Mobile Systems

Synchronized Playback of Music

AALBORG UNIVERSITY
STUDENT REPORT

8th Semester Software
Project Report
SW806F17

Aalborg Universitet
Selma Lagerløfs Vej 300
DK-9220 Aalborg

AALBORG UNIVERSITY
STUDENT REPORT

8th Semester
Software
Selma Lagerløfs Vej 300
9220 Aalborg
http://www.cs.aau.dk

Abstract:

I am the abstract abstract.

Title:
Mobile Systems

Theme:
Mobile Systems

Project period:
8th Semester

Projectgroup:
SW806F17

Group members:
Jesper Jensen
Marc Tom Thorgersen
Mathias Sass Michno
Thomas Pilgaard Nielsen
Troels Bech Krøgh

Supervisor:
Stefan Schmid

Number of pages:
XX

Ended:
29th of May 2017

Preface

Preface

Aalborg University, Wednesday 12th April, 2017

Jesper Jensen
<jejens13@student.aau.dk>

Donald J. Trump
<baby@hands.aau.dk>

Marc Tom Thorgersen
<mthorg13@student.aau.dk>

Mathias Sass Michno
<mmichn13@student.aau.dk>

Thomas Pilgaard Nielsen
<tpni13@student.aau.dk>

Troels Beck Krøgh
<tkragh13@student.aau.dk>

Contents

Preface **iii**

I Introduction 1

1 Introduction **2**
1.1 Initial Problem Statement 2

2 Establishing Use Cases **3**
2.1 Category: Increase Volume and Area 3
2.2 Category: Isolated Playback 4

3 State of The Art **6**
3.1 Android Apps 6
3.2 Technologies 12
3.3 Conclusion 14

4 Testing the SotA **16**
4.1 Test Setup 16
4.2 Test Procedure 18
4.3 Test Results 19
4.4 Test Discussion 21
4.5 Test Conclusion 22

5 Problem statement **23**

II Requirements & Design 24

6 Requirement Elicitation **25**
6.1 Milestones 27

7 Architecture **29**
7.1 External Architecture 29
7.2 Internal Architecture 31

III Implementation **34**

8 Technologies **35**
8.1 The Android Sound Stack 35

Bibliography **39**

IV Appendices **43**

Part I

Introduction

1 Introduction

In today's world it is common to find yourself in a situation where you desire to listen to music in a social setting. Most smartphones have speakers of low quality, to reduce cost and save space, and are unable to play at a volume sufficient for a party or social gathering, without significant distortion. This has given rise to a large amount of small battery driven loudspeakers, which you can connect to, either wired or wireless. However, using these creates a number of new problems, such as: cost, remembering the device, charging the device, remembering the cable, etc.

1.1 Initial Problem Statement

The problem described in the introduction, and the one that will be used in this project as a starting point can be summarized to the following:

> How can we make an application that seamlessly synchronizes audio playback wirelessly between multiple devices?

[1]

In order to minimize the ambiguity of the initial problem statement, we define and explain the terms we use, in the context of this paper as follows:

Seamlessly synchronizes
: From the initial idea generation phase we suspect that the delay between devices, of the audio playback will play a significant role in the design of the solution. Therefore we want to synchronize the audio to a precision that makes it feel seamless to the human ear.

Wirelessly
: Part of the application being seamless is also that external wires etc. are not required for the devices to function together.

Mutliple devices
: We envision that a solution would involve multiple devices to seemingly amplify the signal without the distortion of trying to push sound out of small speakers. The devices need not be the same exact hardware.

To get a further understanding of the problem at hand, and to construct a final problem statement, we explore use cases for such an application. Then analyze some existing related solutions on the market, and test some of them in depth. We do this to understand the problem, and use that knowledge to make a final problem statments, and a list of requirements for the project.

[1] Det er devices vi vil sync of ikke playback, er det for tidligt at lave den distinction uden info omkring hvorfor devices > playback? – *Marc*

2 | Establishing Use Cases

In order to determine the requirements for the system we will develop, we identify a set of use cases, for which synchronized distributed playback of audio could be relevant. We use brainstorming among the group members to initially recognize use cases, and then select the most interesting. This brainstorming is influenced by the use cases which the apps in Section 3.1 *Android Apps* conforms to. The found use cases are then categorized, such that scenarios which may share some fundamental ideas or requirements are grouped together.

In each category we first describe the common traits, then give concrete examples of use cases. The goal of establishing the use cases is not to choose a single use case or category to develop a system for, but utilise the potential requirements of each use case categories in the requirements elicitation.

2.1 Category: Increase Volume and Area

A shared trait for use cases in this category is the use of synchronization to increase the volume of the audio, or the area in which it can be heard. All use cases in this category rely on the synchronization being done well enough such that no difference in audio playback is perceived.

Examples of use cases in this category are:

Social Gathering

At a social gathering or party, dependent on the event, the participants wants to be able to listen to loud music. If no dedicated sound system is available, phones can be used. Phones are generally only capable of playing music at low volume and if some participants are talking and dancing, it can be hard to hear what is being played.

Given the ability to play music synchronized across all their phones, the participants can use their phones and play the same music as one device. This will make the perceived volume of the music louder to the listeners and the overall experience of listening to the music better. If some at the party or gathering just want to talk, they can either turn down the volume of their devices, mute or turn the music off on their own phone, making it quieter for them while the other participants still can hear the music. The participants could also be able to requests songs to the playlist.

For all the participants to have the impression of only one speaker is playing, the synchronization has to be precise enough so that there is no perceived shift in time in playback of the music between the devices.

Festival Scenario

At a music festival, there is often an area for the concerts and an area for people to stay before and after the concerts, often called a camp. These camps can stretch over a large area, dependent on the size of the festival. It is difficult to play music over the whole camp

and it is possible that the festival guests wants to hear it at different volumes, to allow conversation etc.

Many, if not all, participants have phones which can be used to play music. By having these phones connect to a device, which can propagate music, the festival participants can play the music in a synchronized manner, effectively increasing the area covered by the music. As with the social gathering use case, the festival participants should be able to control the volume locally without affecting the rest of the participating devices. Furthermore everybody should be able to request songs, such that everybody can influence the choice of music.

Multi–room Setup

When at home, people often listen to music. What if the music playback would be synchronous across all rooms of the home? Then there should not be a difference in synchronization between each room, it should be an experience of walking from room to room and listen to the exact same song.

By having a system which can synchronize music, devices in each room can be connected to the sound system and play the music synchronized. This renders the area of the music larger and gives the experience of one speaker or sound system playing. The connected devices should be able to individually adjust the volume, mute or turn off the music, without affecting other connected devices.

This category is the one which relates the closest to the solutions covered in Chapter 3. **Multi–room Setup** specifically is exactly what Sonos does to a very satisfactory degree. This being a more static scenario, as rooms rarely change, speakers make sense in this scenario making a more mobile implementation, e.g. smartphones, less needed. As for the other two use cases, these cases are seemingly what both AmpMe and SoundSeeker are designed for, however as revealed through our tests in Section 4.5, these solutions are not satisfactory.

2.2 Category: Isolated Playback

Another category of use cases can be classified as "providing isolated yet synchronized playback between multiple devices". This category does not specify the limits for acceptable synchronization in and of itself, but it does require that the individual devices are synchronized without being able to listen to each other.

Two examples of use cases, which fit this category are:

Silent Clubbing

While one might argue that an essential element of going to a club or disco is experiencing the loud music, try imagining a silent disco — A club where the deafening beat is replaced by nothing but the sound of people shuffling their feet across the dance floor and silence. It would be a significantly different experience than you would normally have at a club, and it would require guests to posses a compatible device and a pair of headphones. These compatible devices could for example be Android smartphones, and the club could provide this equipment along with headphones at the entrance.

The guests at the club, would then be able to take part in "clubbing" by using their synchronized device and a pair of headphones. Because all guests experience an isolated

music playback, the requirements for how precise the synchronization should be, are softer than in the previously described use cases. This means that the music any two guests hears does not have to be precisely synchronized, since they would never hear each others audio. However, the synchronization would still have to be precise enough for the guests dancing to seam somewhat in tune, i.e. guests should feel that they are listening to the same music as everyone else.

In this use case scenario, it would also be possible for the individual user to apply an equalizer to the music, and more importantly control their own volume.

Multilingual Movie Theater

Most popular movies nowadays are translated into a myriad of different languages, mostly by way of subtitles but also by dubbing the dialogue in a different language. This raises a new problem, when watching a translated movie in the theater. Surely only one translation can be used, thereby making it more difficult to follow along if one is not fluent in the spoken or subtitled language.

One solution could be allowing the audience in a movie theater to choose between multiple different audio tracks, and listen to it through a headset. These different audio tracks could be different translations, but also uncensored audio or a narrating audio track for blind persons.

In such a setup the synchronization would need to be precise, since the audio should match the picture. However, because screening a movie does not include "live" or unpredictable audio, the individual audio streams could be sent well ahead of playing, thereby relieving the need for fast and stable network connection. Moreover, to aid in the synchronization the individual devices could utilize the main audio being played from the movie theater's speakers to line up with.

These examples have two significant things in common. Firstly, both of these scenarios require the involvement of the business or organization in charge of the club or movie theater. This means that the data to be transmitted to the guests' or audience's devices, could be sent from something like a dedicated computer, i.e. a device without the strictly limited memory and computational power that mobile devices impose. Secondly, both examples require the users of the individual devices to have some kind of headphones, which is a requirement that should be considered, as it may restrict some people from using the systems. [1]

[1] Der skal nok være en konklusion her som står lidt for sig – *Troels*

3 State of The Art

3.1 Android Apps

In this section we investigate the state of the art for synchronized streaming of music between different wirelessly connected mobile devices. We found three apps capable this:

- SoundSeeder
- AmpMe
- Chorus

These apps are found by searching sources like Google[1], YouTube[2], App Store[3], and Google Play Store[4] and from recommendations on forums. An additional criteria for an app to be considered here is that the app must not be abandoned by the developers, which we defined as being without an update since, at least, 2013, and being incompatible with new devices.

To clarify, Google Play Store and the App Store are the official places to install or buy apps for Android and iOS devices respectively.

The three apps all use a master/slave connection and use different terms for their setup. For clarification we generalise these terms. We call the device which selects the music and streams it, the master, and the devices which connects to the master (SoundSeeder calls it "Speaker"), are referred to as slaves.

3.1.1 SoundSeeder

SoundSeeder[5] is an app made by JekApps. The current version as of Thursday 9th February, 2017 is 1.6.5.

It is compatible with Android 4.1 and above, for older versions of Android (2.2 – 4.0), another app called SoundSeeder Speaker makes it possible for a device to be used as a slave.

SoundSeeder also provides a Java application for compatibility with other platforms e.g. Windows, macOS and Linux, but does not support iOS and Windows Phone[14].

The app consists of a top menu and a burger menu at the left side, as seen on Fig. 3.1a. In the burger menu, the user can choose the different playback possibilities and switch to slave mode. The main view of the app is a music player, where the music can be controlled as any other music player. To play music to other devices as a master, press the "add music button" in the top menu, choose the source of the music and choose the preferred music, and press play.

To connect to a playing master device, as a slave, select the speaker mode from the burger menu. If it finds the device, it connects automatically. It can take a bit of time for the slave

[1] `https://www.google.dk` with search terms "android sync music playback" [2] `https://www.youtube.com/` [3] `https://itunes.apple.com/dk` [4] `https://play.google.com/store` [5] `http://soundseeder.com`

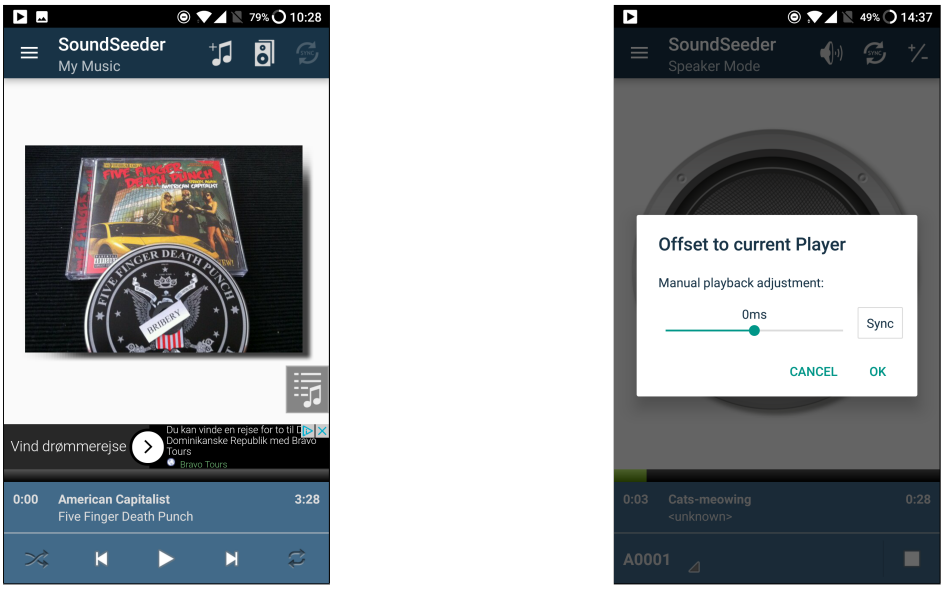
device to find the master device, which can seem confusing and make users think they need to connect manually with an IP address.

The process of joining, when the slave is given the time, is rather simple. In regard to the user interface, it is clotted with features and menus, which makes the app hard to navigate. The design of the app is old and not pretty.

SoundSeeder streams music via Wi-Fi, which means that all phones have to be on the same network[13][6], it is also possible to use an ad–hoc network (Android Hotspot). In regard to the master's music source, it can be Google Music, online radio stations, UPnP and DLNA devices, local media, and YouTube if using semperVidLinks[7], an app for extracting video links. SoundSeeder also supports streams from external sources, e.g. a microphone or AUX device. The supported media formats further depend on the used master device and its Android version.[13]

SoundSeeder synchronizes the audio playback when a slave connects, but it can also be done manually on the slave device. On Fig. 3.1b, the manual synchronization adjustment for SoundSeeder can be seen. This slider is used in the case that the music is not fully synchronized, and goes from $-400ms$ up to $+400ms$, in $10ms$ increments. Additionally there is an auto synchronization button in the top menu and on the slider menu window.

SoundSeeder is free to install but the free version only allows two slave devices to connect for up to 15 minutes at a time. The app license costs 39.90 DKK.



(a) When the app is opened.

(b) The synchronization slider.

Figure 3.1: Screenshots from SoundSeeder

[6] Der mangler måske noget her med NAT – *Jesper*

[7] `https://play.google.com/store/apps/details?id=com.semperpax.sempervidlinksFree`

3.1.2 AmpMe

The second app is AmpMe[8], made by Amp Me Inc. The current version of the app as of Friday 10th February, 2017 is 5.1.1. It supports Android 4.1 or newer and iOS 9.0 or newer.

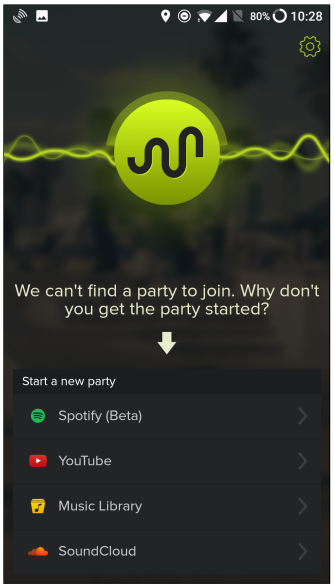
In AmpMe the group of devices is called "a party". When AmpMe is opened, it either displays the nearby parties or encourage you to create your own. If a party is found nearby, you can join it as a slave and play the master's music. If no party is found, or you want to create your own, you can choose to start it by choosing between Spotify, YouTube, your local music library, or SoundCloud as music source, as shown on Fig. 3.2a. When a source and music is chosen, a player appears with the music playing, and your device works as a master.

It is very intuitive to join a party, or host one yourself and play music. The interface is modern, minimalistic, and pleasant to use and look at.

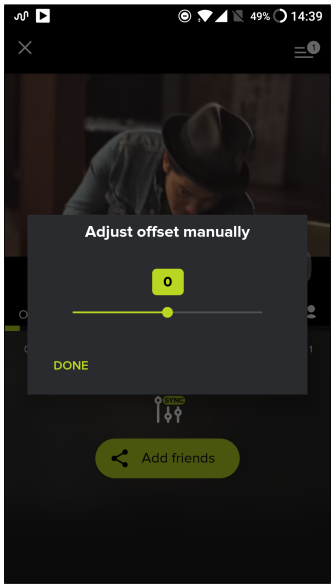
In order to stream music from a master to slaves, AmpMe requires an Internet connection. This means that the connected devices can be on WiFi, mobile data or another network, as long there is Internet access.

In AmpMe, the music is automatically synchronized upon party creation, but if there is an offset, it can be synchronized manually at each individual slave. On Fig. 3.2a, the slider to manually synchronize can be seen. The slider goes from an offset of -15 to $+15$ arbitrary offset units, in 1 increments.

AmpMe is free to use in both Google Play and the App Store.[12][11][10]



(a) When the app is opened.



(b) The synchronization slider.

Figure 3.2: Screenshots from AmpMe

[8] `http://www.ampme.com`

3.1.3 Chorus

The final app is called Chorus[9] and is made by AVR APPS. The current version is 2.1, as of Monday 13th February, 2017. It supports Android 4.0 or newer, AVR APPS mention an iOS app, but the App Store page returns that the app is not available in our region[10].

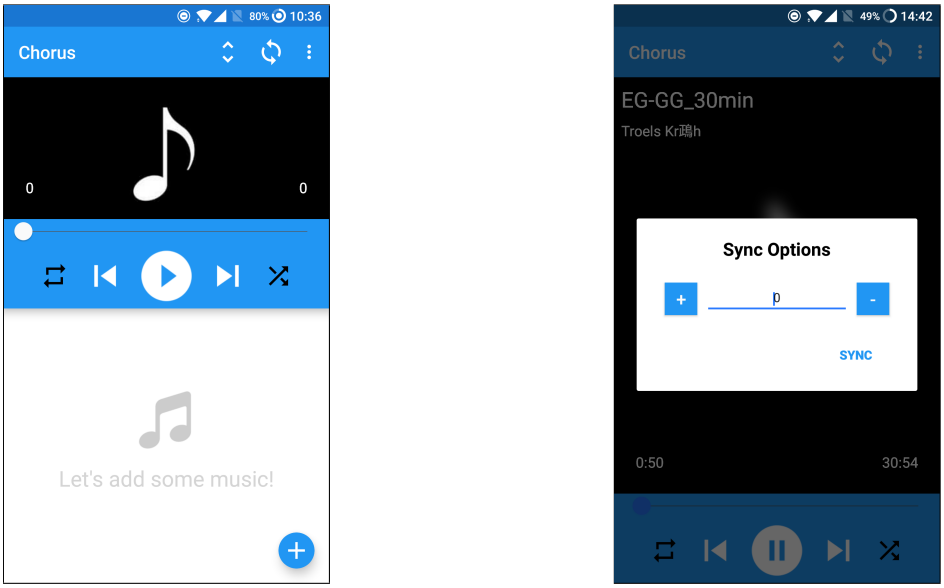
To share music between devices, all devices have to be on the same network. Which means WiFi or a mobile hotspot is required. Chorus only supports playback of local media on the master device.

When the app is opened, it looks like a regular music player, as seen on Fig. 3.3a. Music is added by pressing the small plus sign in the right bottom corner. Which Chorus then streams to connected slave devices. To connect to a master device playing music, you press the menu button in the right top corner, and press join. Here a menu pops up with devices playing on the network, and if you select the device shown, it connects as a slave and starts playing. We encountered a scenario where the app kept loading when attempting to join an active network, after a restart of the app it worked.

It is intuitive to use the app, it is very minimalistic hence easy to navigate. The design is very minimalistic, and seems to follow the new "Material Design"[11] guidelines set out by Google.

Chorus supports manual and automatic synchronization. On Fig. 3.3a the synchronization slider can be seen, it slides from $-400ms$ to $+400ms$, in steps of $10ms$.

The app is free to download and use without any ads.[5].



(a) When the app is opened.

(b) The synchronization slider.

Figure 3.3: Screenshots from Chorus

[9] https://play.google.com/store/apps/details?id=com.avrapps.chorus [10] https://itunes.apple.com/app/chorus/id894014439[redacted]
[11] https://material.io/guidelines/

3.1.4 App Comparison

To determine what makes these apps state of the art, we look at their vital characteristics. These vital characteristics are the user rating, last update date, supported devices, media sources, and connectivity options. We look at these characteristics since we find them important to mobile apps and can use these characteristics to evaluate the apps. There are also other characteristics of the apps, but we deem these are the most important ones. In Table 3.1, a comparison of the apps can be seen.

To start with Chorus, it does not support other devices than Android, narrowing down the supported devices compared to SoundSeeder and AmpMe. Furthermore it only supports local media as music source, requiring all media to be played to be on the device. Chorus also has a lower rating than SoundSeeder and AmpMe, which indicates dissatisfied users, and the last update happened on 11/5 2015 indicating that it is abandoned. On the other hand Chorus streams via WiFi and Android hotspot, just as SoundSeeder, and it has automatic and manual synchronization. Since Chorus only supports Android, playback of local media and do not seem to further developed, we do not deem it as state of the art.

In regard to SoundSeeder and AmpMe, they have their app ratings, Android support, YouTube and local media support in common. AmpMe supports iOS devices, but SoundSeeder has a Java version which means that it supports all devices running Java, which is an advantage. Therefore to be a part of the state of the art, it is important to support different types of devices.

In regard to supported devices, SoundSeeder supports more devices than AmpMe, but AmpMe supports Spotify. Spotify has around 100 million users, which makes it an important source to support[27]. Google Music have not released any official user numbers, but it is expected to be lower than Spotify[8]. On the other hand SoundSeeder supports external sources, so a source supporting Spotify could be used with SoundSeeder that way. This means that it is important to support popular and widely used music sources to be considered state of the art.

SoundSeeder supports streaming via WiFi or Android hotspot. This means that all devices have to be on the same network and have connection to each other, which can be an issue in larger network configurations, where clients can be restricted from communicating with each other, which is the case at Aalborg University. AmpMe does not have this restriction since it uses an Internet connection, this does have the restriction of using mobile data. Furthermore if slow mobile data is used it can cause problems with the synchronization, but so can Internet via WiFi. As both solutions have their respective restrictions we deem that both using WiFi/hotspot and Internet is a part of the state of the art.

Both SoundSeeder and AmpMe synchronizes automatically upon created connection. In the case that the synchronization becomes skewed, both apps have the possibility of manually synchronize and to set a synchronization offset. Since there is a need for manual synchronization options, this could mean that the automatic synchronization can at times be faulty, but it is a good alternative to either disconnect and reconnect or wait for the automatic synchronization synchronize by itself. Therefore to be a part of the state of the art, an app have to be able to both automatically and manually synchronize. The actual performance of the synchronization, in SoundSeeder and AmpMe will be tested in a later section.

	Rating (Out of 5)	Latest update	Supported devices	Media source	Connectivity	Pricing
SoundSeeder	3.9 Play Store	11/11 2016	Android 4.1+ Java Devices	Google Music YouTube external devices UPnP DLNA Online radio Local media	WiFi Android Hotspot	Free (limited) 39.90 DKK
AmpMe	4.2 Play Store 4.0 App Store	30/01 2017	Android 4.1+ iOS 9.0+	Spotify SoundCloud YouTube Local media	Internet	Free
Chorus	3.3 Play Store	11/05 2017	Android 4.0+	Local media	WiFi Android Hotspot	Free

Table 3.1: Comparison between the apps as of the 13th of February 2017.

3.2 Technologies

In this section we take a closer look at streaming and split this into two primary types of streaming: video and audio. As we are focused on audio streaming, services and solutions in this field are the most important ones, however video streaming services are also mentioned in order to compare the two. Furthermore we separate audio streaming and multi–room audio systems, i.e. synchronized audio streaming using multiple devices, as this is a special case of audio streaming and also happens to be closely related to our problem.

3.2.1 Audio Streaming

Streaming music is a common way of listening to music, with services such as Spotify[12] making a large selection of music available through a subscription service. A key part in streaming audio is the streaming protocol, to this end Spotify previously used peer–to–peer networking in order to handle the demand.[15] However, demand is not a concern for us, the streaming protocol used is. As for streaming protocol, Spotify uses their own proprietary protocol, as is represented on Fig. 3.4.[20][13] Since their protocol is proprietary, only a small amount of information about it is available to the public.

Another service which provides audio streaming, is AirPlay[14], made by Apple. AirPlay is not a provider of music like Spotify is, but a service used for wireless streaming of audio between Apple devices. AirPlay also use their own proprietary protocol, Remote Audio Output Protocol, which is based on Real Time Streaming Protocol (RTSP). In fact the general impression received is that audio streaming services use proprietary protocols.

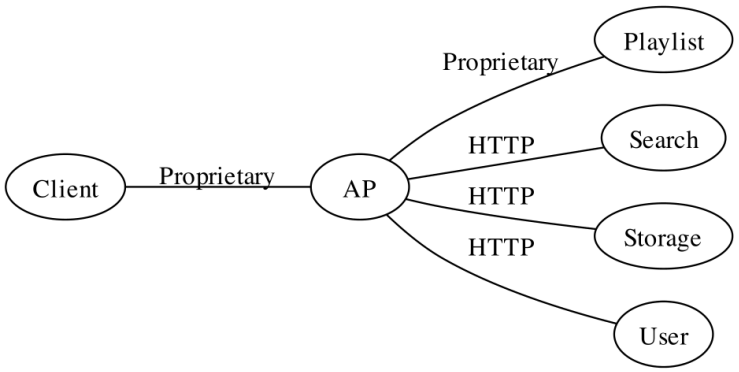


Figure 3.4: Overview of Spotify from 2011 showing the protocols used for communication within the app.[20]

3.2.2 Video Streaming

For streaming video there are a variety of streaming services out there, these are generally of two types, live and pre–recorded. Netflix[15] allows for streaming pre–recorded content, e.g. movies

[12] https://www.spotify.com/
[13] Jeg synes stadig at figuren skal beskrives bedre, ved dog ikke om den overhovedet er relevant. – *Thomas*
[14] https://support.apple.com/da-dk/HT204289 [15] `https://www.netflix.com/`

and tv–shows. Other services such as Twitch[16] and YouTube[17] allow for both livestreaming, and streaming of pre–recorded content. Livestreaming services such as Twitch, which streams from a "streamer" via their servers to the clients, with a baseline short delay.

In contrast to the audio streaming services the video streaming services do not seem to be relying on proprietary streaming protocols. Netflix uses Dynamic Adaptive Streaming over HTTP (DASH), Twitch uses HTTP Live Streaming (HLS) and previously used Real–Time Messaging Protocol (RTMP) which they still support, and YouTube uses RTSP.[24][28][25] Neither of these three streaming services use the same protocol, and all three are in the top of the line. This gives the impression that there must be a considerable difference between audio and video streaming requirements when it comes to the streaming protocol.

3.2.3 Multi–Room Audio Systems

Beyond streaming protocols there is another technology which closely relates to our problem, namely Multi–Room Audio. Multi–Room Audio is fairly self–descriptive, it provides audio to several rooms, this audio however is synchronized. While there are several ways of making a Multi–Room Audio solution for a home, predominately the technology is developed and provided by speaker manufacturers. Several manufacturers provide this service to a degree but one manufacturer in particular is leading the technology and build their entire brand around it, Sonos[18].

It is Sonos' proprietary network software alongside their hardware that have kept them at the top of the market. Sonos' own mesh network was developed due to WiFi not being sufficient at the time when Sonos started development, however that is no longer the case as can be seen my both their competitors such as Bose[19], and their own technology which now also supports standard WiFi, although their own network is still more reliable.[16] The network they use is a proprietary peer–to–peer mesh network, aptly named SonosNet.[26]

Sonos' speakers are wireless and stream the music directly from the Internet, as such they can run even if the device used to start them is turned off. To this end Sonos must support a range of services in order to be useful, and as a well–known brand they support a significant number of services. Their speakers can also play different songs in each room, or play the same songs in sync. SonosNet is also compatible with existing speakers and sound systems one may have through another Sonos device, the versatility and flexibility provided by Sonos alongside their technology being ahead, is what makes them top–of–the–line when it comes to Multi–Room Audio.[23] Multi–Room Audio is a competitive area and as such information on technologies and software used beyond knowing it is proprietary is scarce, as companies want to protect their trade secrets.

Multi–Room Audio is not exclusively a technology which is bought, hobbyists have created their own home Multi–Room Audio systems using open source projects such as PulseAudio.[17] PulseAudio is a sound–server program accepting sound from one or multiple sources and then redirecting it to one or more of sinks; one of their supported protocols is Real–Time Transport Protocol (RTP).[18]

[16] `https://www.twitch.tv/` [17] `https://www.youtube.com/` [18] `http://www.sonos.com/`
[19] `https://www.bose.com`

3.3 Conclusion

From the information gathered in this chapter we derive conclusions to help us formulate the problem and elicit requirements in order to solve said problem.

The apps described in Section 3.1 provide insight into what such an app should be capable of. For each feature we can consider the reasoning behind adding that feature and from there determine possible problems we will need to address in our app.

In our analysis we determined that Chorus did not meet the criteria to be state of the art having low ratings, occasionally stopped working, and poor support for devices and music sources. As such we do not gain much information from Chorus aside from knowing that for an app to be useful, it needs to support more than only local audio files.

SoundSeeder and AmpMe both qualified as state of the art, this prompts us to consider their features, why they exist and why we might want to include some in our app. First off is the audio playback support, both SoundSeeder and AmpMe support various media sources, most important of all, audio streaming sources. The two apps use their own respective solutions for device connectivity, SoundSeeder uses WiFi/ad–hoc Android hotspots where AmpMe uses any Internet connection. This implies that either solution is capable of solving the problem, tests in Chapter 4 evaluate just how good these solutions are and determines whether one solution performs better than the other. Both apps also support the feature of manual synchronization, the need for such a feature implies that to some extend the automatic synchronization is not enough. It can be quite difficult to assert exactly how desynchronized two devices are. SoundSeeder allows the user to change at intervals of $10ms$ between $-400ms$ and $+400ms$ whereas AmpMe uses an arbitrary value from -15 to $+15$.

Having an arbitrary value gives no impression of the degree to which a delay is added, and being able to hear that the devices are playing at an exact $70ms$ discrepancy is impossible to pinpoint simply by listening; as such the manual synchronization feature is largely a trial and error effort to reach synchronization. While we can not disregard that such a feature may be necessary to reach a synchronized state, it is worth considering alternate ways of allowing manual synchronization if it is needed, or perhaps a resynchronization feature.

From the audio streaming segment we can see that there is a tendency to use proprietary software solutions. This gives us the impression that there are certain requirements which the well known streaming protocols do not fully support.

We know specifically from AirPlay that their solution is based on RTSP, thus there are some elements to the protocol that work just fine but does not fulfill all of AirPlay's requirements. We want to extend beyond simple streaming of audio to use multi–room audio technology, i.e. synchronize across devices. In this field we know hobbyists have used programs such as PulseAudio which supports RTP.

Our insight into this field show that whilst Sonos uses their own network, SonosNet, WiFi and Bluetooth both work as well, as Sonos' competitors use these, and Sonos also supports standard WiFi. As for their streaming protocol, similarly to the audio streaming services we know no more than that it is proprietary. Our insight into video streaming may help shed some light on as to why proprietary software is used rather than the more common streaming protocols. In this field the companies use a variety of well established protocols rather than proprietary ones.

This lets us conclude that the difference in requirements for audio compared to video streaming is what is causing audio focused companies to use their own proprietary streaming protocols.

Some of these differences may include latency, file size, bit rate, and buffering requirements. As such going forward we must explore how these factors affect the streaming protocols in concern to audio to ascertain which protocol to use, and whether we should fiddle with the open protocols. Knowing that some audio solutions use RTP and AirPlay is based on RTSP, these two protocols in particular might be worth considering.

4 | Testing the SotA

In the state of the art we identified two apps, AmpMe and SoundSeeder, which are candidates for solving the initiating problem. In this chapter we want to test whether or not they solve it to a satisfying degree. That is to say we want to identify some parameters, and test to what degree the state of art achieves them.

We have identified the following three categories to test: *a*) Offset, *b*) Consistency, *c*) Drift.

These three categories are related but slightly different. Firstly offset, refers to whether or not there is an offset between different devices, e.g. is the playback of one device behind that of another. Secondly consistency refers to whether or not a potential offset is consistent as the playback is manipulated by something diffirent than time, i.e. is the offset consistent when the song is changed, or the playback is paused and then resumed. Thirdly drift refers to whether the offset changes during playback over time. Each of these are required for an app to fulfill the initial problem statement presented in Section 1.1 on page 2. Moreover if the apps fail, to a certain degree, in any of the categories, then they are not fully suited as a solution for the problem at hand. Concretely we want to answer the following questions in each category:

a) Offset

- *1*) Is the audio synchronized without any manual adjustment?
- *2*) Can the manual adjustment make them synchronized?
  - *2.1*) What does the slider do exactly?
  - *2.2*) How much does each step affect the offset?

b) Consistency

- *3*) Is the delay consistent between playbacks?

c) Drift

- *4*) Is the synchronization stable over time?

These question are, to some degree, independent and an application can fulfill some or all of them.

4.1 Test Setup

In order to perform tests for these criteria, we need to have a setup to do so.

Testing the offset between two audio sources is a signal processing task. To make the signal processing easy and effective, a clear signal is needed. Additionally the signal needs to be synchronized between the channels at recording time, since otherwise the analysis would be

mislead by desync in the recording. Luckily most computers include a port for doing just that, a stereo microphone port.

The stereo microphone port in the test computer is a Tip Ring Sleeve (TRS) jack connector, with a common ground between the two channels. To separate the channels into separate mono tracks we used a TRS to RF Coaxial Connector (RCA) splitter, and a custom RCA to mono Tip Sleeve (TS), to capture the two separate right[1] channels from the audio sources. [2]

Furthermore we limit our tests to a Master/Slave/Slave setup, that is; one phone acts as a master, and we perform the tests on two slaves connected to it. This makes the two devices under test as similar as possible, since neither has the reference sound. Ideally we would also record the reference, the Master, but since we only have two input channels that is not possible.

In the tests we use three Android phones:

Master OnePlus One (Android 6.0.1, Kernel 3.4.112-cyanogenmod-g62d75e8)

Slave 1 Moto X Style (Android 6.0.1, Kernel 3.10.84-perf-gb67345b)

Slave 2 OnePlus Two (Android 6.0.1, Kernel 3.10.84-perf+)

4.1.1 Test Scripts

We automate the mechanical part of the testing, such as doing the analysis and controlling the variables. To do so we use scripts written in Python, which use the Android Debug Bridge (ADB) interface to control the Android phones. The tests are executed by a Python script to ensure consistency and minimize the chances of error caused by humans.

Measuring the offset between the two signals is a signal processing task. Within the field of signal processing, the specific measure is called cross correlation. Cross correlation is normally a computationally intensive task, but due to the Convolution Theorem[7], it can be solved rather quickly with fast Fourier Transform (FFT). The algorithm to calculate the FFT of the audio offset thus becomes doing a FFT on one of the signals, and multiplying the results with a time reversed FFT of the other.[3][4] Finally we do an inverse FFT to get the actual correlation value. The index of the maximum value is the sample offset which gives the best correlation.[5] By multiplying the number of samples by the time between samples, we calculate the millisecond offset between the two audio tracks.

The scripts used for testing can be seen in [6]

To determine the accuracy and reliability of the tests, we do what we call an assurance test. The variable we need to assure ourselves of, is inter–stereo delay of the microphone input on the measurement computer. If there is an intrinsic delay between the left and right channel of the computer, the whole test will be compromised, or at least in need of adjustment. We use the delay measuring script above, along with a standard AUX cable, and the test song in mono, to detect the delay between the left and right channel. In our equipment the delay is 0ms consistently, which means the results wont be tainted by inter–stereo delay.

[1] Is it right – *Jesper*
[2] Vejleder vil gerne have et billede el. illustation gider vi det? – *Troels*
[3] Jeg forstår ikke denne sætning – *Thomas*
[4] Den har aldrig givet mening – *Marc*
[5] vejleder har bedt om illustration af dette såvel som en forklaring på hvad cross correleation har med offset at gøre – *Marc*
[6] Insert the final scripts and a ref here – *Jesper*

4.1.2 Test sample

We use the song *Goodness Gracious* by *Ellie Goulding*. We did not want the apps to change song, possibly causing a resync, while testing. To avoid that possibility we loop the song for 2 hours within a single mp3 file.

4.1.3 Connecting

Since the apps being tested are network connected systems, we have to connect them before the test can begin. The connection procedure is slightly different for each app. Common for both apps is that we use three devices to test, one is the "master" while the others are "slaves". The "slaves" are the ones connected to the computer with the dual mono to stereo cable described earlier, and where we are measuring the offset.

AmpMe AmpMe allows the devices to connect over the public Internet. We test them on the AAU internal network, where they are technically connected, but the internal network topology makes them unable to connect directly. To connect slaves with the master, we start the master playing a song, thereby starting a "party" and allowing the slave devices to connect. Once the slaves have connected we restart the song on the master and wait for the app to resume playback.

SoundSeeder SoundSeeder does require the devices to be able to connect directly. We fulfilled that requirement by having the master serve a ad-hoc hotspot, which allowed the devices to connect, while keeping the slaves similarly configured. SoundSeeder does not require a song to be playing for it to connect. For the tests we connect the slaves to the master, and only after they are connected begin playback.

4.1.4 Volume

The volume of the devices should not be particularly important, since the method we use for correlation should not care about the relative amplitude. Regardless we will need to keep the volume low to avoid clipping, since the microphone input is amplified. For the tests we keep the volume levels of both devices at 3 steps above the minimum in Android.

4.2 Test Procedure

To make the tests reproducible we follow a specific and detailed procedure for every test.

Offset To test the offset aspect of the apps, we use the method described in Section 4.1. We plug the devices into a computer, and take five samples, each of which are the median of five ten-second measurements. The measurements are correlated using the FFT method also described in Section 4.1.

Manual adjustment Testing the manual adjustments we generally follow the same methodology as the offset test. We plug the phones into the computer headphone jack, and start the script[7]. The script takes five measurements and moves the slider one step for every iteration. It

[7] Evt. angive hvilket script når de kommer i bilag – *Thomas*

starts off moving the adjustment slider, seen on Fig. 3.2b on page 8 and Fig. 3.1b on page 7 [8], to the right, until the maximum adjustment has been reached, it then moves back to the initial position and repeats for the negative adjustment values. After the last negative adjustment it returns to the median slider position and records a final set of five measurements.

Playback consistency Playback consistency is, like the previous test[9], done by a script. The script takes five measurements and then switches song. It does this five times. For AmpMe switching song can be triggered by clicking the previous song button, while SoundSeeder requires the app to go to the next song and then back to the previous.

Drift The drift of an app necessarily takes a while to test. Like the other tests, a script is written to handle the testing part. The test is accomplished by starting playback of a long song, created by repeating a short song multiple times, and then measuring the offset over time. To give us an idea of the drift over we take five measurements every five minutes for two hours.

4.3 Test Results

In this section we present the results of the tests. The interpretation of these is presented later. For all results shown in tables; each of the measurements is the median of five measurements. Furthermore it should be noted that the measurements of which we took the median were generally within $\pm 0.1ms$ of each other.

4.3.1 SoundSeeder

First we tested the SoundSeeeder app, using the setup and methods listed previously in Section 4.1. In Fig. 4.1 on the next page the results of the first test, using the slider to manually adjust the offset, is shown. A total of 115 datapoints were collected in the test. A trend–line for this data can be described by the function $f(x) = 0.9744 * x - 70.497$ with $R^2 = 0.9974$. The trend–line is a linear regression meaning the relationship between the sliders values and the measured values are nearly linear with 1 being exactly linear. Here it should be noted that SoundSeeder labeled their slides with milliseconds as a unit, if this unit was precise then the scaling for the trend–line would be exactly 1. It is a bit less than that, 9.744 milliseconds to be exact.

In Table 4.1 we test SoundSeeder's synchronize button.

Measurement	1	2	3	4	5
Measured delay [ms]	-67.375	-80.729	-87.937	-70.666	-87.250
Relative to previous [ms]	N/A	-13.354	-7.208	17.271	-16.584

Table 4.1: Results of testing the synchronization button in SoundSeeder.

In Table 4.2 on the next page the results of the test, in which we change the song being played, are shown.

[8] I'm going to bet that it's going to get funky with these page numbers at some point – *Jesper*
[9] Igen, hvilket script? – *Thomas*

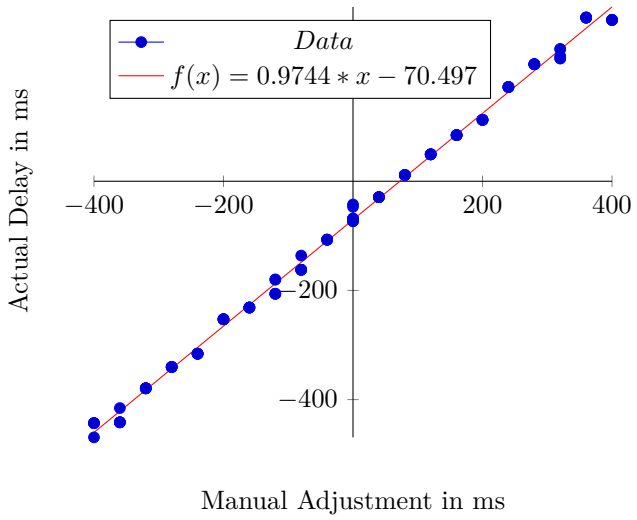


Figure 4.1: Results of testing the manual synchronization slider in SoundSeeder

Measurement	1	2	3	4	5
Measured delay [ms]	-419.854	-376.063	-373.375	-390.646	-361.667
Relative to previous [ms]	N/A	43.791	2.688	-17.271	28.979

Table 4.2: SoundSeeder next song test results.

4.3.2 AmpMe

Secondly, we tested the AmpMe app using the exact same setup as the one for SoundSeeder. In Fig. 4.2 on the facing page the results of using the slider in AmpMe are presented. A linear regression trend–line is drawn onto this data, the function for which is $f(x) = 23.211 * x + 131.260$ with $R^2 = 1.0$. That means that the trend–line matches the data measured in a linear fashion. This also means that each step on AmpMe offset slider is 23.211 milliseconds.

In Table 4.3 we show the result of the song change test. It is notable that the relative offset to previous synchronizations are close to the factor in the trend–line, however the cause is unknown.

Measurement	1	2	3	4	5
Measured delay [ms]	-156.208	-133.146	-110.063	-133.417	-133.562
Relative to previous [ms]	N/A	23.062	23.083	-23.354	-0.145

Table 4.3: AmpMe next song test results.

There is no test for the automatic synchronization in AmpMe, as there exists no button to test.

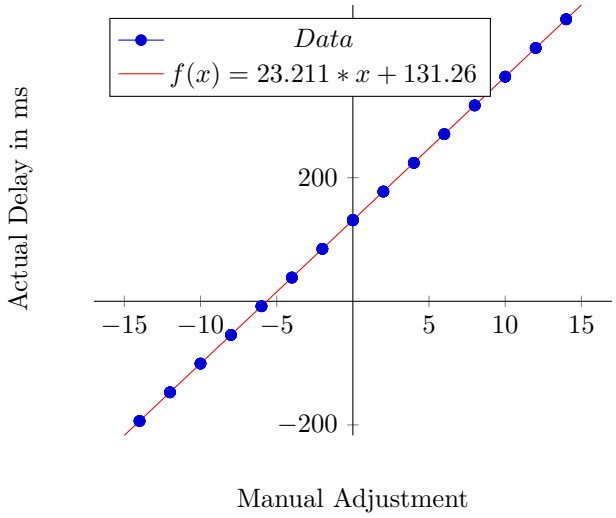


Figure 4.2: AMPme delay Data

4.4 Test Discussion

We have tested the apps which we deemed to be state of the art in Chapter 3, in order to see if they fulfill the initial problem statement we posed in Section 1.1 on page 2. Moreover in Chapter 4 on page 16, we posed several specific questions that the tests should answer. The questions are in three categories: offset, consistency, and drift. We will go through each of them, and try to answer them.

Offset:

Is the audio synchronized without any manual adjustment?

For both apps, the answer is **no**. This can be seen in all of the tests, for example in Fig. 4.1 on page 20 and Fig. 4.2, where the measured delay was $-70.497ms$ and $+131.260ms$ for SoundSeeder and AmpMe respectively.

For SoundSeeder we did the additional test of using their synchronize button. This test, results in Table 4.1 on page 19, show that the measured delay between the two phones are different, after pressing the synchronize button, but it does not get closer to 0. Additionally the change seems arbitrary.

Can the manual adjustment make them synchronized?

In the tests we discovered the actual adjustments the offset sliders make. SoundSeeder shows milliseconds as a unit for their slider, with a granularity of $10ms$. In our tests the results, Fig. 4.1 on page 20, is a bit less than that, $9.744ms$ for every $10ms$ on their slider. This allows users to get within $\frac{\pm 9.744ms}{2} = \pm 4.872ms$ of perfect synchronization.

For AmpMe, which had an arbitrary scale, each value is measured to be $23.211ms$ in either direction in Fig. 4.2. This means that one could, theoretically, always get within

$\frac{\pm 23.211ms}{2} = \pm 11.6055ms$ of perfect synchronization. So the answer to the question, is **to some degree**.

Consistency:

Is the delay consistent between playbacks?

To test this question we played one song, tested the delay, then switched song, and tested it again, repeated five times. If the delay was consistent then the delta of these values should be close to 0, however it is on average $23.182ms$ for SoundSeeder and $17.411ms$ for AmpMe.

For SoundSeeder there was a change when changing song, to such a degree that a user would have to manually adjust the synchronization again. For AmpMe, the change relative to the previous was very close to one step on their slider in three out of four measurements, and the last had almost no change. This is interesting, as using the manual slider could be used to correct this.

However for both apps the delay is **not consistent** between playback.

Drift:

Is the synchronization stable over time?

THIS TEST IS NOT COMPLETED YET.

4.5 Test Conclusion

These test shows the shortcomings of the apps. Neither of them are able to synchronize the audio without manual adjustment, and moreover they can not preserve a synchronization when the song is changed. Thus neither of them answers our initial problem statement.

5 | Problem statement

Our initial problem statement in Section 1.1 refers to the ability to seamlessly synchronize audio playback across multiple devices. Through Chapter 3 *State of The Art* we investigated applications that enable this for smart phone devices, while also exploring proprietary solutions with both software and hardware which use this concept of synchronized playback to produce multi–room audio setups.

We concluded that contrary to the proprietary solutions, the current mobile applications on the market are not solving the problem to a satisfactory degree. This fact, in conjunction with the ease of developing a mobile application compared to a proprietary software and hardware solution, directs our focus on smart phones.

Because of the availability of Android devices in our project group, and no need for special additional hardware, we also choose to develop for Android and not iOS or Windows Phone.

In Chapter 2 *Establishing Use Cases*, we envision different use case scenarios, where some form of manipulated playback of audio on multiple devices could be useful.

All of the above can be summarized as the following problem, which will be examined throughout the rest of this paper.

> How can an Android application be used to achieve manipulated playback, wirelessly, across multiple smart phones, such that psychoacoustic effects can be utilized to enhance playback?

As in Chapter 1 *Introduction* we introduce some new terms, whose definitions in the context of this paper we define as follows:

Manipulated Playback
: The term *manipulated playback* means that the audio is not simply broadcasted to a myriad of devices and played as soon as the device receives the data. Instead we want to control when the playback happens, therefore we define it as manipulated playback and not synchronized playback.

Psychoacoustic Effects
: This manipulation of the audio playback, will also enable us to achieve what is called *psychoacoustic effects*, which means that we can utilize the way the human brain perceives sound to our advantage. This concept of psychoacoustic effects will be expanded upon in [1].

[1] ref to the future – *Michno*

Part II

Requirements & Design

6 | Requirement Elicitation

To elicit requirements for the design and development process of a solution, we gather on the loose requirements posed in Part I *Contents*. For each requirement, we first state the requirement and then we elaborate, thus giving it context and where it came from. We have grouped the requirements in three categories: **technical requirements**, **User Experience (UX) requirements**, and **non–requirements**.

Technical Requirements
The technical requirements are concerned with features relevant for how the devices can interact with one another, they are as follows:

a) **At least two devices must be able to communicate wirelessly with one another.**
To be able to achieve multiple playback across multiple devices, as mentioned in Chapter 5 *Problem statement*, then two or more devices have to be able to communicate wirelessly. Active use of the application, i.e. when devices are communicating in order to play audio, is referred to as a "session".

b) **The device which starts the session, i.e. session host, must be able to assume playback control of all devices in the session, i.e. volume control, pause, resume, change media, seek**
This requirements relates to both psychoacoustic effects and Section 2.1 *Category: Increase Volume and Area*. The session host should be able to control playback to ensure that devices are playing the same media, and obtain the desired psychoacoustic effects. Furthermore some psychoacoustic effects may require that one device controls the volume across any connected devices as the relative volume affects the psychoacoustic effects.

Under normal circumstances each individual device in a session should have control of their own volume and whether they are playing or not, however if the volume affects the desired psychoacoustic effect then the session host may assume control of volume on any device in the session.

c) **It should be possible to manipulate the playback on each device separately.**
As mentioned in the problem statement in Chapter 5, the application must be able to manipulate the playback of the audio. The manipulation refers to applying an offset to the playback on the respective devices such that the playback, can be adjusted to deal with device placement and to help achieving psychoacoustic effects. Furthermore the manipulation may help acquire audible sync.

d) **The application must have the ability to perform an automatic synchronization.**
To be able to use our system for the use cases we identified in Section 2.1 *Category: Increase*

Volume and Area[1], the application must be able to automatically synchronize the devices in the session. This means that when a device is connected to a session, the application must be able to synchronize the playback automatically, so the device plays in sync with the rest of the connected devices in the session.

d.a) **Following an automatic synchronization, the audio offset between any two devices must not exceed x ms.**
With the ability to synchronize automatically, the audio offset between any two devices must not exceed x ms. x is the maximum desynchronization that will not be noticed by a listener.[2]

d.b) **The change in offset following a change in media must not exceed y ms.**
When the media is changed, for instance a change in song, the offset change must not exceed y ms. This means that the media change must be consistent to the extent of y ms.

d.c) **The change in offset after pausing, and resuming playback of any media must not exceed y ms.**
When the media is paused and resumed by the session host, the offset change must not exceed y ms. This means that the pause and subsequently resuming must be consistent to the extend of y ms.

d.d) **The drift must not exceed z ms over n seconds.**
When the media have been playing for n seconds, the drift must not exceed z ms.

UX Requirements
The UX requirements are concerned with features that should be available in the application from the users point of view in order for the application to be usable.

e) **A session should have support for basic audio player functions, e.g. pause, resume, seek and change media**
In order for the application to be useful, simple audio player features are required such that users can pause, play, seek and change media. The change media feature should be reserved for the session host, this limiting of control helps to let songs play out for the use cases mentioned in Section 2.1 *Category: Increase Volume and Area*.

f) **Display song information**
When users use the application in according to the use cases mentioned in Section 2.1 *Category: Increase Volume and Area*, then the song information, i.e. artist, title, duration/length must be displayed.

g) **Support a play queue.**
This requirement relates to the use cases mentioned in Section 2.1 *Category: Increase Volume and Area*. The application should support a queue feature such that a play queue can be developed by the devices in the session. Any device in the session should be able to add audio the the queue.

[1] Automatic sync should be mentioned in use case – *Thomas*
[2] x, y, z and n must be specified. – *Thomas*

h) **Play mp3 audio files from the session host's local storage.**
The application must be able to play audio files of the format mp3 from the local storage on the device. mp3 is chosen since it is in widespread use, Android supports the file format and because mp3 is a compressed file format, resulting in a smaller file size[4][6].

i) **Support Spotify as an audio source.**
The application must be able to use audio from Spotify as source, which means that a session host device must be able to stream Spotify audio to the connected slaves. This requirement is related to the use cases mentioned in Section 2.1 *Category: Increase Volume and Area*, where users having a social gathering must have access to music which is not local on the device.

Non–Requirements
The non-requirements are areas of the application where we do not wish to invest our time, our focus is on synchronization and psychoacoustics, as such the following areas have been down–prioritized:

i) **Security**
Security is one area in which we will not be focusing our efforts. Within a session we choose to trust those that we connect with. As such we will not be putting effort into security concerns.

ii) **Resource Efficiency**
Efficient use of resources, e.g. energy and network, is not something we will be focusing on this project. Optimizations in resource usage might improve the quality of the application, but is not important in order to find a solution to our problem statement.

iii) **Usability**
While we have UX requirements this varies from usability. The UX features necessary for the application to be used, when speaking of usability we consider features like robustness, learnability etc. and we choose not to focus on these elements.

iv) **Scalability**
When mentioning scalability we consider both number of devices and type of devices. We only work with Android devices and will not concern ourselves on how to include iPhone or Windows Phone devices. As for number of devices, we will be working with four phones, expanding beyond this and testing when the system breaks is not something we prioritize.

6.1 Milestones

In order to partition the app into smaller manageable parts we create what we call milestones. We define a milestone as a functional constituent of the ideal app, each milestone must satisfy a subset of requirements and add or improve features from the former milestones. Creating these milestones provide us with more tangible small goals; each milestone acts as a constituent of the application. Thus every time a milestone is completed the application is improved, this provides an iterative structure where we can create milestones ranging from simple and feasible milestones to very complex and time consuming milestones and segment the requirements into these categories.

Basic Audio Player

The first milestone is to make a simple audio player able to play local audio files and display the information of the file. A simple audio player includes only the most basic control features such as skip, play, pause, and volume control. With the success of this milestone the following requirements would be fulfilled: *e*), *f*), and *h*).

Multi–Device Audio Player

With a working basic audio player the application can be extended to multiple devices. This includes limiting control from slaves such that they can not skip or change songs. The successful implementation of this milestone would fulfill the following requirements: *a*) and *b*).

Basic Synchronization

The synchronization is the most central part of the application, we have split this into two separate milestones, the basic synchronization encapsulates reaching the same level of synchronization as the applications in Section 3.1 *Android Apps* did. Reaching this milestone would fulfill the following requirements: *d*).

Manipulating Playback Offset

Regardless of how well we synchronize the audio playbacks will never be exactly the same. In order to adjust for any delays the application should support manipulating playback audio on each device, both automatically and manually. This is also required for us to fulfill the problem statement, as such completing this milestone would not only fulfill the following requirements, but also be considered minimum viable product: *c*).

Advanced Synchronization

We consider the advanced synchronization to be anything better than the applications we examined in Section 3.1. This milestone would not fulfill any new requirements, but it would improve the overall quality of our application.

Psychoacoustic Effects

Our large milestone is to reach a level of synchronization and audio playback manipulation where we can be precise enough to induce psychoacoustic effects upon users. Psychoacoustic effects covers a wide range of features from the precedence effect, which essentially amplifies volume, to 3D sound manipulation. While achieving multiple psychoacoustic effects would create a marketable application, the theory and time to do so, particularly for 3D sound manipulation is not feasible, as such we our goal for this milestone, and ultimately our ideal application goal, would be to successfully utilize the precedence effect.

Advanced Audio Player

This milestone we consider less relevant as our primary focus is on synchronizing devices and manipulating playback, not on the utilities offered as an audio player. This milestone concerns adding features you would expect from an audio player such as playlists, queues and support for external sound sources such as Spotify integration. The implementation of these features would fulfill the following requirements: *g*), *i*).

7 | Architecture

In this chapter we strive to determine both an external and internal architecture for the development of our solution. The external architecture, could also be described as the network architecture, and cover the architecture of the orchestration of multiple devices; while the internal architecture embodies how the intricate parts of the Android app should be structured and composed.

The architectures are presented and discussed in an abstract manner, but will be reevaluated and elaborated upon in a later chapter regarding the concrete implementation.

7.1 External Architecture

This section presents our thoughts on the external architecture. External architecture refers to the architecture between devices, i.e. the network architecture. We consider three different external architectures namely remote centralized, local centralized and Peer–To–Peer (P2P); before deciding upon the one we deem the best fit. Table 7.1 presents some noticeable differences between the proposed architectures.

7.1.1 Remote Centralized

Firstly we consider the option of using a remote centralized solution, i.e. having a server which propagates communication. This solution simplifies the communicative part of the problem by simply using the Internet and using a server as an intermediary between all devices. Using a server also provides the system with a powerful control device, the server, which would alleviate the Android devices from any significant computational needs as they would simply act as speakers and information providers. Furthermore the server would act as an authority, providing a clock to synchronize towards.

This solution also relies on an internet connection being available and is affected by bandwidth, furthermore a server needs to be available, in the case of widespread use we would need enough hardware to support multiple users. Lastly the server would also provide a single point of failure; no server, no application.

7.1.2 Local Centralized

While having external hardware available and requiring an internet connection would limit the application use scenarios, this can be rectified by localizing the solution. To do so a device would act as the server, a master device. The session host would become the master, and any connecting devices would be slaves. A master device would, similarly to the server, provide an authority to synchronize towards. Slaves would retain the status which Android devices holds

in the remote centralized solution, simply acting as information providers and speakers, whereas the master would manage computational requirements, LAN management and file distribution.

In contrast to the remote centralized idea this would remove the need for external hardware and decrease latency however, we also produce the need for peer discovery and LAN management. We would still have the single point of failure in the master, but with no internet requirement.

7.1.3 Peer–To–Peer

A common denominator between the two previous solutions is their single point of failure produced by their master–slave type relationship. A P2P solution would remove this single point of failure, but not without adding its own complications. In a P2P solution all devices would be equal, this in turn means that should a device fail or leave the session, it does not matter which device it was as the network of devices would be unaffected, given that no device is of higher authority. With all devices being equal it also means having to reach consensus to make decisions, simple decisions like playback control could be managed through a majority voting system, but the more complex underlying communication such as synchronization increases in complexity significantly due to not having an authority.

This solution could be expanded to consider a rotating master which would help resolve possible consensus issues, but at that point it becomes overcomplicating the local centralized solution in order to reduce the chance of the master leaving or failing, however that would still be possible.

7.1.4 Conclusion

Ultimately the key challenge and the most important consideration for the external architecture, is clock synchronization. To properly manipulate the clock to achieve psychoacoustic effects, even a few milliseconds matter; as such with how the external architecture can affect this, it is a very important choice.

The three aforementioned proposals are all “pure” proposals, however reality is that they can be modified to resolve some of their weaknesses. An example would be P2P, the particular difficulty with this architecture is the problem of consensus which drastically increases the complexity of clock synchronization, a way to rectify this issues could be having a rotating master. While still making synchronization more complex than the local centralized proposal, it would solve the issue of consensus, however it would also create a single point of failure. Despite some modifications being available, certain variables for each proposal can not be modified, such as the requirement of internet and external hardware for the remote centralized proposal.

The remote centralized proposal holds the most restrictions on the users by requiring internet, and also requires a server to communicate with, which based on location could impose further issues. As such we prefer both P2P and local centralized to this proposal. The pure P2P proposal on the other hand imposes few requirements, but do complicate the clock synchronization a lot, as such we do not wish to use a pure P2P implementation. This leaves us with local centralized or a modified version of P2P, a particular issue with a rotating master however, would be how do we provide the music source, queue system etc. These issues are primarily UX issues, which as mentioned in Chapter 6 *Requirement Elicitation* is not a priority, with this in mind both of these ideas will be considered when implementing the system and the final decision is made.

	Single Point of Failure	Network Resilience	Peer Discovery	Internet Required	Consensus	External Hardware Required
Remote Centralized	True	Medium	N/A	True	Easy	True
Local Centralized	True	Low	Easy	False	Easy	False
Peer–To–Peer	False	High	Easy	False	Hard	False

Table 7.1: Comparison of external architectures

7.2 Internal Architecture

In this chapter we present different options for internal architectures of our Android app. By internal architecture we mean how the parts of our Android app will be composed, and how they will interact. We deem it important to choose an architecture, which will allow for extensibility, maintainability and testability; extensibility, because we want to be able to easily extend the functionality of the app during an agile workflow; maintainability because we want to have the ability to easily rewrite old code and fix bugs; and testability because we want to produce a well tested Android app without having to spend all our time testing.

All proposed internal architectures, will be evaluated in regard to how well these requirements are met, and lastly we choose an architecture to use in the implementation of our Android app.

Choosing a fitting internal architecture is as important as the overall system architecture, and especially when developing something as complex as Andorid apps. Moreover, agreeing on an internal architecture helps the development team stay on track and allows for a faster implementation since the architecture will be the foundation for the app development. However, all of this only holds if the right architecture is chosen, which we will strive to do in the remainder of this chapter.

7.2.1 Candidates for Internal Architecture

In this section we will present three proposed internal architectures; *a*) Repository architecture; *b*) Model–View–Controller (MVC); and *c*) Client–Server architecture. These architectural patterns are often used when designing full systems, but can be adapted to the inner workings of components in a system, such as an Android app in our case.

Repository Architecture

The repository architecture, as described by Ian Sommerville in *Software Engineering*[19, p. 179-180], consists of a single "repository" which manages all data. This data is then accessible by all components in the system, thereby restricting any component to component interaction.

When implementing the repository architectural pattern, the different components can be developed completely independent, and does not need to know anything about each other. This results in extremely low coupling; components can be added and removed without breaking the system, and thoroughly tested independently. An example of applications with the repository architecture, are IDEs; they consist of a single repository containing all project data, and multiple components interacting with the data such as code generators, editors, and analyzers.

A disadvantage of the repository architectural pattern is, that the central repository is a single point of failure. Additionally, this architecture is used in systems that needs to store a large amount of data for a long time, which does not fit our use.

Model–View–Controller

This architectural pattern, also presented in *Software Engineering*[19, p. 176], seeks to separate interaction and presentation from the data, thereby the name; *model*, the data, *view*, the presentation, and *controller*, the interaction. The pattern allows for data to change form without necessarily affecting the presentation or interaction and vice versa. It is a pattern that yields low coupling, and it can help us ensure that the code regarding UX does not interfere with the business logic. This means that the maintainability and especially the testability, of code adhering to the MVC architectural pattern, is significantly higher than code without a clear separation of model view and controller.

A downside to using the MVC pattern is that simple data interactions can become complex and require additional code. Furthermore, there needs to be a clear data model, something our Android app does not have.

Client–Server Architecture

Normally when addressing a client–server architecture, it is in the context of web applications consisting of a client running in the browser which connects to a server running application code. In our case, where we talk about the internal architecture of an Android app, the server is a service, i.e. some components which runs separately and can be connected to. The Client–Server pattern is described by Ian Sommerville in *Software Engineering* [19, p. 180-181]. The clients are only used to make the services accessible by the user, as the services does not depend on the clients, which means that services can be developed and tested without being influenced by user interfaces.

Moreover, this architectural pattern allows for a myriad of different clients to use a given service, such that the same functionality can be available in different scenarios. E.g. if playback of music is implemented as a service, a local client driven by the user could change songs just as easily as a client controlled remotely. This would give a seamless experience of control, since both clients are connected to the same service; meanwhile, they do not need to know about each others existence.

The separation of client and service also allows for more focused testing, since the components can be tested completely independent and the in conjunction. In our case the client–server architecture lies somewhere between the two previously presented architectures, and gives a middle ground where we do not need a heavy focus on data. However, because we want to disconnect client functionality from presentation, to improve testability and maintainability, the concept of a client in the client–server architecture, will have to be designed and developed with a more granular aspect.

7.2.2 Choosing The Internal Architecture

To choose a fitting architecture is a balance act and may end up limiting a project if not done correctly. This is something we want to avoid and as such, we will be picking certain concepts from different architectural patterns, to achieve a tailor made internal architecture.

We deem that the internal architecture should have the client–server architecture as foundation. This means that the internals of our Android app will consist of several services and clients; where the services provides features, and the clients accesses these features and exposes them to different parts of the system or the user.

However, because we are developing an Android app, which inevitably will have some form of user interface (UI), we want to incorporate the idea of views from the MVC pattern. These views will be anything that is to be presented to the user, such as playlists or playback controls. This further separation will enable us to decouple client functionality from client presentation, and allow for testing of the clients functionality independently from the graphical presentation.

Part III

Implementation

8 | Technologies

8.1 The Android Sound Stack

In this section we describe the Android sound stack as per the official *Android Open Source Program*[1]. We look into the Android sound stack to identify parts of the sound stack, which can have an effect on the sound latency and furthermore identify parts of the sound stack which can be customized to cater our needs.

We go through the stack and briefly describe the different parts. During the description we point out the parts of the stack which can be relevant in the design of our system.

The Android sound stack is all the different components which makes up the sound system on Android devices. On Fig. 8.1 the full sound stack can be seen for a standard Android system.

Application Framework

On the top of the stack is the application framework. This is the framework an app, coded in Java, use for sound. For instance the `android.media.*` frameworks, which are a part of the application framework. When using the application framework, the Android Software Development Kit (SDK) is used.

Using the application framework can potentially be slower, compared to the native framework. This is caused by the fact that the application framework is higher up the stack than the native framework and therefore further away from the "bare metal". Therefore it is important to consider if the application framework should be used, since it is the only thing the Java app interfaces with directly, and the choice of framework can have big consequences on the performance and functionality of the app.

JNI

Below the Application Framework in the stack is the Java Native Interface (JNI). The JNI makes it possible for Java code to call and be called by the native applications[9].

Native Framework

The native framework is below the JNI in the stack and used for implementing apps which run natively on the system. To run natively on the system an app can be written in C++. When writing apps for the native framework, the Android Native Development Kit (NDK) is used.

Using the native framework can give improved performance compared to the application framework used with Java[3]. This can result in lower latency, and could potential be an advantage when dealing with synchronization of audio. Therefore the native framework should be considered when designing an app to solve the problem statement.

Binder Inter–process Communication Proxies

One level down in the stack, below the native framework, are the Binder inter–process communication (IPC) Proxies. The purpose of the Binder IPC Proxies is to facilitate communication over process boundaries.

Media Server

Below the Binder IPC Proxies stack wise, but on Fig. 8.1 at the right of the native framework, is the media server. The media server contains the audio services, which is the code that interacts with the Hardware Abstraction Layer (HAL). The HAL is right below the media server, in the sound stack. The sound server implementation on Android is called AudioFlinger, and runs within the media server process[2].

It is possible to change the media server, for instance an example is given where changing from AudioFlinger, to a media server called Superpowered Media Server for Android[1], saved $8ms$ on the round–trip latency on a HTC Nexus 9[22]. The round–trip latency is the time it takes sound to be recorded by the microphone, travel through the audio stack to the user application and then back through the stack to the speakers[22]. Since audio latency can be reduced by making changes to the media server, then it should be taken into account when designing an app to solve the problem statement.

Hardware Abstraction Layer

Below the Media Server in the stack is the HAL. The HAL is the standard interface between the audio services and the audio driver. The HAL is driver–agnostic, which means it can work with any drivers without making any special adaptations.

Linux Kernel

At the bottom of the stack is the Linux Kernel, where the audio driver resides. Different audio drivers exist, e.g. Advanced Linux Sound Architecture (ALSA) and Open Sound System (OSS), which are made for Linux but custom drivers are also available.

Since the HAL is driver–agnostic, then the audio driver can be changed without having to alter the HAL. Several audio drivers exist for Android, e.g. “Voodoo Sound” or “ViPER4Android 音效 FX v2 版”, but it is also possible to make a custom audio driver[21][29]. Therefore the sound driver in the Linux kernel should be considered as well, in the design of an app to solve the problem statement.

8.1.1 Summary

The sound stack on Android have several layers, which sound have to go through, where each layer can potentially add latency to the sound. Some of these layers are more relevant for this project than others, especially the application framework, native framework, media server and Linux kernel. These layers should be considered when an app to solve the problem statement is designed.

[1] `http://superpowered.com/`

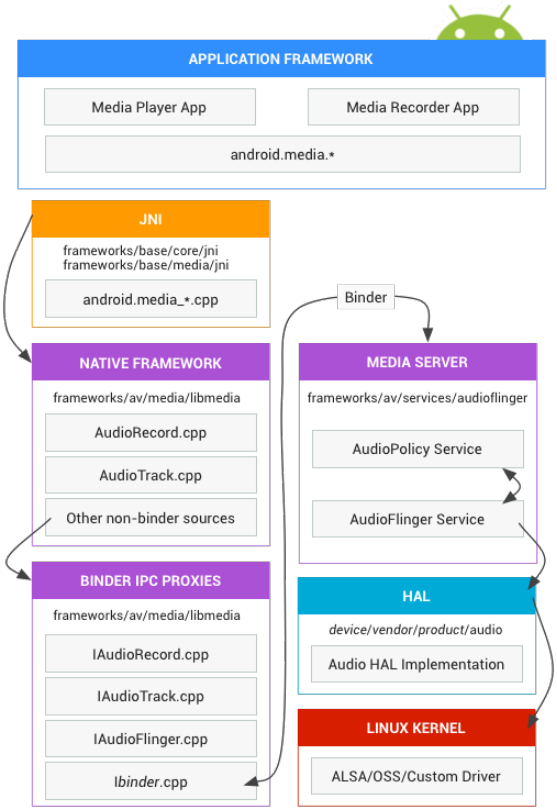


Figure 8.1: The Android audio stack[1].

Bibliography

[1] Android Open Source Project. *Audio*. URL: https://source.android.com/devices/audio/index.html.

[2] Android Open Source Project. *Audio Terminology*. URL: https://source.android.com/devices/audio/terminology.html.

[3] Android Open Source Project. *Getting Started with the NDK*. URL: https://developer.android.com/ndk/guides/index.html.

[4] Android Open Source Project. *Supported Media Formats*. URL: https://developer.android.com/guide/topics/media/media-formats.html.

[5] AVR APPS. *Chorus*. May 2015. URL: https://play.google.com/store/apps/details?id=com.avrapps.chorus.

[6] Ian Corbett. *What Data Compression Does To Your Music*. Apr. 2012. URL: http://www.soundonsound.com/techniques/what-data-compression-does-your-music.

[7] Eric W. Weisstein. *Convolution Theorem*. URL: http://mathworld.wolfram.com/ConvolutionTheorem.html.

[8] Gus Lubin. *Google could be bigger in streaming music than you think*. Aug. 2016. URL: http://www.businessinsider.com/how-many-people-use-google-play-music-and-youtube-music-2016-8?r=US&IR=T&IR=T.

[9] Chua Hock-Chuan. *Java Programming Tutorial Java Native Interface (JNI)*. Feb. 2014. URL: https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html.

[10] Amp Me inc. *AmpMe - A Portable Social Party Music Speaker*. Feb. 2017. URL: https://itunes.apple.com/app/ampme/id986905979.

[11] Amp Me inc. *AmpMe - Social Music Party*. Jan. 2017. URL: https://play.google.com/store/apps/details?id=com.amp.android.

[12] Amp Me inc. *Frequently asked questions*. URL: http://www.ampme.com/faq?locale=en_US.

[13] JekApps. *FAQ*. URL: http://soundseeder.com/help/.

[14] JekApps. *iOS and Windows support soon?* Mar. 2015. URL: http://soundseeder.com/support/topic/ios-and-windows-support-soon/#post-751.

[15] Martin Irvine. *Understanding How Spotify Works*. 2013. URL: https://blogs.commons.georgetown.edu/cctp-797-fall2013/archives/557.

[16] Chris Mellor. *Sonos burns its Bridges: Our home-grown Wi-Fi mesh will do*. 2014. URL: https://www.theregister.co.uk/2014/09/02/sonos_ditches_music_streaming_bridges/.

[17] Dan Planet. *Multi-room audio with multicast RTP*. 2014. URL: http://www.danplanet.com/blog/2014/11/26/multi-room-audio-with-multicast-rtp/.

[18] PulseAudio. *RTP/SDP/SAP Transport*. 2016. URL: https://www.freedesktop.org/wiki/Software/PulseAudio/Documentation/User/Modules/#index38h3.

[19] Ian Sommerville. *Software Engineering: (10th Edition)*. Edinburgh Gate, Harlow, Essex CM20 2JE, England: Pearson Education Limited, 2016. ISBN: 1292096136.

[20] *Spotify — Behind the Scenes*. 2011. URL: https://www.csc.kth.se/~gkreitz/spotify/kreitz-spotify_kth11.pdf.

[21] "supercurio". *Voodoo Sound*. May 2013. URL: https://play.google.com/store/apps/details?id=org.projectvoodoo.controlapp.

[22] Gabor Szanto. *Android's 10 ms Problem? SOLVED.* July 2016. URL: http://superpowered.com/superpowered-android-media-server.

[23] Mike Tanasychuk. *What is Sonos and how does it work?* 2016. URL: http://www.imore.com/how-does-sonos-work.

[24] *Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery*. 2011. URL: https://www-users.cs.umn.edu/~viadhi/netflix.pdf.

[25] Wikipedia. *RTSP*. 2016. URL: https://en.wikipedia.org/wiki/Real_Time_Streaming_Protocol.

[26] Wikipedia. *Sonos*. 2016. URL: https://en.wikipedia.org/wiki/Sonos.

[27] Zac Hall. *Spotify now has 100M users, but only twice as many paid customers as Apple Music*. June 2016. URL: https://9to5mac.com/2016/06/20/spotify-apple-music-users/.

[28] Cong Zhang et al. *On Crowdsourced Interactive Live Streaming: A Twitch.TV-Based Measurement Study*. 2015. URL: https://arxiv.org/pdf/1502.04666.pdf.

[29] 影（尹湘中）. *ViPER4Android 音效 FX v2 版*. Mar. 2015. URL: https://play.google.com/store/apps/details?id=com.vipercn.viper4android_v2_fx.

Appendices