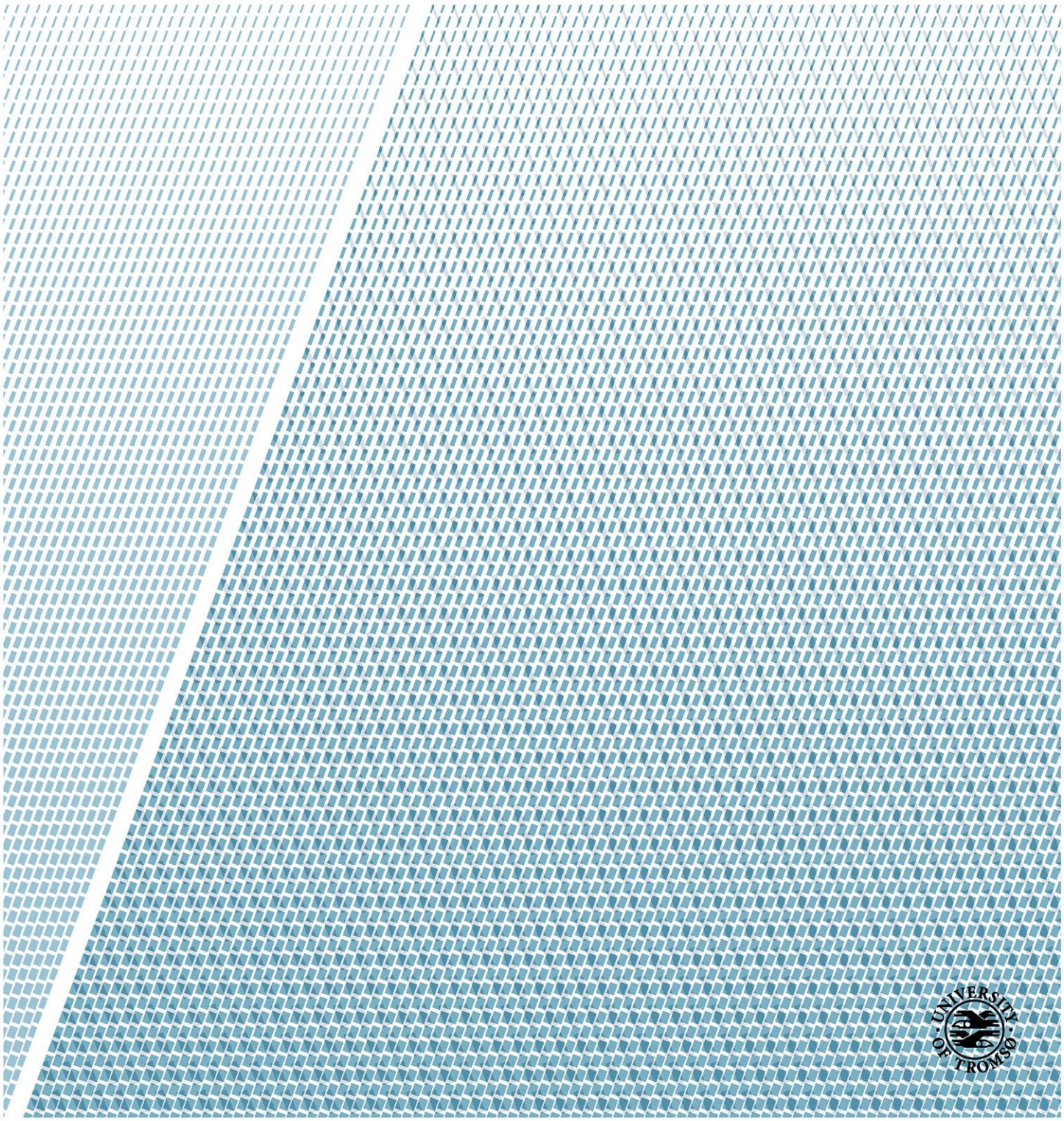
UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Cloud Based Surround Sound System

Johannes Larsen

INF-3983 Capstone Project in Computer Science - December 2015



Abstract

The HPDS[1] research group at UiT has during the last decade done lots of research on how to logically map visual content onto display capable computers organized in a distributed system, but these systems have up until now had little or no support for presenting audio components related to the material that is being displayed. One of the most promising projects for mapping visual content is Lars Tiede's Display Cloud[11], which introduces a cloud model for users to organize freely both displays and visual components on an abstract canvas, such that anywhere a visual component intersects with any display on this canvas, that part of the visual component is shown on that physical display. This project extends that model with an audio dimension, thereby creating a system capable of organizing freely audio-visual components on a canvas among both virtually mapped displays and speakers that together comprise a surround sound system. It maps the source's original stereo image relative to its position with respect to the virtual speakers in order for different components to utilize different parts of the displays and the sound system simultaneously, without losing their intended stereo image, or having the components interfere with each other.

This system is suitable for utilizing the potential of multiple displays organized together in order to form gigantic screens and creating arbitrary surround sound systems made up of independent computers and/or speaker systems. Another potential usage scenario is to have audiovisual content virtually following different users, such, as a virtual museum guide that can be configured to present trivia interesting to the specific user, instead of babbling about stuff that the user deems boring.

A prototype has been created that is capable of playing 4K 60FPS video onto a 7×4 display wall with an aggregated resolution of 7168×3072, together with mapping and playing its surround audio track onto a 5.0 surround sound system. The playback happens with maximum 10ms (configurable) inter-player delay, which is synchronous enough to avoid any nauseating effect, and, although synchronization events are noticeable, they do not spoil the movie viewing experience.

[1]High Performance Distributed Systems: http://hpds.cs.uit.no/

Contents

Chapter 1

Introduction

This project mainly focuses on how to achieve synchronous audiovisual playback among lots of different computers. It builds upon Lars Tiede's Display Cloud[11]. because synchronous multimedia playback is a feature that it is missing, and it would benefit greatly from having. His project solves, and dictates, the design for how the displaying of a single visual component is to be distributed among separate viewers running on different computers, so this project focuses on the audio part, because that is yet to be incorporated into his system. However, having separate media players synchronize audio and video playback is more or less identical, so there is no reason to limit the project scope to only consider audio, as video synchronization comes more or less as a free perk.

1.1 Related work

Virtual stereophonic audio mapping has been around for ages. The concept was introduced to the author by a demo provided with audio drivers for Sound Blaster cards that had a buzzing bee flying around in the projected sound image. One of the somewhat commonly known application utilizing it is the VoIP system Mumble[1], which use players relative position and bearing in games to acoustically indicate the position of nearby speaking players. However, these kind of systems tend to require headphones, because that gives them a controllable acoustic environment [12, html/binaural.html], but there exists acoustic research, such as [12], that achieves the same effects using speakers. That kind of research requires much more control over the acoustic environment than what is suitable for this project. There exists research, such as [7], into virtually moving a surround system's acoustic sweet spot, which, as described in Section 2.1.2, is more or less the opposite of the virtual audio mapping part of this project.

Streaming multimedia among different devices is not groundbreaking either. There exists commercial systems, such as Apple's AirPlay[2] and Google's Chromecast[3], that enable devices to synchronously play something on one external multimedia system, but there seems to be no market for extending such systems to include more than one multimedia systems. Researchers at Norut[4] designed and implemented a Media State Vector[1] for distributing synchronized motion among different web browsers. That project is the bases for ongoing work to create a W3C web standard[5] for sharing a perspective of time among multiple devices. Using web browsers does not give enough external control to be easily incorporated into Display Cloud, so their system is not relevant for this project, but their underlying design is the basis for how this project synchronizes different devices. Both [2] and [4] are previous projects extending commodity media players by using NTP-like clock synchronization in order to achieve synchronized audio playback among regular computers with their own sound system. There already exists numerous commercial systems for transferring raw audio via network, and PulseAudio[6], which is the default sound system on most of todays `*nix` systems, supports doing so out-of-the box, but, as these systems are only designed for local networks, they do not work well with the online design of the Display Cloud.

[1] Mumble, open-source VoIP application intended for gaming: https://wiki.mumble.info/wiki/Main_Page
[2] AirPlay, wireless streaming between devices: https://en.wikipedia.org/wiki/AirPlay
[3] Chromecast, remote controllable media playing device: https://en.wikipedia.org/wiki/Chromecast
[4] NOrthern Research institUTe: http://norut.no/
[5] W3C Multi-Device Timing Community Group: https://www.w3.org/community/webtiming/
[6] PulseAudio, open-source sound system for POSIX OSes: https://wiki.freedesktop.org/www/Software/PulseAudio/

Chapter 2

Background

2.1 Stereophonic sound

Stereophonic, or more commonly, stereo, sound refers to it being three-dimensional [8], so it means sound that can be perceived as originating from a given direction. Both stereophonic and stereo are equivalent terms, and the general definition of neither of them refer to audio tracks / speaker arrangements with exactly two channels/speakers, but that is what people commonly associate with *stereo*, since more complicated sound systems are usually referred to as surround sound systems. To avoid ambiguity, all uses of stereo that refers to a two channel/speaker system, as opposed to its more general definition, is typographically emphasized

Lots of mammals, most notably humans, have multiple ears in order to perceive some indication of the location from which sounds originate. The human brain primarily localize sounds by comparing the loudness and time differences from when each ear perceives the same sound. Additionally it uses out-of-phase echo cues formed as the sound waves interacts with the outer ear to further localize the sound in the elevation plane and whether it originates in front of you or behind you [12, /html/localisation.html], but such cues are out of the scope for this project.

Sound localization consists of determining the direction from which the sound originates, and the distance to the sound source, but, as this project only deals with audio generated by stationary sound sources (i.e. our speakers do not move around), this report will ignore the distance measure altogether. This section use the term perceiver extensively to describe something listening to the sound, and when its facing direction is referenced, it refers to the direction that the stereoscopic sound perceiving sensor (i.e. head) is facing.

2.1.1 Human sound localization

Human hearing measures the time difference between when each ear perceives the same sound, and uses it to determine the direction from which the sound originates. When outer-ear echo cues are being ignored, whether the sound originates from the front-half or the back-half of the head is indistinguishable [12, /html/localisation.html], so let us assume a front-facing perceiver for simplicity, but the following description also applies to backwards-facing receivers using a mirrored direction view. Periodic signals pose a problem, because each period consist of an identical waveform, which makes the periods indistinguishable from one another. This leaves phase (relative position within the current period) difference, so let us assume the frequency is so low[1] that the signal reach both ears within the same period. Sound originating directly in-front of the perceiver travels the same distance to each ear, and, as the wave travels with a constant speed, it is perceived at the same time or, given the same period assumption, more precisely with identical phase by both ears, hence if the sound reach both ears at the same time it is perceived as originating directly in-front of the perceiver. If, on the other hand, the sound reach one of the ears first its direction is angled towards this ear. How much, is determined by the measured delay between when each ear perceives it, but that is irrelevant to this project, so it is left out of the scope of this report.

[1] Frequency bound to stay within same period for both ears: $f < F_{\text{Nyquist}} \times 1/\frac{d_{\text{ears}}}{v_{\text{sound}}} \approx \frac{1}{2} 1/\frac{0.2}{340} \approx 850\text{Hz}$

Sound waves do not store infinite energy, so their energy dissipates over time, hence their perceived loudness decrease as a function of the distance from their source. Without knowing the original loudness this cannot be used to determine the distance of their source. However, the head act as an obstacle to the wave, so it partly shadows the waves, which impacts the loudness that is perceived by each ear [12, /html/localisation.html]. A sound wave directed at one of the perceiver's ears will have more energy when it reaches that ear than the smaller part of it that reaches the other ear, since it has been shadowed by the head in between the ears. Hence the same sound perceived with a greater loudness by one ear than the other will be perceived as originating from a direction angled towards that ear.

2.1.2 Virtual sound sources

Having a single sound source creates a sound image where listeners fairly easily can localize this sound source independently of their position in relation to it. Adding more sound sources complicates the sound image significantly, because they *crosstalk* each other, which is to say that the sound waves from one speaker interacts with the sound waves from the other speakers so that the resulting sound wave is different from the intended one [12, /html/xtalk.html].. Figure 2.1 illustrates the crosstalk effect from two sound sources producing identical sound waves. It clearly shows that the loudness (i.e. degree of whiteness) of the resulting sound wave will depend on the location of the perceiver. Since the illustration uses a periodic signal, the time domain can not be directly extrapolated from the wave, so the green and red outlines have been added to show that their intersection point originates from different positions in the signal. The period outlined with green in Figure 2.1 shows the waveform produced 12 periods ago by the right sound source, and the red outline shows the waveform produced 2 periods later by the left sound source.

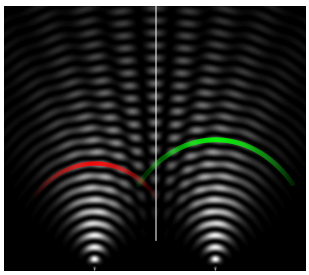


Figure 2.1: Interference diffraction pattern. *Double slit simulated 2 [sources]* license:public domain

A perceiver located somewhere on the vertical line in the middle of Figure 2.1 will have the same distance to both sound sources, so the sound will be perceived as originating directly in front of the perceiver. This yields a virtual, as in perceived but not actually existing, sound source in the center between both of the actual sound sources. If the perceiver moves parallel to the sound sources it would be closer to one of the sound sources than the other, hence it would perceive the sound wave from that sound source earlier and with greater loudness than the one from the other sound source, so the direction of this virtual sound source would be angled towards the sound source the perceiver moved towards.

For the argument's sake let us think of the outlined waves in Figure 2.1 as chirps (i.e. on-off, as opposed to periodic, signals) and ignore the other waves. A perceiver located in the intersection point between the green chirp and the vertical center line clearly perceives the green chirp before the red chirp. So in this scenario the loudness perceived by the left ear (facing towards speakers) will peak earlier (when the green chirp arrives) than the peak perceived by the right ear (when the red chirp arrives), hence the perceived direction will be angled leftwards. On the other hand, a perceiver located in the intersection between the red and the green chirps would perceive both chirps at the same time, and in this scenario the green chirp will have traveled further, so it will be perceived with a lesser loudness, hence from this location the perceived direction of the same chirps would be angled in the opposite direction.

The previous argument presented two different techniques of influencing the perceived direction by timing the sound generation differently, and it tried to introduce the connection this has to the perceiver's relative location. The latter part of the argument introduced amplitude stereo indirectly by moving the point of perception, but the same argument applies to a scenario where the initial loudness of the sound generated is the variable that differs between sources. The perceived direction will be some aggregate of the impact from both these techniques, but they do work independently [12, /html/localisation.html], and exactly how the human brain aggregates these technique is out of the scope of this report. Amplitude stereo is by far the most common technique, but the stereo image it can produce is limited to somewhere between the speakers, because it only angles the direction towards the loudest sound source. The other technique does not have this limitation, but it creates a much more complicated sound stage and the location of the perceiver is fundamental for it to work as intended [12, /html/vsi.html].

Points where the sound arrives at the same time from all sound sources, are known as an acoustic sweet spot [12, /html/vasi.htm]. In Figure 2.1 this would be any point on the vertical line in the center, and it is a line, as opposed to a spot, because there are only two sound sources and they are perfectly

Channel\Configuration	Mono (1.0)	*Stereo* (2.0)	5.1	7.1
Front left	0	0	0	0
Front right	-	1	1	1
Front center	-	-	2	2
Low Frequency Effects	-	-	3	3
Back left	-	-	4	4
Back right	-	-	5	5
Side left	-	-	-	6
Side right	-	-	-	7

Table 2.1: Speaker numbering arrangement and respective audio channel per speaker arrangement

parallel, hence their facing direction will never intersect. This focal point is the only place where the intended sound image can be perceived, because signals perceived at any other point would originate from different points in time and their loudness would differ as the time it has taken from their source differs. However, as presented in Section 2.1.2, this focal point can be artificially moved by delaying sound sources and proportionally reducing its loudness. This is a technique that is actively used in surround sound systems, so that they are not limited to placing speakers at exactly the same distance from the intended sweet spot to dynamically move it, without needing to move speakers [7].

2.1.3 Audio channels

Audio tracks tend to have multiple channels that should be played from different sound sources, as a means of encoding stereo sound. Table 2.1 gives an overview of such channels and their presumed location for speaker arrangements relevant to this project. It presents the de-facto standard channel numbering, but there exists incompatible deviations, however, as the majority follows the presented numbering, all other channeling schemes have been ignored by this project. The table clearly shows that the more complex configurations builds upon their simpler alternatives, so that given a multichannel track the first n tracks are more or less positioned correctly for a n-channel speaker arrangement. So playing back only the first n channels works, but as information is lost, a more common approach is to downmix (play more than one channel per speaker) the channels to the available speaker arrangement. If there are more speakers than audio channels, the track can be upmixed (play the same channel on more than one speaker) However, upmixing is only common from mono to *stereo* to avoid an unbalanced sound stage.

The speaker arrangement notation $n.m$ specifies the number of regular channels, n; and the number of LFE (Low Frequency Effects) channels, m. LFE are typically in the $20 \leq f \leq 80$Hz range and is separated from the main mix as a means of directing them to suitable speakers (i.e. subwoofers). Such low frequencies are problematic to localize, so multiple LFE channels is only common in professional systems (e.g. cinemas). It is up to the speaker system to choose how these sounds are generated, so some systems use dedicated subwoofers to generate them whilst others use main speakers with a frequency range that makes them suitable. This channel may also be filtered out from the main mix automatically by the speaker system and delegated to subwoofers instead of main speakers, something which is common for *stereo* speaker systems as most *stereo* tracks (e.g. music) do not have a separate LFE channel.

2.1.4 Real world

The entire Section 2.1 has presented acoustics from the perspective of an ideal world where nothing reflects sound signals. That is not a property of the physical world we live in, and Figure 2.2 shows the kind of environment that is used for acoustic experiments in order to minimize echoes (i.e. sound reflections). The research focus of this project is computer science, not acoustics, so such an environment is neither available nor suited for the application, which limits the possibilities and incentives for experimenting how correctly the system produces the intended stereophonic image. Nevertheless, this section is important as a prerequisite for understanding the working of virtual stereo imaging, and what impact inter-channel delay will have in any multi-channel acoustic system.

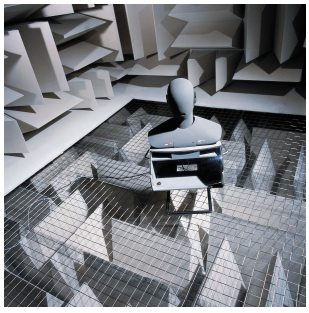


Figure 2.2: Anechoic chamber by:Andrew Eckel license:CC-BY-SA-4.0

Chapter 3

Display Cloud

The Display Cloud is an abstraction created by, and a platform designed and developed by Lars Tiede, for projecting content onto arbitrary displays. The abstraction is a user-specific canvas onto which they can logically map displays and graphical content. Displays in this model are presented to users as rectangles that they can arrange freely in their own canvas. Alongside these displays the users can spawn and freely arrange *visuals* that provide graphical content projected onto the corresponding parts of whichever displays they are intersecting.

The simplest scenario, a single display and a single *visual*, yields two different use cases. The most common is the case when the *visual* is smaller than the display, so that *visuals* acts like a window in a regular desktop user interface presented on the display. This window can be positioned (e.g. dragged by the title bar) and scaled (e.g. dragged by the corner) freely on the display by positioning and scaling its representation in the user canvas. The other extreme would be a *visual* larger than the display, for instance a full-scaled gigapixel image or Prezi[1]-like presentation. In this case the display acts as a viewport to the content, and the content can be panned around in order for the display to show different parts of this content.

Adding more displays produces lots of new possibilities. One would be to overlap displays in order to show the same content on different displays, for instance streaming the same presentation across multiple auditoriums or each computer in a classroom. Another one would be to stack physical displays together and mapping them accordingly in the user canvas, so that they together form a virtual high-resolution display. See Section 4.1.1 for an example of such a system.

The previous examples presented displays coordinated together to present the same content. Another possibility would be to have user-specific content moving from display to display. Such a scenario could form the basis for a virtual museum guide system[2], which keeps track of user movement, directs them to exhibitions they have yet to visit, and presents information about the exhibitions tailored to the preferences of individual users. Or it could act as a good old-fashioned thin-client, which people could use for work not suited for the display on their mobile phone, instead of dragging around laptops for such occasions.

3.1 Overview

Display Cloud is a distributed system, so this section is going to present its different components and their coordination flow.

First there is a set of displays associated with the system. These displays are independent components that are ignorant of each other and the rest of the system, and there does not exist any central component keeping track of which displays are associated with the system. Physical display devices are dumb, so this system requires some known computer acting as a controller for the content presented on any given display. Such computers run a controller daemon associated with this system per physical display connected to it, and these daemons stand by to handle user requests for displaying content on its

[1] Prezi, zoom-and-pan based storytelling tool: https://prezi.com/

[2] Virtual guide, 30 minutes into H.G. Wells's *The Time Machine* (2002): https://youtu.be/CQbkhYg2DzM?t=55s

associated display. The original architecture suggested using NFC[3] tags or QR[4] codes to discover nearby displays [11, 3.], but how these controllers are known is out of the scope of this project, so let us assume the controller daemon for any given display is made available through a known URI. Users of these displays also require some metadata about these displays, such as physical size and resolution, so let us assume that this is either available through the same source providing the URI or that it can be queried from the given URI.

The controlling components in this system are users. As with displays, users are independent components ignorant of each other, and there does not exist any central component keeping track of active users. These user applications decide what part of the graphical content is to be shown on which displays, and coordinate the displaying of the content it would like to have displayed. This coordination means instructing every display overlapping a given piece of graphical content to show a given rectangular selection of the content onto a given rectangle of the screen (i.e. scaled arbitrarily). In these coordination requests users only reference the content that is to be displayed, which decouples users from the content that they would like to have displayed. They can reference publicly available content, such as a picture on the Internet; or pass along authorization information that gives the daemon access to otherwise unavailable content, such as a VNC[5] server for the user's display. How to achieve this kind of peer-to-peer connection is out of the scope of this project, so let us assume that display daemons, users and whoever serves the content can initiate connections with each other either because they are in the same local network or are publicly accessible.

A *visual* is the abstraction of some graphical content with a representation that the user can control. From the user's perspective the *visuals* is an abstract rectangle with some metadata about the content it represents. The design only specifies that the user should act as some sort of broker between the display and the content source [11, 1.], but all content viewers that have been implemented for Display Cloud use off-the-shelf content providers, so there is yet to be such a *visual* source to describe.

Content viewers are the components that display content on their associated display. There is exactly one viewer per *visual* per display that intersects with the user's representation of that *visual*, which is spawned by the display daemon on request by a user the first time this *visual* and that display intersects and is killed when they are no longer intersecting. Upon initialization, viewers initiate a control socket to their corresponding user in order to receive commands, such as for updating their viewport. How they retrieve and display their content, depends on what kind of content they are displaying. Section 3.2 describes viewers relevant to this project.

3.2 Multimedia player

Prior to this project, the Display Cloud supported a picture viewer and a VNC client. VNC screen sharing could in theory be used to support video playback, but that protocol is based on partial frame buffer updates, so it is neither intended for, nor suitable for, continuous video playback, resulting in horrendous video frame rates [6]. Multimedia streams comprised different content sources, out of which the most common are video, audio and subtitles. This project focuses on providing a player capable of synchronized audio and video playback. Adding subtitles could also be useful, but that would benefit greatly from having a opacity controlled text-overlay *visual*, so this is left for future work. Lars Tiede have been working on how to do dynamic video cropping, scaling and positioning, so those parts will be left out of this report.

A multimedia player has some major differences when compared to any prior *visual*. Firstly, it will require audio playback in Display Cloud, which is a system mainly focused on display projections, and, as acoustic span is much less localized and more analog (i.e. not chunkable) than visual span, it is a non-trivial task. Secondly, since this player spans multiple displays and/or speakers it require some sort of synchronization of the different actual players, in order for them all to play the same position in the media source at the same time. The prior *visuals* could make do with implicit synchronization, because they were either static (picture viewer), or push based, with low enough frame rate to avoid noteworthy jitter (VNC client), but that is not good enough for jitter free synchronized audio/video playback. And,

[3] Near Field Communications: https://en.wikipedia.org/wiki/Near_field_communication
[4] Quick Response code, a type of 2-D barcode: https://en.wikipedia.org/wiki/QR_code
[5] Virtual Network Computing, screen sharing protocol: https://en.wikipedia.org/wiki/Virtual_Network_Computing

as opposed to prior *visuals*, there is a time component to the playback, so there will be the need for some sort of interface to control the current position in the media source.

Creating a media player from scratch would a have been too much work for such a small project, and it would be kind of pointless, as there exists open-source media players with external programming interfaces. MPlayer[6] is an example of such an open-source media player, which supports a slave mode with a stream interface for issuing commands and reading their output. MPlayer's slave interface has some known limitations, such as regular textual output being mixed with slave interface output, but it acts to show that there are alternatives and, since MPlayer is more than a decade old, that some of them have been around for some time. Instead this system uses one of its forks, MPV[7], as the media player backend, which solves MPlayer's programming interface problems by replacing the entire slave interface with an improved C library interfaces.

3.2.1 Audio

Projecting a partial *visual* onto a display is a matter of finding the intersection of the source and the given display, and this particular intersection will be irrelevant to any other display which is not intersecting the same area, so it is easy to split the source among different displays. Audio, on the other hand, has much more analog span, where the sound source radiates signals from more or less a single point, as oppose to displays, which forms fixed planes of individual pixels. The simplest virtual audio localization scheme would be to have the virtually closest speaker to a given audio source be the only speaker generating its sound, but this is infeasible, as it would have a terribly low resolution for any realistic sound system. However, as described in Section 2.1, human brains can use echo localization to interpolate sound sources between two of such distinct sound source points, so by generating the same sound by different speakers and variating their balance (relative loudness) and/or timing, results in much greater resolution.

In Display Cloud both *visuals*, or at least their bounding box; and displays are rectangles, but sound radiates from single points and, in an ideal world, sound sources can radiate the same amount in all direction, so it would be better represented as a circular object (i.e. an origin and an equidistance) or semi-circular (i.e. some given periphery of a circle) for directional sources. Sound travels fairly well in air, so the required unhindered distance from a speaker before the sound it generates becomes unnoticeable would be fairly large. For instance sound generated by the author's headset (i.e. a tiny speaker) is noticeable from across the Display Wall Lab ($\approx$ 5m). This fact makes it infeasible to emulate a real hearing range when calculating how an audio source should affect any speakers in the system. A common scenario would be to have some graphical content associated with some sound content and having a playback systems capable of displaying both graphical content as well as generating sound content, hence a good starting point for a virtual sound system range would be for it to span more or less the entire display it is related to, but not much more. This would allow for a audio/video stream with its video projected onto a display to have its audio played back by the associated sound system, and if this audio/video stream were to be moved to another audio/video system, there should be no need for the original sound system to continue playing the audio, even if these sound systems are physically close enough that the sound waves they generate impact each other.

Given such a limited range there are two intuitive methods of encoding virtual localization. One exploits the brain's time-difference localization scheme by simply adding the delay it would take the sound to travel the distance between the virtual audio source and the respective speakers. This is trivial to implement in Display Cloud as the canvas already use real distances (i.e. it uses meter as the unit for offset vectors), and going from real distance to delay is only the matter of defining the speed of sound traveling through this virtual medium to be, for instance, the speed of sound in air. The other exploits the brain's relative loudness localization scheme, by having a sound located between speakers output sound from each of these speakers with a loudness that depends on the audio source's distance from that given speaker.

An ideal speaker does not have a rectangular span, but overlap calculations in Display Cloud implicitly use rectangular bounding boxes. As a compromise to adapt Display Cloud's overlap calculation it only produces sound by speakers within a given ellipsis, but calculates possible overlap based on its enclosing bounding rectangle. That results in superfluous audio players if a *visual* intersects with the corner of

[6] MPlayer, open source media player with scripting interface: https://mplayerhq.hu
[7] MPV, MPlayer clone with C-bindings: https://mpv.io/

the span of a speaker, but these audio players are muted, so they do not directly impact its noticeable output. If it turns out that these superfluous audio players have a noticeable impact on the total system performance it should be trivial to implement ellipsoid overlap calculation, but as of now it seems to be unnecessary, so its left as future work.

3.2.2 Synchronization

Splitting the playback of an audio or a video track among different media players requires the players to be fairly synchronized in order for them to yield a coherent outcome. In order for video players to mostly show the same frame at same time, their relative delay must be within $F_{\text{Nyquist}}\frac{1}{f}$, f being the frame rate of the video, which results in 20ms for regular 25FPS video and a little less than 10ms for the newer 60FPS standard used in motion intense videos (e.g. video game streams). Even less delay would be preferable in order to minimize scene shifting jitter. Audio is even more sensitive to delay, as shown in Section 2.1.2, in order for the playback of different channels from different speakers to achieve the desired stereophonic effect.

Synchronizing these players is a matter of ensuring they play the same position at the same time. In the real world, these players run of different computers, and the clock drift differs between these computers, but for the moment let us assume the different players have the same conception of wall clock time. Section 3.2.2.2 will explain how that is achieved. A simple approach would be to do any operation, such as play, pause and seek, on all affected players at the same time, and assume they will need approximately the same time to complete the operation. This seems to be a somewhat valid assumption, but it do depend on relative position and I/O. Though a major caveat of such an approach would be that any operation, including clock drift compensation, must be executed by every player.

If the execution time of these operation could be predicted, then you could find some point in the future to begin the execution, so that the playback of a given player catches up with the others when its operation finishes. It turns out that resuming playback when the player is ready to play from its given position executes in less than 1ms, which is accurate enough for it to catch up with other players. Which yields a possible synchronization scheme where each player pauses playback and seeks when the operation is received, but waits until some synchronization point in the future, presumably when the pausing and seeking have completed, before resuming playback. This yields the question of how to determine a synchronization point in the future, which requires knowledge about what supposed position playback should have at the point in time.

3.2.2.1 Arbitrary-Time Clock

Querying properties from the MPV client takes somewhere between 5 and 10ms, and when querying something that varies over time, such as playback position, this delay has the side-effect that it adds an uncertainty to the result, because there is no way to know when, within this execution period, the actual measurement was taken. Making the system dependent on such an uncertain position measurement is infeasible, as this uncertainty would further impact the relative delay between players, so the intuitive path forward would be to decouple the reference position from the actual players.

This decoupling of player and control was proposed as a MSV (Media State Vector) by [1]. The MSV is an object that deterministically describes linear motion in real-time as a initial condition and time passed since that condition [1, 3]. This abstraction provides an interface to query what its position will be at any given point in time, which can be used to determine synchronization points locally without the need to involve an other player or any coordinator. In this project a simplification knowns as an ATC (Arbitrary-Time Clock) implements their proposed abstraction.

Their original design incorporates position, velocity and acceleration [1, 3.1], whilst the simplification in this project only support position and speed (i.e. non-negative velocity), because this is the only motion supported by MPV's playback, so any more would be unnecessary, and could introduce misconceptions if someone were to implement a user interface supporting either accelerated playback speed or negative playback speed (i.e. continuous rewinding). And in their design, the actual state vector is an immutable enumeration, whose interface is wrapped by an object that couples modification of its internal state with other functionality, such as clock synchronization and distributed updates [1, 3.4–4]. This kind of coupling makes sense in the peer-to-peer browser scenario they are targeting, where components are split into different websites, hence different applications, but in this project the actual state is going to be

shared among lots of components within the same application, so in this project those concerns are split among different components.

An ATC refers to motion relative to a given timestamp, so, as this timestamp is local to whoever created the ATC, the position it yields for any given time will depend on the local concept of time. Hence, if such an ATC is to be shared among different computers, they need to be aware of their relative time difference. In order to cope with these different concepts of time an ATC has the possibility of being exported, such that it describes motion relative to any remote clock, as long as the exporter is aware of the relative offset of that clock. The different computer clocks tend to be out-of-sync, because they drift differently, so, as these ATCs calculate the position based on their local clock, the result from an ATC on one computer and an exported ATC on another computer will drift relative to their local clocks. When ATCs drift differently, they will yield different results, so the exporting solution requires some sort of synchronization scheme in order for the ATCs shared among different computers to run synchronously. The MSV design incorporates acceleration, so in their system acceleration is being leveraged to give a more correct estimator for the relative clock difference that incorporates linear drift differences [1, 4.2], but it still requires a player capable of explicit slewed synchronization (i.e. de-/accelerate playback motion without frame loss) to make any major difference in how to implement synchronization.

For simplicity, the synchronization design assumes that playback follows the ATCs motion. See Section 5.1 for a discussion on when this assumption holds. If the source requires buffering, MPV will automatically pause the playback, which, if not handled, breaks the preceding assumption, so after the buffering has completed (i.e. MPV unpauses the playback) an explicit synchronization event is triggered to ensure that the playback follows the reference ATC. An alternative approach that catches similar scenarios automatically would be to employ a feedback loop in the player that monitors the position, and explicitly synchronize itself if the playback drifts outside some given threshold.

3.2.2.2 Clock synchronization

Regular clock synchronization (i.e. computers with a NTP daemon) provides more than enough accuracy (i.e. $<$ 1ms offset) to do synchronized multimedia playback, but it requires all participants to have it activated and configured correctly. So in order to avoid such a dependency and to get better control of synchronization events, this system incorporates a passive clock synchronization scheme, which keeps track of the relative clock offsets between the different participants.

Each display daemon runs a fairly standard NTP server with read-only capabilities, so anyone can query it to determine a relative offset, but it does not support updating the actual clock. In the Display Cloud, a user has the sole control of its *visuals*, so the clock synchronization of its *visuals* use the user's clock as their reference, instead of anyone that knows the actual wall-clock time, because relative offset is enough to use for synchronization. Each user has a component that keeps track of the relative clock offset of all the hosts affected by its *visuals*, and it measures the relative clock offset of these hosts periodically using regular NTP queries to their NTP server.

These clock offsets are used to export ATCs relative to the clock of its recipient, so that the motion these ATCs represent, both the user's and the client's, are synchronous. The component that keeps track of relative clock offsets informs the other components within the same application when the relative offset of any host exceeds some configurable threshold (5ms[8]), which in turn inform the affected players with a synchronization request, in order for the media playback to synchronize itself with the updated clock.

3.2.2.3 Synchronization window

A major advantage of having local control of the synchronization points is that the players themselves can determine how large a synchronization window they require in order to catchup with the reference position. This control allows for a dynamic window size, whose size can be determined stochastically.

MPV notifies the controller when a seek operation has completed, so determining whether or not there was enough time for the given operation to complete is a trivial task. Managing the synchronization window happens much like the inverse of how TCP manages its transmit/receive windows ([9, p. 43]), which are being used for network flow control. This system optimistically shrinks the window a tiny amount for every operations that completes within its deadline, and then backs off (i.e. grows the window)

[8] A 5ms clock drift threshold is within the Nyquist ratio of a single 60FPS video frame and constitutes a $t \times v_{\text{sound}} \approx 0.005 \times 340 = 1.7\text{m}$ offset to the acoustic sweet spot.

by a much more extensive amount when deadlines are missed. The current implementation is configured such that it takes about 5 window shrinking to make up for a single back off growth, so as to avoid extensive jitter caused by window size oscillating between every synchronization event. In order to grow a large enough window in a short period of time, every successively missed deadline accelerates the growth rate, much like the window scalar ([3, 2]) in TCP. This enables newly formed *visuals* to achieve synchronization with its peer *visuals* within a couple of synchronization iterations, which in turn reduces jitter caused by spawning viewers (e.g. moving *visuals*).

3.2.3 Composite *visuals*

In Display Cloud *visuals* are organized as hierarchical rectangular spaces, so *visuals* can be defined relative to any arbitrary space, such as another *visual*, instead of the user's canvas. The point of doing so is that moving and/or scaling a space in relation to its parent also moves and/or scales any space defined relative to that space in relation to that space's parent. This enables composite *visuals* that can be moved and/or scaled in unison, such as a multimedia *visual* composed of both video and audio components. And their relative positions are kept intact, so that this multimedia *visual* can have audio channels, such as left and right, being defined relative to the video component, so that the stereophonic effect defined in the source is preserved in this virtual space. This multimedia space is then projected into a virtual space of speakers capable of generating this intended stereophonic effect by differentiating the position of these audio channels.

Having these hierarchical spaces solves the problem of positioning and/or scaling composite *visuals* together in the spatial dimensions, but multimedia content also has a time dimension. And in order for the different components to playback their content in unison they need to agree upon what position to play. Section 3.2.2.1 describes the ATC as a method for defining a reference position, which can be shared among different parties, such as all viewers for a set of separate *visuals*, in order for them all to playback their media in unison. However, as described in Section 3.2.2.2, synchronization happens lazily, so there is a need to ensure synchronization whenever the reference ATC changes. This responsibility is delegated to playback controllers. Each one responsible of controlling the motion of a given ATC.

The playback controller supports regular playback controls[9], and all of these controls simply wrap motion modifiers of its underlying ATC. This only operates on the motion of the reference ATC shared among the different *visuals*, and forwarding instructions to the actual viewers is simply a matter of exporting the updated ATC to them, by forcing a synchronization event. Decoupling the playback controller from the *visual* grouping adds another level of dependency registration, because this playback controller requires a list of callback objects to synchronize, but it also enables coupling media sources in the time domain, whilst keeping them spatially separate. One example of such an application would be the playback of a recorded keynote presentations, which commonly has some audio/video stream of the speaker timely coupled with the slide show that is being presented. Being able to move the slide viewer independently of the video stream offers greater control to viewer, which can avoid the common pitfalls, such as too small keynote fonts or having the presenter video shadowing important parts of the keynote, resulting from exporting the entire keynote as a PiP[10] video.

3.3 Prototype

A working prototype for this system is built as an extension to the original Display Cloud prototype, which is under active development. The entire Display Cloud system is primarily written in python 3.5, and to maximize the likelihood that this extension is kept as a part of the code base the extension is also written in python, and tries to follow the project's conventions. Both the extension written in this project and other parts of Display Cloud use python's *asyncio* language extensions, so it is not backward compatible with older python versions. At the time being, the project use MPV 0.13.0 as the off-the-shelf media player, but, as MPV is under active development, keeping it up to date would probably be a good idea. MPV's C API is interfaced through a python wrapper module, `mpv.py`, created as part of a web

[9] Supported playback controls: pause/resume, absolute/relative seeking and speed control
[10] Picture-in-Picture: https://en.wikipedia.org/wiki/Picture-in-picture

interfaced media player, aesop[11]. In MPV the `pan` audio filter[12], is used to direct a given set of input channels to an arbitrary speaker connected to the player's computer, and the overall media volume is used to position it in relation to its output speaker.

Media sources are URIs that MPV is capable of handling. The synchronization scheme requires sources capable of absolute indexing, so the system is only capable of synchronizing streams if it can determine some absolute position shared among all players. In addition, the playback controller automatically starts playing from the beginning, so today's system will not be capable of handling live streams. For this project, video cropping and placement is statically handled by MPV, so these video viewers do not support being moved around or scaled after initialization. However, Lars Tiede have a working prototype capable of dynamically moving and scaling the video viewers, by manually controlling an OpenGL[13] surface onto which MPV renders its video playback. It should be fairly easy to make this compatible with the synchronization scheme developed in this project, so in the near future, his project will probably support synchronized video playback that can be moved around.

In the Display Cloud *displays* are simply rectangles associated with a display specific controller and *visuals* are simply rectangles with an *visual* controller. In an ideal world adding an audio component to such a system is the matter of refactoring their commonality to some shared ancestor, such as a component producing observable content (i.e. *emitter*) and something that can be observed (i.e. *observable*); and adding the audio specialization, such as a *speaker* and an *audio*. However, doing so requires tremendous changes in the class hierarchy of Display Cloud, and, as this is a distributed system, that is a non-trivial task, since components are implicitly designed around the current set of base classes. A possible simplification would be to have the new *audio*/*speaker* classes inherit from *visual*/*display*, but that yields an inherently skewed class hierarchy. So, as an incentive for the class hierarchy to be refactored correctly in the future, en equivalent feature is implemented as hack, where *displays* associated with and audio output channel are the only ones handling and handle only *visuals* associated with some set of audio source channels.

[11] aesop, web interfaced media player: https://github.com/nathan-hoad/aesop
[12] `Pan`: matrix with the relative impact of each input channel (column) on each output channel (row) [5, man/af.rst]
[13] Open Graphics Library: https://www.opengl.org/

Chapter 4

Experiments

4.1 Experimental setup

4.1.1 Display Wall lab

The HPDS group at UiT has a distributed systems lab called the Display Wall (*Visningsveggen*). The actual Display Wall is a wall-to-wall canvas fitted with a 7×4 projector grid behind it. Each projector has a 1024×768 pixels resolution, so, if they are coordinated, the entire wall makes a 7168×3072 pixels rear-projected display. The coordinated control of these projectors is handled by a compute cluster having one compute node per aforementioned projector.

Along with the display wall, the lab is fitted with numerous input sources and a 7.1 surround sound system. This surround sound system has a single amplifier, and it is connected as a 5.0 sorround sound system, shared among 3 different compute nodes[1]. All of its channels could be have been connected to a single compute node, but that would route all the audio through the same audio pipeline, and as this is a **distributed** audio system experiment, that would be kind of counterproductive. Some of these nodes are responsible for more than one channel, which is not strictly necessary, as there is no shortage of audio capable compute nodes, but it is configured like this, since a realistic usage scenario is single computers with one display and its own stereo sound system.

In some experiments the audio output bypasses the amplifier, and the audio output from separate computers are being recorded in separate channels by the audio line input on one external computer for further analysis. The recording happens externally, as opposed to using software defined loopback interfaces, to measure channel correlation after the audio has propagated the entire audio generation pipeline. It is recorded by a single audio card, so that the audio propagates through the same processing pipeline, thereby being recorded without yielding further impact to its real-time correlation.

4.1.2 `rocksvv.cs.uit.no` (the display wall cluster)

This is a uniform compute cluster, so all compute nodes have identical hardware. The login node has an extra Ethernet interface, and a less powerful graphics card, but otherwise it has identical hardware to that of the compute nodes. These computers are HP Z400 workstations with an i7 3.06GHz hyperthreaded quad-core CPU[2] and 12GB of DD3 RAM. Each compute node has a GeForce GTX 560 Ti graphics card with 8×48 CUDA-cores and 1GiB GDDR5 RAM.

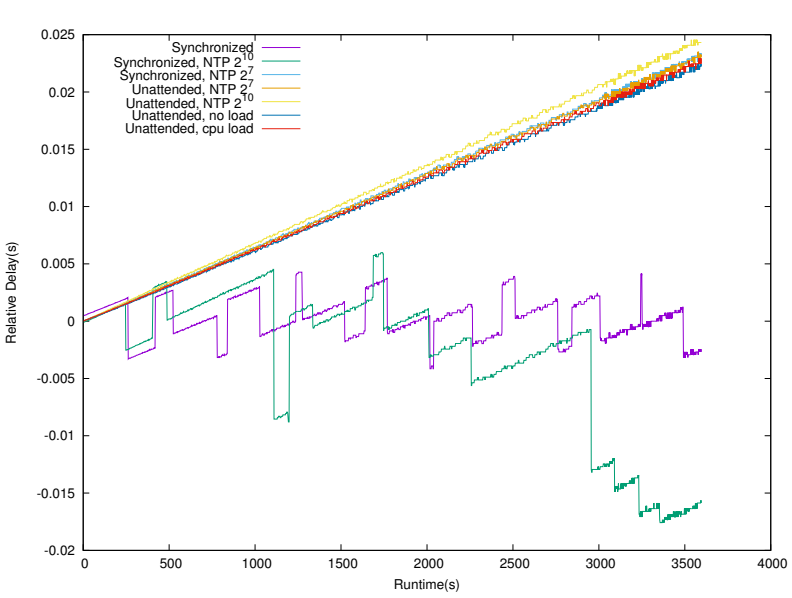
The compute nodes use PulseAudio to multiplex audio applications, and output audio through an Intel 82801JI HD Audio Controller via ALSA drivers. The audio pipeline is configured identically across all compute nodes, and it uses more or less the default Ubuntu 14.04 configuration.

The computers are connected together in a flat hierarchy with 1Gb/s Ethernet, and WAN is routed through the login node. Clients are connected to the internal LAN through Ethernet, or a through an old WIFI access point.

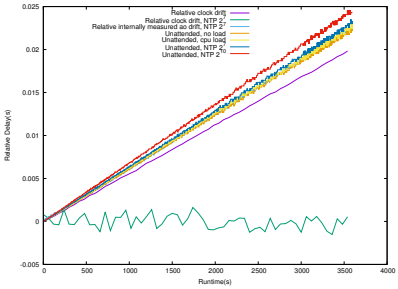
[1] front/rear left($= 0, 4$)→`tile-6-0`, front/rear right($= 1, 5$)→`tile-6-1` and center($= 2$)→`tile-6-2`

[2] `Intel(R) Xeon(R) CPU W3550`

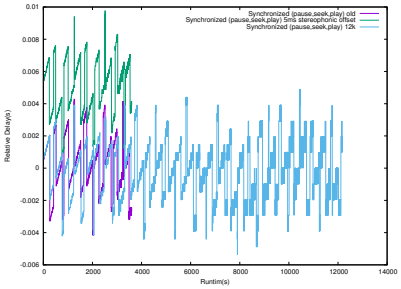
4.2 Synchronization



(a) Synchronization schemes



(b) Playback impact



(c) Synchronization stressed

Figure 4.1: Relative drift between two separate computers. "NTP n_{Seconds}" refers to the period between NTP synchronizations.

The first step of figuring out how to achieve synchronous media playback is to determine what factors impact the time-domain of media playback. The frequency of an oscillator depends on the actual unit and it drifts with temperature, and, as oscillators are used as a basis to define the CPU clock rate in computers, the difference between oscillators tend to be the major culprit when it comes to drift between different computers. In systems doing precise measurement of relative signal phase a single oscillator tend to be shared among the different parts of the equipment to avoid relative drift, but this is not common for computers, and in any case it is not viable for an online system, so different clock drift must be considered a potential candidate for playback drift. A media player has to do some sort of processing to playback its source, so lack of resources during the processing phase can cause it to miss the deadline for when the computation has to be completed. There are different methods of dealing with real-time computation constrains, among others ignoring it or sacrifice n frames to ensure that the deadline after n frames is met, and some of them might impact playback, so load (i.e. lack of resources) must also be considered a possible candidate.

An experiment was formed to measure the difference in playback motion. Two players on different machines[3] are responsible for playing the same signal on their own channel in a *stereo* recording system. Then the recording from this system is analyzed to determine how their playback drift in relation to each other. The test signal is a 0.5Hz click track generated by Audacity[4], which is analyzed with a simple peak detector to determine the position of each tick for each channel, and if we assume the clicks always are within the same period (i.e. less than 1s drift), then their relative time difference represents the drift difference of the signals. It should be mentioned that some of the synchronization procedures cause intermediate interrupts in the playback, so, if this happens during one of the peaks, that player skips this given sample, and in such case the whole sample is ignored. The following plots present runtime, but from different experiments, so they only present relative difference as a function of runtime, not actual time correlation.

MPV has an option to dump various statistics during playback [5, options.rst#program-behavior], and one of the measurements[5] represents difference between audio playback position and the system clock. Data from this is compared to the experiment above, as a argument for its correctness, and to show that MPV is internally aware of how synchronized its playback is with respect to the system clock.

In order to compare playback motion to clock drift, some sort of clock drift baseline is needed.

[3] `tile-6-0` → left, `tile-6-1` → right

[4] Audacity, open-source audio editor: http://audacityteam.org/

[5] MPV `--dump-stats=<filename>` measurement, *ao-dev* (Audio Output DEVice)

To obtain this baseline, clock drift with respect to a reference wall-clock time[6] has been measured periodically. For this to be comparable to the relative playback motion, it also needs to be a relative measurement between the different computers, so difference in, as opposed to absolute, clock drift is presented in the results.

Figure 4.1b shows how relative unattended playback drift (squiggly lines) relates to relative clock drift (smother lines). Figure 4.1a shows this system's synchronization scheme's impact on playback motion, and it is compared with the impact external synchronization has on this system's synchronization scheme. Figure 4.1c is included to show (cyan) that the synchronization procedure works over longer periods of time ($3:30$hours) and (teal) that the time-based stereophonic filter works as intended (i.e. mean is offset by 5ms) when the playback is being synchronized.

4.2.1 Playback jitter

Source	Media size	Window size (ms)
Audio	any	5–10
Video	(4K) 2160p	300
	1080p	200
	≤720p	100

Table 4.1: Synchronization window required during regular playback (i.e. buffered)

To get an impression of how synchronization events impact playback, a simple experiment was formed to measure the minimum synchronization window size needed for players to meet their deadline. The experiment involves decreasing the synchronization window until different parts of the system starts missing their deadline, and it was repeated for different media sources, to show how the media type and resolution impacts time required to achieve synchronization. The window decrements were initiated manually, and, although repeated multiple times per media source, these experiments are only meant to present ball park synchronization window requirements. In order to avoid I/O rate as a variable, these measurements are taken during regular synchronized playback, because then the players have enough data in their buffers to achieve synchronization, without requiring I/O operations.

Table 4.1 summarizes the synchronization window requirements for different media sources.

4.3 Media sources

This system have been tested with regular files, both on the local file systems and via NFS[7]; and network streaming from HTTP[8]/-S[9] sites serving static videos.

The highest resolution video tested with this system is a 4K 60FPS version of Big Buck Bunny[10], and, as this video is free to use, Figure 5.1 have been included to show[11] the coordinated, amongst 28 video players and 13 audio players, playback of this movie at the Display Wall Lab. The highest bitrate tested with this system is 22.3Mb/s, which was from a 4K version of a VGHS[12] episode[13].

[6] Wall-clock time is defined by the NTP stratum 1 (i.e. connected to GPS receiver or atomic clock) server `ntp.uit.no`

[7] Network File System: https://en.wikipedia.org/wiki/Network_File_System

[8] HyperText Transfer Protocol: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[9] HyperText Transfer Protocol Secure: https://en.wikipedia.org/wiki/HTTPS

[10] Big Buck Bunney, a 10 minute long animated movie licensed with CC-BY-3.0: http://www.bigbuckbunny.org/

[11] Video recording at https://static.johslarsen.net/uit/vv-bigbuckbunny.webm, as a single picture can hardly represent multimedia playback. The recording happened during a time when NTP compensated for the computers' clock drift, so there are no synchronization events during the entire recording, yielding synchronous video and audio of up to about 10ms drift in relation to each other and the video.

[12] Video Game High School, a crowdfunded web series: http://www.rocketjump.com/VGHS

[13] Digital download of VGHS S03E01, and played for academic purposes from the author's personal collection

Chapter 5

Discussion

Figure 5.1: Big Buck Bunny (60FPS 4K) played on display wall, media source by:Blender Foundation license:CC-BY-3.0

5.1 Synchronization

Figure 4.1b clearly shows that playback drift is more or less proportional to clock drift. There has been some work trying to figure out why there is a slope difference between the experiments, and the most prominent theory is that it is caused by clock drift on the recorder. Comparing playback when both computers are idle (yellow in Figure 4.1b) to when one of them is dummy loaded (brown in Figure 4.1b), shows that this particular track is not affected by lack of compute resources. However, this is a small audio file, so it is not representable when it comes to resource requirements, and during movie playback excessive load causes observable playback lag.

The playback motion when the computer clocks are NTP synchronized (blue and red in Figure 4.1a) overlaps with those from when it is not, which indicates that clock synchronization does not directly impact playback motion. This is explained by how MPV internally handles audio playback and its synchronization with video. The default behavior is that audio is passed directly to the audio output driver at a rate determined by the output driver and video frame rate is synced to the audio rate, or system clock if there is no audio. There are options to control how audio and video are synchronized during playback, but these settings falls back to using the audio output driver rate if certain properties of the video and/or video driver can not be determined, which is always the case if video is missing or

disabled. And, in any case, the other modes are meant to synchronize video to its output driver instead of the system clock [5, options.rst#miscellaneous]. MPV's internal audio drift measurement (cyan in Figure 4.1b) is proportional to the equivalent measurements (among others, yellow in Figure 4.1b), so MPV is aware of the audio drift, and, as it correlates to the externally measured drift, these different experiments act to produce confidence in each other's correctness. However, there does not seem to be any method of externally (i.e. via the API) querying MPV's drift measurements, or having MPV exploit them to counteract drift, though, as this is an open-source project, anyone, who deems it useful, is free to implement such behavior.

This system uses separate players for audio and video, so with the current implementation MPV implicitly synchronizes the video and the system clock (i.e. synchronized if NTP is configured) whilst audio is implicitly synchronized to the audio output driver (pulseaudio), which, as shown by the NTP unattended plot (yellow in Figure 4.1a), drifts proportionally to clock drift and is not affected by NTP-like clock synchronization.

Figure 4.1a shows that doing nothing (red and blue) results in linear drift proportional to the difference in clock drift on the different computers, and it has been included in this plot to act as a baseline for the different synchronization schemes. The pause-seek-play synchronization scheme (purple) keeps the playback within the slack (±5ms) defined by clock monitor, so this scheme keeps the playback more or less synchronized. In this plot the sudden jumps are the effect of a synchronization event happening on one of the players, and during the actual event that player suffers intermediate jitter, so minimizing the amount of synchronization events is preferred.

The clock offset monitor in today's system measures the system clock, and, as this might be externally synchronized (e.g. NTP), having system-wide clock synchronizers cause some problems. This interference manifests itself in different manners depending on clock drift in relation to the allowed synchronization slack. Figure 4.1a (teal and cyan) shows how it impacts playback motion. Having NTP synchronize the clocks so often that the difference between the monitor and the player never exceeds the allowed slack (cyan) results in no synchronization events, since the monitor thinks the player is synchronized within its slack, which results in linear actual playback drift, since the audio playback drifts proportional to the raw clock drift. On the other hand, letting the clock drift exceed the allowed slack before synchronizing it with NTP (teal) results in the different clock synchronizer compensating each other, which yields an error up to the sum of their slacks, so it is always worse than not having the system clock synchronizing itself. This problem can easily be avoided by consistently using a monotonic clock not affected by any adjustments (i.e. `CLOCK_MONOTONIC_RAW`) for clock monitoring and the ATC. Doing that would impact synchronization between audio and video, since video is synchronized in relation to the adjusted system clock, but in any case both audio and video players should be within the slack allowed by the clock monitor. In this scenario players only playing video should automatically synchronize themselves to wall-clock time, so it is possible video players require less explicit synchronization than audio players, though too much synchronization should not hurt anything more than video synchronization in today's implementation. In any case this will require more research if the system is modified to be usable along with external system clock synchronization, but today running it on computers that do not synchronize their clocks solves the interference problem and is good enough.

5.1.1 Playback jitter

As shown by Figure 4.1a (purple) synchronization events occurs around every 5 minutes per player. However, the length of this period will depend on the player's clock drift in relation to that of the monitor. The impact of jitter on multimedia playback is a subjective matter, so determining the optimal trade-off between jitter and drift slack is out of the scope of this project. Table 4.1 summarizes ball park requirements for synchronization window sizes during regular playback of different media. It differs between type of (i.e. audio or video), and resolution of media, because decoding high resolution video requires more computation than decoding lower resolution video or audio.

Players buffer enough data to seek within this window during regular playback, hence no data is fetched from the underlying I/O device, so the storage medium have been left out of this comparison. Pretty much any jitter from major[1] audio components is noticeable. Minimizing this jitter helps reducing

[1] Given audio sources: a_1, a_2, playing from the same speaker with a relative volume: $a_2 = \frac{1}{3}a_1$, a_2 is will be barely noticeable, hence a_1 is the only major audio component

its impact on the audio image, but eliminating it completely would require some synchronization scheme that does not pause the audio. Video jitter between frames would not be noticeable, but this does not seem achievable with the current synchronization scheme and its binding to MPV, so the synchronization results in different frames being displayed by different players. 100ms is a little more than two frames, which is enough to notice, but the fact that the system simply seeks to a future frame and waits for the rest makes it much less visible than if it would have faded to black as in a scene shift. Large synchronization windows ($>$ 300ms) has a kind of funny spoiling effect on video playback, because the synchronizing player jumps ahead of the other players and waits until they catch up. So if it happens to be movement between the edges at the time of synchronization, a puzzle effect occurs, where the synchronizing player shows a static frame, hence misaligned edges between the players, and as the other players catch up with it, the edges align more and more until they suddenly match and the synchronizing player resumes playback.

The impact both audio and video jitter has on the multimedia playback depends on the content that is playing at the time the jitter occurs. The most obvious example would be that audio jitter can not be noticed if it happens during a silent part of the audio track. A similar effect happens for static video sequences, so if a player displaying some static background happens to execute a synchronization procedure this is not noticeable, whilst a panning shot (i.e. background moves) tends to seem very jittery with the slightest of frame differences. Since the players are free to decide when to do the actual synchronization procedures, they could in theory decide to do them such that their noticeable effect are minimized, for instance during silent periods or static sequences, but this kind of optimization is left as future work.

5.1.1.1 Window size hysteresis

Having the synchronization window dynamically change it size cause some problems when there are large fluctuations in the time it takes to synchronize a player. This usually happens when the data is not buffered (e.g. during initialization and after extensive seeks), since this adds I/O delay to the operation. Then a missed deadline results in a subsequent seek, where new data needs to be buffered, which results in a domino effect until the synchronization window is large enough to accommodate the I/O delay. Unfortunately, I/O delay is not representative of regular synchronization events, so afterwards the window is much larger than it needs to be, and, as the synchronization window grows exponentially, it takes the player large amounts of synchronization events to stabilize the window size. In the prototype this have been counteracted by manually setting the synchronization window after initialization have completed, but that is not a viable option in the long run. A possible solution would be to have different windows for synchronization (presumably buffered) and initialization / extensive seeks (presumably unbuffered), however this requires information about whether or not the data is/was buffered, or some assumption about the time it takes to complete buffered operations in order to classify the operations. Another possibility could be to add some sort of low-pass filter to lasting window size changes, such as having it temporarily grow exponentially until the current delay is accommodated, but afterwards only grow the lasting window once or, as a compromise, some function of the number of successive times temporary exponential growth has been needed.

5.1.2 Alternative synchronization schemes

Dropping frames at given intervals would be another procedure for doing video synchronization, and it is probably doable with MPV as the backend, but doing so would result in players having a relative drift of up to an entire frame before synchronization occurs, which would make the video seem laggy during scene shifts. MPV's speed control does not seem to be dropping frames, so it could in theory be used to do video synchronization, but it might lack the resolution to do accurate tuning. Minor changes to playback speed does not seem to cause noticeable audio jitter, so it might also be used for audio synchronization, but this has not been explored further, and in any case speed changes might also cause unintended audio effects such as changes in pitch[2]. Volume fading might also be considered a possible method of reducing noticeable jitter during audio synchronization.

Many of the schemes presented above probably require some sort of feedback to what MPV is actually playing. Querying current position takes 5 to 10ms, and without determining where within this period the

[2]MPV supports pitch correction to counteract this particular effect [5, options.rst#audio], but it is just an example

actual measurement was taken it adds an uncertainty making it unsuitable for the task. MPV supports dumping statistics to a file. These statistics includes periodic[3] measurement about synchronization delay, and this file provides a timestamp with the measurement, so it avoids the uncertainty yielded by the querying position. This statistics output might be used to interpolate an accurate position (assuming neglectable drift since last measurement) if some thread parses the statistics in the background, or a similar feature might be available, or at least be made available if needed, through the API.

5.2 Media sources

Having the files on local storage results in an aggregated I/O bandwidth much greater than the network bandwidth shared between the nodes and that to the outside world, so this system is able to play larger files from locale storage or its NFS cache than what it could play via the network. The current system simply fetches one file per player, so given the 1Gb/s network bandwidth it yields a media bitrate for network sharing of about 25Mb/s[4], and though media files with such bitrate do exists it is usually in their uncompressed format, so for the actual streaming part most media files do not exhaust this available bandwidth. However, MPV seems to be fairly greedy when it comes to buffering from file paths (NFS in this scenario) and HTTP/-S resources, so for larger files the cluster easily saturates individual nodes link bandwidth (assuming source with equal or greater bandwidth) during the time all the players fill their buffers, but this usually levels out to a little more than the aggregated bitrate during regular playback (i.e. filled buffers).

This eager buffering cause problems when playing media files with an aggregate bitrate bordering on the network bandwidth, because saturating the network reduces goodput[5], because TCP needs to retransmit packages when they are not delivered. Starting to play such large files via the network causes saturation, which results in missed deadlines as no players manage to buffer enough data for stable playback, so the players never achieve proper synchronization and all their playback remains jittery. Pausing the players at the beginning, and after any major seeks, then awaiting initial buffering before resuming playback of the files in question seems to work, but, as that is only avoiding the problem, not solving it, this is only achievable manually, so it is left as a feature for expert users.

The system requires some metadata, such as resolution and available audio channels, about the source in order to correctly instantiate the viewers. If the user happened to be the source of the media it could simply determine this metadata from its copy of the media, but as this system supports arbitrary URIs the viewers can fetch the data independently of the user, so there exists scenarios where the user never has a copy of the media source. Some services, such as Wikipedia's image viewer, supports arbitrary scaling the actual source by modifying parameters in the URI, but this does not scale very well when it would require reencoding the media. If the media is accessible to the user it could fetch its headers to determine the metadata, but there are scenarios where it will only be accessible to the players, of which the most obvious would be files in the players local file system. As with the URI of the media source, how this information is determined have been left out of the scope of this project, so the implementation either requires it to be statically configured or available from a source that is accessible to the user, so that it can fetch the metadata.

5.2.1 Caching

Having each player fetch the same file is kind of a wasteful, so a trivial improvement would be to cache it along the way. The effect is most noticeable if the file is to be transferred via a network link with limited and/or costly bandwidth. In such a case, having a local server fetching the file, then having the players fetching the files to from this server, minimizes the amount of traffic transferred through the expensive network link. Some scenarios could enable broad-/multicasting the cached data to the players, but this would require a media player implemented such that it can play data from a caching server that takes the initiative, which would require some sort of streaming protocol instead of having them play regular files.

[3] Measurements are taken at a 25Hz rate, so presumably once every frame

[4] 25Mb/s $\approx \frac{1000}{28+13} = \frac{v_{\text{Network}}}{n_{\text{Display}}+n_{\text{Audio}}}$ for 5.0 surround sound

[5] Goodput := $\frac{\text{Payload}}{\text{Total}} \times$ Throughput, https://en.wikipedia.org/wiki/Goodput

On the other hand, if this server scatters the file by unicasting it to every player, its link could act as a bottleneck, because everybody still needs to get the file from this particular server. In such a scenario the system as a whole could exploit a greater fraction of the backbone switching capacity by organizing a hierarchical caching scheme where one fetches the file from the expensive source, some given fraction of the players fetch it from this server and the rest fetch it from these intermediary caching servers. This would distribute the scattering responsibility over a greater number of links, as to give a greater aggregated bandwidth, at the cost of some minuscule extra latency caused by having the data propagate the caching levels, but that should be unproblematic in local network for non-live transmissions.

5.2.2 Track separation

When the system use different players for video and audio transferring the entire file to each player is wasteful as each player only use the parts that it is playing. Muxing[6] a multimedia file into one file per track, then having the *visuals* (i.e. either video or a subset of audio channels) play different URIs, is a possible optimization. The system supports per *visual* URIs, so doing this optimization manually is supported in the current system. In theory the system could support predefined track suffixes used by files that have been manually muxed to single tracks, so that players could use track specialization if they exists or fall back to the entire file for unoptimized playback, but as it still leaves the manual muxing part, such an optimization have not been explored.

Audio tracks contain every channel, so in order to split these into smaller chunks, it would require reencoding the audio tracks, which is possible but much more expensive then simply remuxing the container. Encoding one channel per file would require on-the-fly merging during playback of a subset of the channels, which can cause more synchronization problems. Alternatively, the files could be encoded to include every needed subset of channels, maybe even resampled down to mono at the same time, but this would require a priori information of which channels are to be played together. In any case, the entire audio track tends to be order of magnitudes smaller than the video track, so trying to split an audio track probably causes more hassle than it is worth.

Reencoding the video tracks into differently cropped subvideos is also a possibility, and in most cases it would probably yield greater savings than splitting the audio tracks. Multimedia content already need some sort of composite *visual* in order to support both audio and video, so adding relatively positioned video players using the same playback controller would be a trivial task. A 4K video can be split evenly among 4 separate 1080p subvideos, whose aggregated file sizes would be more or less that of the original file, hence transferring all of them to a single display would require about the same amount of bandwidth as the original 4K video. This results in the same amount of bandwidth being required for the display where all of these subvideos intersects, half the bandwidth for displays on the lines where only two of them intersects, and a quarter for the rest. It is theoretically possible to organize displays and the *visuals* such that the intersection edges between the subvideos is aligned to edges between displays, so that every display only shows one of the subvideos, hence every impacted display only require a quarter of the bandwidth, but this system assume arbitrarily positioned content, so this scenario will not be explored any further. Stitching the playback together would require more players on the displays where the subvideos intersect, and, as this is a hypothetical optimization that is yet to be tested, it is unknown how the processing power of playing a 4K video relates to that of playing its 4 separate 1080p subvideos.

Splitting the original multimedia file fits well with caching, because a premise for saving bandwidth with splitting is that every player does not receive the original file. Remuxing the container, such that the audio/video tracks can be transferred respectively to audio/video players, is trivial to achieve in real-time, but reencoding, on the other hand, impose substantial latency and requires extensive processing power. Splitting can also benefit from doing hierarchical caching, for instance the first node can remux the streams and serve as the primary source for audio tracks, whilst the video track is served by some intermediates, which also reencode their copy of the video, such that they all serve separate subvideos, instead of the original video, in order to distribute both the reencoding load and the bandwidth consumption.

[6] Combining separate multimedia tracks: https://en.wikipedia.org/wiki/Multiplexing

5.2.3 Streams

Any sort of automatic caching and/or track synchronization tend to fairly quickly optimize away the file concept in favor of streams. It should be fairly easy to extend the cross-player synchronization scheme to include streaming protocols that support some method of relating time-wise to the start of its source, such as Normal Play Time in RTSP [10, 3.6]. Having a primary cacher/separator would also yield a perfect reference point to obtain some initial position as opposed to today's prototype, which simply assumes that it should start at the beginning. In Display Cloud these cachers/separators, serving either files or streams, could have been implemented as *visual* sources, but that has not been a priority in this project, as files are good enough as a proof of concept.

5.3 Logical mapping speakers and audio sources

Display Wall Lab's actual 7.1 surround sound system consist of speakers along the ceiling, of which the front (including center) channels are placed along the top edge of the display wall, and the rest are placed outwards (from the display walls perspective) into the room. The system also has a subwoofer placed behind the canvas, but spatially differentiating LFE and other audio channels have not been solved by this project, so LFE channels are rather sent to every regular speaker, and the subwoofer is therefore left unused. The surround/side channels of the system are not used, since most sources do not have more than 5.1 audio tracks, and it would require an asymmetrical mapping of the virtual speakers.

In the Display Cloud the display wall is mapped as a 7×4 display grid with a DPI close to that of the real world in the Display Cloud's virtual environment. The only obvious sound image representation corresponding to this setup is to have the front channels mapped along the top edge of the virtual display, since these are the only speakers that are physically close to the display, and so that *visuals* physically close to these speakers are also virtually close to them. Only mapping the front channels leaves a *stereo+* image, where *visuals* moving downwards the display gets further away from the virtual speakers, and in order to utilize the surround sound system the bottom edge have been mapped to the rear speakers. The resulting sound image represent the display wall tilted upwards and stretched to the back of the room.

Different multimedia system arrangements would yield different virtual mappings, and mapping the one dimensional sound space into a multidimensional space is not a trivial task. Having the front speakers by the floor requires a virtual mapping that is pretty much a horizontal mirror image of the current virtual mapping, and, if the front left/right speakers were located beside the display wall in ear height, their virtual position should be more or less in the halfway between the top and bottom of the display wall, which would make the virtual space even more asymmetrical and pretty much incompatible with regular surround sound systems, since the front left/right speakers are virtually closer to the rear speakers than the center speaker. In any case, as pointed out before, this is a computer science project, so getting perfect acoustics is not prioritized, and as such this problem will not be discussed any further.

The media sources also need some method of positioning their audio channels in this virtual environment. The simplest alternative is to mix all channels down to mono and position the audio source in its origin. *Stereo* is also trivial as the left/right speakers only yield a one-dimensional sound space, so the channels are simply positioned at the same height (e.g. center) on their respective edge. A surround source, on the other hand, needs to have its channels positioned in accordance with how the speakers are related to the displays in the multimedia system it is to be played on. For the virtual mapping of the display wall, this would be front and center channels along the top edge and rears in the bottom corners, or rear/side in the bottom corners, but offset a bit along respectively the bottom/side edge for 7.1 sources. The problem with having a mapping that relates to the arrangement of the multimedia system it uses, is that this source should be movable to other multimedia systems, so if they have a mirrored speaker arrangement its channels will be perceived as mirrored on that system. The possible workarounds this project has produced is either to mix every surround source down to *stereo*, or to ignore the problem, and only support moving sources among multimedia systems with compatible speaker arrangements, so finding a more general solution is left as future work.

The audio positioning system can exploit both the loudness and time difference cues humans use to position, by respectively decreasing the volume of, and delaying the signal from audio sources virtually far from a given speaker. Exploiting the time difference requires a much greater control of the sound stage than what this system is designed for, and what has been available during this project. The

synchronization slack (5ms) cause relative channel delay of about a third of the size of the display wall, so the precision that is achievable using delay in such a system is limited and very variable with respect to synchronization. This kind of localization is currently implemented as a proof-of-concept by delaying the playback using this system's internal synchronization scheme, but this requires one channel per player, and requires explicit synchronization (i.e. observable side-effect) for updating the delay. An alternative approach would be to use MPV's `delay` filter designed to introduce this kind of delay, but it was never tested, so how noticeable changing its values are to the playback, is still unknown. By requiring synchronization to update the position, it would result in jittery movement, so instead the delay was meant to be updated implicitly during the next synchronization event, but, as this could cause loudness and time difference cues to disagree about the position, stereophonic delays have been disabled entirely.

5.4 Scalability

The locality of a *visual* to its display, is much greater than the locality of how an audio source impacts the sound system, so, in the case of the display, wall this results in five virtual speakers with a linear range covering more or less[7] the entire wall, and, in order to have its origin on the edge, a mirrored span outside the wall, but, as the virtual environment spans indefinitely, the system can be organized so that it does not cause any problematic interference. Having a 5.0 source arbitrarily placed on this wall requires $5 \times 5 = 25$ players to represent its intended sound image with respect to its virtual position. Whilst video is local to whichever subset of displays it occupies, the sound from every *visual* with audio tracks visible on the wall will require players for more or less every speaker, which, if surround sources is assumed, yields 25 players per *visual*. As each of these players needs to be synchronized this clearly yields a scalability problem.

As described in Section 5.2 these players fetch the entire multimedia file from the source, so it creates tremendous I/O load for the affected computers. Each audio player requires much less CPU and memory resources than its video complement, but having dozens per speaker quickly adds up to a considerable load. In the beginning a bug[8] caused the audio player to actually decode the video track, and in these cases a single 4K video with surround sound exhausted the computation resources of computers handling both audio and video playback. This got significantly better when they stopped decoding video, but it is still a noticeable load difference among the computers doing both video and audio, in comparison to those doing only video for the playback of a single *visual*, hence there will come a point where enough *visuals* will cause load problems.

To achieve any analog, as oppose to digital (i.e. play channel only from closest speaker), stereophonic effect, the sound must be generated by multiple speakers. That does not mean it has to be played by every speaker, but any interpolation to non-speaker positions will require at least 2 speakers per dimension that is offset from actual speakers. However, having multiple synchronized players of the same media source on the same computer is kind of a design flaw, as having one player, with or without video mixing, arbitrarily volumed input channels to output channels should be doable. This couples video and audio tracks together, but yields a less expressive system, so it is left as a possible optimization for when having tremendous amount of audio players becomes a problem.

[7] Speaker range span up to, but not including, the opposite edge, so a channel on the edge of a *visual* spanning exactly (or outside) the entire screen in the x and/or y dimension does not impact the opposite speaker in that/those dimension/-s

[8] MPV's scripting interface did not support simply disabling video so the manual suggested disabling video output (i.e. `--vo=null`) [5, man/options.rst#video], but it turns out its still decodes it, so disable the track instead (i.e. `--vid=no`)

Chapter 6

Conclusion

This report has presented the design of a distributed audio system capable of logically mapping multimedia sources arbitrarily onto a virtually mapped surround sound system. A working prototype has been developed and used to coordinate the distributed playback of videos with 5.1 surround sound, on a 28 computer 7×4 display wall connected to a 5.0 surround sound system shared amongst 3 of these computers. The system supports arbitrarily positioning the multimedia playback within the coordinated displays and having its position on the display being reflected in the room's sound image.

The prototype offers controllable trade-off between synchronization slack and the period between noticeable synchronization events. Its playback has good enough synchronization to avoid any nauseating effects, and it has been used to have the Display Wall Lab act as a glorified projecting system, with 5.0 surround sound for doing full-length movie viewings. Both the design and implementation is immensely influenced by Lars Tiede's Display Wall[11], but this was planned from the beginning, and it has the added benefit of adding useful features that can be incorporated into his system. The synchronization system designed by this project can be extended to synchronizing arbitrary information, so future projects may include features such as distributed subtitles and coordinated presenter tools.

Neither the design nor the prototype are perfect, but that is not to be expected for such a small project. This report discuss the capabilities and shortcomings of the system, and tries to compare it to alternative, but hypothetical, designs and implementations. It explains why plainly depending on clock synchronization among the affected computers is not enough to achieve synchronous multimedia playback, and presents an implementation of, and demonstrates the usefulness of the Media State Vector introduced by [1] and being actively developed into a W3C standard[1] for cross-device synchronization. Several optimization schemes, including, but not limited to, caching and multiplexed audio players, have been introduced, in order to achieve further scaling of the system.. Lack of time and resources has left these optimizations unexplored, but it should be mentioned that today's system is already capable of playing videos with 4K resolution and 60FPS, coupled with 5.0 surround sound. More complicated audio arrangements are common, but 4K video resolution is more or less today's high-end market. However, that being said, the future will surely introduce higher quality movie experiences that will require more powerful playback systems.

[1] W3C Multi-Device Timing Community Group: https://www.w3.org/community/webtiming/

Bibliography

[1] Ingar M Arntzen, Njål T Borch, and Christopher P Needham: *The media state vector: A unifying concept for multi-device media navigation.* In *Proceedings of the 5th Workshop on Mobile Video (MoVid'13)*, pages 61–66. ACM, 2013.

[2] Ryan Barrett: *Synchronizing mp3 playback*, 2002. https://snarfed.org/synchronizing_mp3_playback.

[3] D. Borman *et al.*: *TCP Extensions for High Performance.* https://tools.ietf.org/html/rfc7323.

[4] Edoardo Daelli and Michael J. Kopps: *Media playback synchronization in xbmc*, 2010. http://cs.uccs.edu/~cs525/studentproj/projS2010/edaelli/doc/ReportFinal.docx.

[5] mpv-player *et al.*: *MPV reference manual*, 2015. https://github.com/mpv-player/mpv/blob/0262295c4b6821d92770db0ad5c152fb6b04336b/DOCS/man/.

[6] kanaka: *Linux: Screen desktop video capture over network, and vnc framerate*, 2010. https://stackoverflow.com/a/4317412.

[7] Sebastian Merchel and Stephan Groth: *Analysis and implementation of a stereophonic play back system for adjusting the "sweet spot" to the listener's position.* In *Audio Engineering Society Convention 126.* Audio Engineering Society, 2009.

[8] Merriam-Webster: *Stere- [def. 2b]*, n.d. http://www.merriam-webster.com/dictionary/stere-.

[9] John Postel(ed.): *Transmission Control Protocol.* https://tools.ietf.org/html/rfc793. Last visited: 2013-10-17.

[10] H. Schulzrinne, A. Rao, and R. Lanphier: *Real Time Streaming Protocol (RTSP).* https://tools.ietf.org/html/rfc2326. Last visited: 2015-04-10.

[11] Lars Tiede, John Markus Bjørndalen, and Otto J Anshus: *Cloud displays for mobile users in a display cloud.* In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, pages 12–18. ACM, 2013.

[12] University of Southampton — Institute of Sound and Vibration Research: *Virtual Acoustics and Audio Engineering*, 2012. http://resource.isvr.soton.ac.uk/FDAG/VAP.