intuitively, users will require a low response speed to accurately schedule their actions.
- Inter-device communication relates to group size. For instance, high inter-device communication could demand exponentially more bandwidth per additional participant, and thereby limits the maximally supported group size.

Additionally, we could consider the influence of performance requirements between the different categories.

- Relation of time-criticality with computative load: If an application requires a low response speed, and in this time audio responses have to be computed as well, the amount of computations for such an application have to be limited or the response speed will increase.

5.2 Categorizing audio activities

We could question how to consider the activity: maybe it can be sub-categorized into it being a game, or an artwork. If we consider it to be a game, we could extract insights from the gaming industry, which already has a lifetime of knowledge creating digital experiences. The industry tends to first directly categorize based on it's most prevalent gaming-type, such as puzzle, action, adventure, etcetera. Afterwards, tag-based filtering/selection further filters down on criteria, which could still be quite general and standard, which contains relevant attributes such as the properties of in-game atmosphere, community, game-play, or any other type of custom yet relevant selection criteria.

If we consider it to be an art form, research in the area of interactive artwork [38] gives us a list of taxonomies of interactive systems, levels of interactions, and taxonomies of interactive art. Additionally, more on the interactivity of the activity could be described by considering the underlying model which is being applied [39].

Whichever approach we use, we will need to be able to relate the categorization/separation of activities with it's performance requirements: that is the only way we could tell whether the activity could be performed with the aid of a web-application. The descriptions and taxonomies are very useful for describing activities, however unfortunately they won't offer a quantitative specification on the performance requirements. It may imply certain functionality or requirements to be more relevant, e.g. a co-authoring interactive application most probable requires a lower response speed than a passive interactive application, however this doesn't provide a concrete value of the technical requirement.

5.3 Benchmarking audio activities for fitness based on it's performance requirements

I don't see fit how any type of approach to categorization could provide a structural link to the performance requirements of the application. For example, how could an abstract description of an activity to be a puzzle game, guarantee anything about it's performance requirement on time-criticality? Even if we consider it from a more technical perspective, such as the type of interaction model, it doesn't provide much information on the scalability or computative requirements. Even in consideration of the time-critically, which can be more directly related to the interaction models, it cannot say exactly how accurate an application must be.

Therefore, instead of approaching the challenge by attempting to categorize audio activities and dividing these categories on being implementable by a web-application. Instead, we approach the challenge by setting up a benchmark, in such that any separate use-case can be checked/tested against this benchmark whether a web-application could be provision it's technical solution. The benchmark consists of the seven points on performance requirements outlined in 5.1.

There may be a variety of other types of performance requirements, which we haven't covered here. Such as, the need for localization, and whether this localization has to be granular or fine. Or, recording requirements in it's sensitivity and/or audio-analyzing capabilities to, for instance, recognize or even understand speech, tones, instruments, etcetera. Also, even if a certain activity

it's requirements are met by the web-application as we have defined above, the activity may still not to be provisionable by a web-application.

5.4 The fundamental qualities of microphone and speaker usage in audio-related activities

Both Bluetooth and WiFi function by communicating sound signals. A Bluetooth transmitter and receiver could be used for location estimation in the same manner as the speaker and microphone. However, microphone and speaker usage offers us two fundamental qualities that both Bluetooth and WiFi cannot offer.

The first fundamental quality, is the ability for the mobile device to sense the same audio signals we are able to hear. It enables the device to directly extract knowledge from the audio source without requiring an interface that translates intentions. A prominent use-case where this fundamental quality becomes clear for human to device interaction, is speech recognition. Not only we are able to hear what a person is saying, so is the computer. It shows this receival of audio microphone and analysis of audio data is capable to understand complex behavior.

A typical example of signal perception from the the device would be audio queues. Phone ringing, alarm clocks, notifications. Which brings us to another fundamental quality of the microphone and speaker: the mobile device is capable to create audio signals in the same frequency domain as humans are. Signal perception with signal creation allows us to create a bi-directional communication channel, from user to device, from device to user, device to device, and user to user. As mentioned in [38] it allows for a dialogue between human and computer.

All communication is hearable, it can create transparency and visibility on what is going on, and thereby allow users and devices to anticipate on what other agents may, or may not, be doing in their surrounding. This can strengthen autonomous, decentralized behavior. And in addition, considering collaborative efforts, a dialogue could target more than just two agents and thereby introduce more complex behavior.

6 Discussion

6.1 Background

In the background in 2.2 we mentioned how audio feedback could be provided. Additionally, the web-browser specifications support requesting it as mentioned in 2.2.3.3. Yet, as the changes occurring with the switch to audioWorkletNodes, it shows considering the web as an audio application target is not mature.

The calibration efforts are all a workaround to circumvent the missing functionality of audio output latency feedback, which was supposed to be implement by the `AudioContext.outputLatency`. Then it would be able to perform like software as Shairport Sync. If both the web-audio API and it's implementations would be more complete the need for calibration would become unnecessary for a few audio-activities, such as the default activity of sound-reinforcement.

6.2 Calibration

We mention in 3.1.2.1 a cross-correlation algorithm, however this implementation isn't performant. A far more performant algorithm transforms the signals into the frequency domain, in which cross-correlation is applied more efficient. However, it requires to consider windowing, overlapping, circular buffering, and the conversion itself, for which each could influence the cross-correlation result. Even more improvements could be made by writing this computative-intensive part of the application in web assembly. Implementing such improvements would cost time which wasn't available.

Due the long computation time of the algorithm we stick with, it influenced the design of the proof of concept. Namely, if cross-correlating would be so cheap to apply that it could conveniently and continuously analyze incoming audio streams, or even multiple audio streams, there wouldn't be need to write this much logic about ensuring simultaneous measuring. There wouldn't even be a need to initiate this process, the web-application would simply detect signals as it runs.

Additionally, we made a choice to stick with 4096 sample sized target signals due performance speed. Yet, if the algorithm would be optimized, we might be able to increase this number significantly. This therefore might as well greatly decrease the SNR on which the signal detection can perform, and thus drastically improve the signal detection range.

We have done some testing on the behavior on signal detection in . The virtual tests we perform for the signal detection algorithm, tests against background noise consisting of Gaussian noise. However, Brownian noise would be more natural to test against, as higher frequencies are attenuated more quickly in audio transmission. Due to the high spectral energy density of Gaussian noise, it does remain a good comparison to function as an upper boundary on the SNR accuracy.

Additionally, the target signal has been Gaussian noise as well. Interesting would be, what the behavior would be for signal detection accuracy if the target signal was a song. In particular it is interesting, how well it could be able to pinpoint at what time of the song is being played back. If this is possible, it would enable devices to calibrate without having to emit noise at all.

Input and output hardware sample rate conversion is serious challenge that requires research, in order to figure out how to detect conversion takes place, and how to alter the target signal at transmitter and receiver to be able to be detectable in case such conversion does take place.

The calibration performance model neglects variability in measure results and measure errors. This variability and errors however may be non-trivial, and therefore incorrect measurements have to be flagged, and over time measures should be repeated to ensure drifted latencies are corrected.

We haven't touched upon the challenge on how to realize signal emission and detection in the calibration process when there are a lot of devices. One challenge here, is to decide in what order devices emit their signal, in such they won't interfere with others. Another challenge is, how this could be optimized in contrast to the proof of concept, which takes approximately 2 to 3 seconds per measure. Particularly interesting with such long measure time, is the eligibility to perform measurements concurrently, as this could heavily reduce the total calibration duration.

6.3 Proof of concept

The proof of concept is functional, yet very minimal. In particular, the incomplete support of the Web Audio API for up-to-date browsers has been annoying during testing and experimenting. Unit-testing could be done in a large degree, however integration-tests are problematic as it would require emulation of physical space and the full audio path, including audio path latency. As we mentioned already, there was incomplete support for different web-browsers. This made experimenting challenging, and additionally it reduced the total amount of experiments.

The scope of this project is on audio synchronization within a web-application, therefore we embrace all the limitations and restrictions this platform provides to us. The results are promising, only the audio-context time-drift seems to be a barrier to succeed in synchronized audio playback. Additionally, hardware sample rate conversion may be a serious challenge, however we managed to set-up a test case in which this did not occur. Resolving this however, would require both detection and resolution.

The most problematic in consideration of the signal detection is the hidden layer of up-/down-sampling performed under the hood. However, more functionality could simplify development extensively.

- There isn't an API function provided to retrieve frequency information *without* converting to dB - thus keeping the complex audio values.
- Nor functionality to adjust FFT parameter settings, such as overlapping percentage or windowing function.
- Nor a convenient audio analyze part. There is a recorder API, which stores recording directly as '.ogg'. This may be convenient for straightforward recording and playback, yet for more advanced functionality that requires sample-precise and/or direct audio sample analyzing requires some juggling with inter-process communication - audio processor port communication with audio worklet node.
- As we already mentioned, Web API specifications aren't implement in every browser. The `outputLatency` parameter is support by only a single web-browser, and it doesn't even guarantee or suggest an upper boundary on the precision.

- Older web browsers only support AudioScript-nodes, while newer ones expect AudioWorklet-nodes, yet there are polyfills available to support both variants with a single code implementation.
-

Platform-specific software solutions could provide substantially more freedom in deciding on the implementation approach, as well provide far better accuracy. Therefore, native applications are feasible to consider as a solution for mobile phones. And the drawback in a reduced plug-and-play, for which the reduction is quite small anyway, could thereby be worth taking if there are large performance gains. Playback participation by listening to the song and playing along would be an interesting use case we haven't explored. Arguably the signal detection becomes harder to do as music is both repetitive and using natural signals which will make it harder to signal detect due the decreased target signal uniqueness.

We have neglect security concerns. Which in this case seem to not be present for the proof of concept besides a denial of service by flooding noise signal to prevent calibration. However, if it is used for security-sensitive tasks, e.g. identification, important commands, etcetera, security will become a more important aspect to consider in the development process.

6.4 Activities

The performance requirements that a web-application can adhere to require further description of it's concrete values. We could consider the requirements that can be met for different hardware. For instance, the web-application would be more scalable if all participating devices communicate on a 5G network rather than on a 3G network. Likewise, older phones and older web-browsers can only support a reduced computative load. A typical approach, within the domain of web-technologies, is considering concrete requirement values which target different percentages of devices. Also, if we require a very high time-criticality, we would target e.g. 40 percent of consumers that carry a high-end device.

7 Future research

7.1 Providing a solution under the current web-browser landscape

Thee proof of concept we have build is susceptible to all the shortcomings in performance and available functionality of the web-browser running on the mobile phone. Yet, with more research it may be able to resolve some problems to create a usable solution that can already be used in audio applications. Therefore the first field for future research is in consideration of the proof of concept itself.

7.1.1 Fixing the proof of concept

The first thing to do is solving the audio context time drift by stabilizing it. This is the only barrier left to obtain proper calibration of the proof of concept under our test environment.

7.1.2 Improving stability of the proof of concept

Even though the proof of concept works, it lacks a lot of support, both for modern web-browsers and older web-browsers. The error handling isn't complete, which makes debugging much harder. Furthermore, partly due to the dynamic nature of the web-application, there isn't any testing performed, which causes the debug process to take a long time and to be cumbersome.

Additionally, there is the sample rate conversion challenge we have talked about in 3.1.8.2. And we have neglect the presence of time drifting over time, which might be cause performance issues over time.

There isn't any differentiation between target signals, which may become problematic both from a scalability perspective - it hinders concurrent signal detection. As well from a security perspective, it is easy to interfere with the calibration process. This requires more research to find a proper technique.

7.1.3 Providing modular integration to other software

Ideally, the concepts and techniques developed could be applied and integrated to other platforms, or allow them to extend support for this calibration technique. If it becomes an available module to integrate in other web-applications, more functionality could be build that relies on calibrated audio output.

Additionally, if better specifications are available, it as well could be integrated in native applications, or to embedded devices, which may improve further on the possibilities with spatially distributed audio activities. This would for instance allow embedded devices to participate, without being required to own the full software stack to run a web-browser.

7.2 Considering web-browser technology

The AudioContext `outputLatency` variable is currently defined by the Web Audio API standard as a static value, without any additional information or guarantee on it's accuracy. Besides, currently it is not widely implemented. A step worthwhile to research into is whether more implementations for this function can be build, how accuracy could be guaranteed with and without Bluetooth output, and how the Web Audio API standard could be adjusted in order to provide better support and assure performance.

8 Conclusion

Multiple hardware and software solutions have been engineered to tackle the challenge of spatially distributed audio playback. However, the solutions provided for mobile devices is very limiting, and doesn't guarantee automatic calibration once audio output latencies are large. Which is inconvenient, since both hardware such as Bluetooth-speakers, and software such as Android, introduce non-trivial audio output latency.

This thesis has explored how a spatially distributed audio synchronization could be engineered for the mobile phone, and addition implemented it in a web-application. A proof of concept is build, which succeeds in performing automatic audio output calibration of two devices, and significantly improves the playback in comparison to uncalibrated playback. However, the proof of concept did not succeed to stay within an audio playback time offset less than 10 milliseconds. This inaccuracy can be addressed to the variable latency offset between the system time and audio context time, what requires additional research to understand on how this variability can occur, and how to resolve it.

The integrated microphone and speaker of the mobile phone provide fundamental qualities that show potential to provision interactive, collaborative, co-located, spatially distributed audio-related activities. However, web-applications, mobile phones, and Bluetooth integration, all lack certain functionality or performance specifications, to create low-latency, yet highly accurate, audio-related interactivity. And therefore further development of functionality is required to obtain stable performance for the vast amount of web-browsers and available audio consumer hardware.

References

[1] Best practices in network audio. Technical report, 2009.

[2] Cisco - smb university - networking fundamentals. https://www.cisco.com/c/dam/global/fi_fi/assets/docs/SMB_University_120307_Networking_Fundamentals.pdf.

[3] Ipcisco - lessons - osi referance model. https://ipcisco.com/lesson/osi-referance-model/.

[4] Wikipedia - osi model presentation layer. https://en.wikipedia.org/wiki/Presentation_layer.

[5] The linux information project - presentation layer definition. http://www.linfo.org/presentation_layer.html.

[6] Ietf - rfc 3439 - some internet architectural guidelines and philosophy. https://tools.ietf.org/html/rfc3439#section-3.

[7] K. Gross. Q-lan - the architecture and network redundancy. Technical report, 2009.

[8] Volumio. https://volumio.github.io/docs/FAQs/General.html.

[9] Shairport-sync. https://github.com/mikebrady/shairport-sync.

[10] Ampme - home page. https://www.ampme.com/.

[11] Soundseeder - home page. https://soundseeder.com/.

[12] Soundseeder - frequently asked questions. https://soundseeder.com/.

[13] Ampme - faq. https://www.ampme.com/faq?locale=en_US.

[14] Soundseeder - help, syncing the speaker playback. https://soundseeder.com/help/sync-playback/.

[15] Ampme - google play store review sync problem. https://play.google.com/store/apps/details?id=com.amp.android.

[16] Ampme - google play store review sync problem, second. https://play.google.com/store/apps/details?id=com.amp.android&reviewId=gp%3AAOqpTOGlSNbjXCSmfPH5hQsKapmPrCKyYrgtJyKWLYn3-DXGjy9Awh4uXWGq93AeWmJQE9bgBTlaJ7rYneg2w

[17] Ampme - google play store review sync problem, third. https://play.google.com/store/apps/details?id=com.amp.android&reviewId=gp%3AAOqpTOGlSNbjXCSmfPH5hQsKapmPrCKyYrgtJyKWLYn3-DXGjy9Awh4uXWGq93AeWmJQE9bgBTlaJ7rYneg2w

[18] Superpowered - mobile phone audio round-trip measurement results. https://superpowered.com/latency.

[19] Soundguys - android bluetooth latency.
https://www.soundguys.com/android-bluetooth-latency-22732/.

[20] Superpowered - the android audio path latency explained.
https://superpowered.com/androidaudiopathlatency.

[21] Android developer - aaudio.
https://developer.android.com/ndk/guides/audio/aaudio/aaudio.

[22] Bluetooth - audio/video distribution transport protocol specification. Technical report, 2012.

[23] Android developers - stable ndk api's - audio.
https://developer.android.com/ndk/guides/stable_apis#audio.

[24] Android developers - ndk audio - aaudiostream - getframeswritten. https://developer.android.com/ndk/reference/group/audio#aaudiostream_getframeswritten.

[25] W3 web audio api standard. https://www.w3.org/TR/2018/CR-webaudio-20180918.

[26] Whatwg - media - current playback position.
https://html.spec.whatwg.org/multipage/media.html#current-playback-position.

[27] Mdn - audiocontext outputlatency.
https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/outputLatency.

[28] Android developers - audio round-trip measuring.
https://source.android.com/devices/audio/latency/measure#measuringOutput.

[29] Jun Liu, Hongbin Li, and Braham Himed. On the performance of the cross-correlation detector for passive radar applications. *Signal Processing*, 113:32 – 37, 2015.

[30] R. (Rene) Carmona. *Practical time-frequency analysis : Gabor and wavelet transforms with an implementation in S*. Wavelet analysis and its applications ; Volume 9. 1998.

[31] J Palmer, S Palumbo, A Summers, D Merrett, S Searle, and S Howard. An overview of an illuminator of opportunity passive radar research project and its signal processing research directions. *Digital Signal Processing*, 21(5):593–599, 2011.

[32] Takuya Yoshioka, Zhuo Chen, Dimitrios Dimitriadis, William Hinthorn, Xuedong Huang, Andreas Stolcke, and Michael Zeng. Meeting transcription using virtual microphone arrays. Technical Report MSR-TR-2019-11, Microsoft, July 2019. Revised version.

[33] M.N. Ayyaz M. Farooq-i Azam. Location and position estimation in wireless sensor networks. Master's thesis, 2017.

[34] Web performance api - performance.now().
https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.

[35] Github - audioworkletnode polyfill that uses a web worker.
https://github.com/jariseon/audioworklet-polyfill.

[36] Github - audioworkletnode polyfill that does not use a web worker. https://github.com/GoogleChromeLabs/audioworklet-polyfill.

[37] Rfc 5905 - ntp specfications. https://www.ietf.org/rfc/rfc5905.txt.

[38] Hanna K. Schraffenberger and Edwin van der Heide. Audience-artwork interaction. *International Journal of Arts and Technology*, 8:91 – 114, 2015.

[39] Danyi Liu and Edwin van der Heide. Interaction models for real-time participatory musical performance usingmobile devices. *International Computer Music Association*, 2017:305 – 310, 2017.

[99]