Design and Development of a Multi-Zone Audio System

Cătălin Bălan, Simon Dibbern

June, 2016

Supervisor: Michael Boelstoft Holte

Abstract

Home audio systems are constantly improving and gaining new capabilities with the advent of the Internet of Things. New Multi-Zone Audio Systems, which allow users to wirelessly play music anywhere in a house, are becoming common place. This thesis explores the most commonly available setups and proposes a system designed to empower the user's control over audio. A prototype for a peer-to-peer audio signal transfer network, as well as a distributed user interface are presented. The development challenges, implementation details as well as performance characteristics are analysed. Finally, the overall design of the Multi-Zone Audio System is assessed while establishing the requirements for the next phase in development.

Contents

1 Introduction

Audio systems have been a part of every day life for decades. They are an indispensable technology that comes in various shapes, sizes and capabilities, from portable headphones to high end studio equipment. However, most audio systems act as simple passive components that either play or record sound, with little control over any parameters beyond the quality of the sound. With the advent of the Internet of Things, we believe that sound systems in consumer households will become increasingly complex, with several interconnected speakers and microphones situated around a single home. As complex setups become more common, it is critical that users are in control and feel empowered to use them, rather than intimidated.

One main trend in the development of complex audio systems is Multi-Zone setups. These systems allow users to place speakers in different configurations in different rooms of a house. This report aims to explore the state of the art of such systems and identify where improvements can be made. Furthermore, this report presents the development of one such system designed to overcome the discovered limitations.

1.1 Target group

A multi-zone audio system is designed to fit in large homes and service multiple users simultaneously. Families are therefore the most appropriate target group. This group includes a wide range of age groups and cultures, we therefore limit that group for design purposes to families who wish to have more control over their sonic experiences. This group would have more demanding requirements when it comes to the features provided by the system and a higher expectation for quality.

1.2 Goals

A series of goals were established in order to create a feasible prototype and validate it. These goals were selected based on reviewing the available solutions presented in Section 1.4 (State of the art) and on a series of open ended interviews conducted with eight participants. Five of these participants identified as audio enthusiasts and either produced music in some form or had a good understanding of audio systems in general. A more detailed description about these interviews can be found in Appendix A. The results of these interviews presented us with the following set of criteria valuable to our users about the multi zone audio system:

1. It should be easy to configure and set up;
2. It should allow playback from any source (phones, computers, old vinyl players);
3. It should allow playback from multiple sources simultaneously (as a common use case games and music were mentioned);
4. There should be some form of account management so that people in the house do not accidentally change someone else's music;
5. There should be a way to control the quality of the sound, via both equalization and bass/treble controls;

The literature and product review, however, provided another set technical requirements for the system:

6. The audio delay between two different speakers should be maintained within 25ms at most (Haas, 1972);
7. The audio playback on different speakers will drift over time and should be resynchronised constantly (Sommer & Wattenhofer, 2009; Lamport & Melliar-Smith, 1985);
8. The audio delay between the playback source and speakers should be kept low if gaming is a requirement;

1.3 Use case scenario

In order to further clarify the intended uses and individual parts that make up such system, the following use case scenario is described.

The system is set up in a typical home. A 5.1 surround speaker system is located in the living room and intended to be used with a TV. Furthermore, a set of stereo speakers is located in the kitchen for playback of music from a mobile device or for listening to radio. Equally, the children's bedroom is equipped with a stereo system for a gaming console and music playback from mobile devices. All speakers are connected to the home LAN network, either through Ethernet cables or WiFi.

All ten speakers can be assigned to three separate *zones* representing their physical position in the house. Now, the user can map the audio output of any capable device to one or multiple *zones*. For instance, during the day, the music being played from a mobile device is routed to both the speakers in the living room and the kitchen simultaneously. In the evening, the users turn off the music in the living room and use the speakers to playback audio from the TV. Finally, on special occasions, when children have guests over, they decide to play music in the entire house.

These changes are all performed through a simple *control interface* on any mobile device or desktop in the house.

1.4 State of the art

Sound systems which support similar user interactions have been commercially available for several years now. This section presents a review of some of these systems in order to understand how they function, what they are capable of as well as their limitations.

1.4.1 Control4

The manufacturer Control4 serves a wide array of home automation products (Control4 Corporation, 2016). Their solutions are cooperating to form a smart home experience. Products range from home entertainment to security devices. At the core of the system is a central controller unit. Furthermore, the eco-system consists of *Zone-Amplifiers*. These provide a amplified stereo-only audio signal ready to be fed to speakers. These amplifiers power either 4 or 8 zones at once. Hence, a setup does require a significant amount if wiring.

Additionally, they provide a *Speaker Point*. These act as a wireless alternative to the amplifiers. They have a single output (stereo), but crucially, a analogue stereo input. The intended use is to feed a audio-signal from a third-party product, such as a vinyl player or TV, into the system. Similarly, the *Wireless Music Bridge* allows users to stream audio into the system using Bluetooth or Airplay. Whilst covering a wide set of features, the products are designed to be installed by a professional and form a static solution.

Control4 is designed to be a complete smart home solution and despite its impressive features, it is not an accessible option for people who just wish to enhance their audio experience. In contrast to this design, audio control can be a stand alone solution and can be easily integrated with other smart home technologies.

1.4.2 Sonos

Sonos offers a variety of speakers with integrated wireless and multi-zone capabilities aimed for music listening (SONOS, Inc, 2016). The wireless speakers interconnect to form a network and can be grouped to form zones. The exact architecture is not transparent. The system gets audio-input primarily from its dedicated player with an integrated online streaming service. This allows users to collaboratively decide on exactly what is being played and, instead of streaming the audio from the device the interface is used on,

the speaker itself streams the audio from the online service. Additional products, such as the *Sonos CONNECT* do have an audio input, allowing to stream alternative audio signals into the system.

Sonos is one of the few multi-zone audio systems we had direct access to during development. In our tests with the Sonos speakers we observed the system locks users down and forces them to use the Sonos app to play any sound. It is impossible to play a movie on a laptop and use Sonos speakers for audio. Furthermore, by analysing network traffic we observed that streamed data is sent to only one of the speakers directly and is presumably distributed to the other speakers in a later step.

1.4.3 Samsung

The central device in Samsung's approach to multi-zone audio is their hub.

> "The Hub [..] works with the app to allow you to connect multiple speakers into a multi room system. By simply plugging the hub into your router and connecting via the app, you can connect, group & control your speakers [..]." (SAMSUNG, 2013).

The wireless speakers in their product range only work in the traditional paradigm of a direct connection between the playback device and the speaker. The described hub is required to form a multi-zone system.

1.5 Theoretical framework

This secion presents the main theoretical framework required for the design and development of the prototype.

1.5.1 Audio

> "Sampled digital audio (or simply digital audio) consists of streams of audio data that represent the amplitude of sound waves at discrete moments in time" (Burg, Romney, & Schwartz, 2012)

The fundamental element of a digital audio stream is a sample. A sample represents the amplitude of the sound wave at a specific point in time. As this value is stored digitally, it takes up a certain amount of storage space. This amount is the *bit-depth* (or simply *depth*), how many bits there are per sample. The amount of samples the audio stream

puts through over time is measured in Hertz and represents the *sample rate*, samples per second. The format for basic, uncompressed audio data is Pulse Code Modulation (*PCM*). The audio data is communicated as binary data, representing electronic *pulses* (Burg et al., 2012).

1.5.2 Network architecture

A core property of multi-zone audio systems is that they are required to play back (and optionally record) audio streams in multiple locations on multiple speakers. Handling all of the necessary mixing on one system and playing back the audio directly in each zone is not an option for several reasons:

1. It would require a prohibitive number of audio cards as they typically have a limited number of outputs.
2. It would require connecting wires throughout the entire home back to the centralised control.

In order to work around these impediments, the multi-zone audio system should be composed of a network of several nodes designed to play back music in unison. This section presents the reviewed theory required to design and implement this network.

A distributed system is "a collection of independent computers that appears to users as a single coherent system" (Tanenbaum & van Steen, 2007, p. 2). There are several possible architectures to structure such a network.

The simplest form of organizing a network is a *layered* architecture. A common example would consist of a website which has an interface layer, a database layer and a processing layer that directs business logic in between the other two. These layers can be split across multiple machines, such that users only have direct access to the interface, while processing and data access take place on two or more different machines in the network (Tanenbaum & van Steen, 2007). Furthermore, it is possible to have duplicate nodes within a layer to service multiple customers, as shown in Figure 1. This type of architecture organization is known as *vertical distribution* (Özsu & Valduriez, 1991).

These architectures are simple to understand and implement, and are efficient in regards to data transfer performance. Communication is well defined to only take place between layers and each node has full authority over its computation. Despite the simplicity, layered structures are typically rigid and difficult to scale for layers which handle data persistence (Özsu & Valduriez, 1991).

Another common approach is *horizontal distribution*. In this configuration, nodes are typically homogeneous and each operates on its own share of the complete data set. The

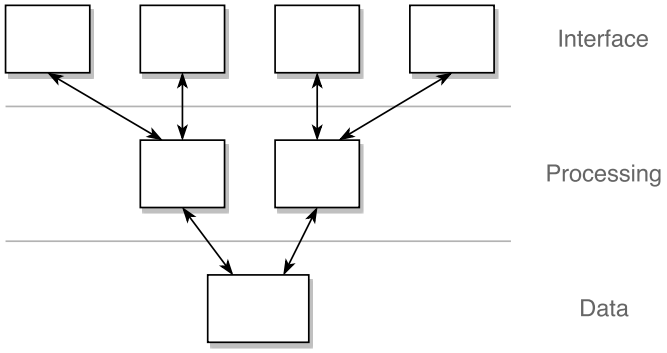

Figure 1: Layered network architecture with multiple nodes within a layer.

largest class of networks that support *horizontal distribution* are *peer-to-peer* networks. These networks build an *overlay network* which acts as a resource location service where queries can be made to find any particular node that either stores some data or performs a certain function (Aberer et al., 2005). This *overlay* is essentially a routing table.

Two major categories of *peer-to-peer* networks exist, *structured* and *unstructured* architectures (Lua, Crowcroft, Pias, Sharma, & Lim, 2005). *Structured peer-to-peer* designs construct the *overlay network* and the possible connections between the nodes following a deterministic process. This process is predominantly a *distributed hash table*. This class of algorithms aims to efficiently map the key of a data item to the identifier based on some distance metric (Tanenbaum & van Steen, 2007; Balakrishnan, Kaashoek, Karger, Morris, & Stoica, 2003).

On the other hand, *unstructured peer-to-peer* designs use randomized algorithms to produce an *overlay network*. In this class of networks, each node maintains a list of neighbours. This list of neighbours is known as a *partial view* of the system, and nodes regularly update this view. Locating a data item is done by broadcasting queries throughout the system (Tanenbaum & van Steen, 2007).

1.5.3 Consensus

Apart from addressing and identifying nodes, *peer-to-peer* networks present another set of problems not present in a *layered* architecture. Data has to be replicated (partially or fully) across the nodes and during normal operation, no conflicts can occur. This concept is known as *consensus* and requires a series of properties (Correia, Neves, & Veríssimo, 2006; Hadzilacos & Toueg, 1994).

1. **Validity.** Any value decided upon, is a value proposed to the system.
2. **Agreement.** No two processes decide differently.

3. **Termination.** Every correct process eventually decides on a value.

4. **Integrity.** Every process decides at most once.

A protocol that satisfies *consensus* will also satisfy the definition of a *distributed system* when communicating with any individual node, as any node would behave like the entire system. Protocols can achieve *consensus* by using a fault tolerant algorithm like *Paxos* (Lamport et al., 2001) where new values are proposed to other nodes and after a series of phases of the algorithm, the values get agreed upon by the system. Furthermore, *consensus* can only be achieved under at least one of these conditions (Turek & Shasha, 1992):

1. If the nodes within a system are synchronized and messages are delivered within a predetermined bounded time frame.
2. If messages can be guaranteed to be delivered in order and nodes can broadcast (send messages to all other nodes within an atomic operation).
3. If messages can be guaranteed to be delivered in order and the nodes are synchronized.

If a system can guarantee to reliably deliver messages in order, then that system supports *atomic broadcasts*. Note that to deliver a message means to actually process its contents, while broadcasting means sending the message to all other nodes. Delivery and broadcast can be separated in time and, in theory, a broadcast message is not necessarily also delivered. Formally, *atomic broadcasts* require the following properties (Hadzilacos & Toueg, 1994):

1. **Validity.** If a node broadcasts a message, then it eventually delivers that message.

2. **Agreement.** If a node delivers a message, then all nodes eventually deliver that message.

3. **Integrity.** Each process delivers a message at most once.

4. **Total Order** All processes deliver all messages in the same order.

Finally, it is proven that *consensus* and *atomic broadcasts* are equivalent problems in distributed systems (Hadzilacos & Toueg, 1994). An algorithm that is designed to solve one problem can be trivially converted to solve the other. Therefore, *atomic broadcasts* require the same conditions that *consensus* requires in order to be implemented.

In a synchronized system, where the clocks of all nodes are guaranteed to be within a bounded time difference, have a known drift rate and there is a known bounded limit for the transfer and processing time of a message, *timed reliable atomic broadcasts* can be achieved by delivering all messages at a fixed finite time after they are broadcast.

1.5.4 Continuous synchronization

Synchronization is important for both audio processing and *consensus* within a distributed systems. It is therefore important to discuss what synchronization is and how it can be achieved. A network is synchronous if every node has a *local clock* with a known bounded rate of drift from real time and there is a known bounded limit required to send, transmit, receive and process a message (Hadzilacos & Toueg, 1994).

It is important to know that *local clocks* do not measure real time directly, but still satisfy *monotonicity*: they never decrease and they never skip values. Furthermore, the clocks of two different nodes are known to be *logical clocks* if for every event e that causally precedes event f, e takes place at a time before f for each of the clocks (Hadzilacos & Toueg, 1994). Satisfying these two properties allows for a straight forward implementation of *atomic broadcasts* (Hadzilacos & Toueg, 1994).

Two clocks are synchronised if the values of their *local clocks* differ, in real time, by a known bounded constant. Many algorithms exist for synchronizing clocks over a network (Sommer & Wattenhofer, 2009; Lamport & Melliar-Smith, 1985; Mills, 1991, 2006), but they generally do not require to satisfy *monotonicity*. These algorithms work in phases to measure the difference between two *local clocks* and periodically set them to the same value. In order to achieve *monotonicity*, one can simply interpolate between the values set during synchronisation phases such that the *local clock* never skips (Sommer & Wattenhofer, 2009).

1.5.5 Distributed User Interfaces

The described system is dependent on some configuration taken care of by the user. Due to the distributed nature of the system and the multi-user environment, a Distributed User Interface (DUI) is proposed. DUIs, as described by Villanueva, Tesoriero, and Gallud (2013), are User Interfaces (UI) that can be shared across different devices.

DUIs can be seen in two broad forms: as multiple instances of UIs on different devices, each providing a part of the functionality to a single user. Or, alternatively, as multiple instances of the same UI running on different devices used by different users simultaneously. This approach is further described by Melchior, Mejías, Jaradin, Van Roy, and

Vanderdonckt (2013). Such DUIs allow great amounts of collaboration, but do require *consensus* on the state among the different instances. Solving *consensus* for the other nodes in the system would satisfy this requirement, if the interface only communicates with those and not directly to other instances of itself.

2 Methods

This section presents the methodology used to implement the application, as well as the techniques used to validate whenever the proposed goals were achieved.

2.1 Overall multi-zone system architecture

The final design of the prototype uses a hybrid approach to the network architecture. As there are expected to be relatively few zones in a single house (limited by the number of rooms) and relatively few speakers in a single zone (ranging from 2.0 stereo to 7.1 surround) it is feasible to assume that each node would be able to keep track of all other nodes in a system. It is also easy to observe that the entire system would have low amounts of persistent data to store (metadata about nearby processing nodes, metadata about playback and recording streams and configuration data such as equalizer and volume settings). We estimate the amount of data required to fully configure and describe a particular setup of a multi-zone audio system to be within the order of at most tens of kilobytes.

These two observations about distribution scale and storage requirements suggest the best and simplest approach for the network architecture is a fully replicated, unstructured and synchronised system where all nodes have complete knowledge about the entire system, and where all nodes can communicate to each other. This choice eliminates the requirement of implementing a DHT to identify each node and simplifies the replication algorithm. The only major requirement so that the system acts uniformly is to handle *consensus*.

Despite the low requirements needed to implement such a small distributed system to handle its configuration, the prototype is still required to transmit audio data to each node. At RAW 16bit PCM audio at $44100Hz$, this would amount to $\simeq 86.14Kb/s$ per channel. Assuming the extreme case of four simultaneous stereo and one 5.1 signal being played in the same zone, this would amount to a total of $\simeq 1.17Mb/s$ of uncompressed data to be distributed through the nodes. We consider this to be an upper bound as the final value can be brought down by compression ($50-60\%$ by using lossless FLAC compression) and as the original interviews suggest users would play at most three distinct streams within the

same zone, only one of which would be more than stereo.

It is not feasible to replicate the audio data fully between all the nodes of the system the same way configuration data is replicated. Therefore, the prototype would use a centralized/layered approach to transmitting audio data, as this method has the lowest amount of data transmission requirements and scales linearly with the number of nodes. Within each zone, a node would be designated as a leader. This leader would handle mixing of all inbound signals and then redistribute the needed signals to all the other nodes.

A final problem is providing audio data to the system. By using a centralized approach, audio sources cannot transmit data directly to any node in a zone, but instead have to first identify who the leader is and transmit data towards that particular node. This solution goes against the definition of a distributed system, as the entire network structure would no longer appear as a single entity to the external world. To solve this issue, we introduce virtual nodes that receive raw audio data and identify leaders and transmit data to them. These nodes would be implemented as dedicated audio players or virtual audio drivers that can interact with the multi-zone audio system.

For clarity, we shall refer to nodes which transmit audio data within the system as *relays* and virtual nodes which provide an interface for audio transmission to the system as *emitters*. It is important to note that this is a logical separation, not a physical one. A hardware component within the multi-zone audio system could have both *relay* and *emitter* software and provide both output and input. Figure 2 illustrates the proposed setup.

Relays are designed to be hardware independent and to support outputting audio on as many channels as their sound card supports. However, internal testing was carried out using various versions of Raspberry Pi[1]. Each of this versions is equipped with a single 3.5mm stereo output port, and thus the hardware design of the system associates one relay for every one or two physical speakers.

2.2 Relay development

The main focus of development, in terms of making the system functional, was on *relays*. They define the communication protocols, handle audio mixing as well as integrate a set of discovery subsystems that empower the prototype. In contrast to the relays, for testing purposes only a simple emitter was developed that could read WAV files and transmit them to a leader relay.

Both the relays and emitters were developed in C++ with strong emphasis on per-

[1]https://www.raspberrypi.org/products/ “Information about each of the official Raspberry Pi products”.

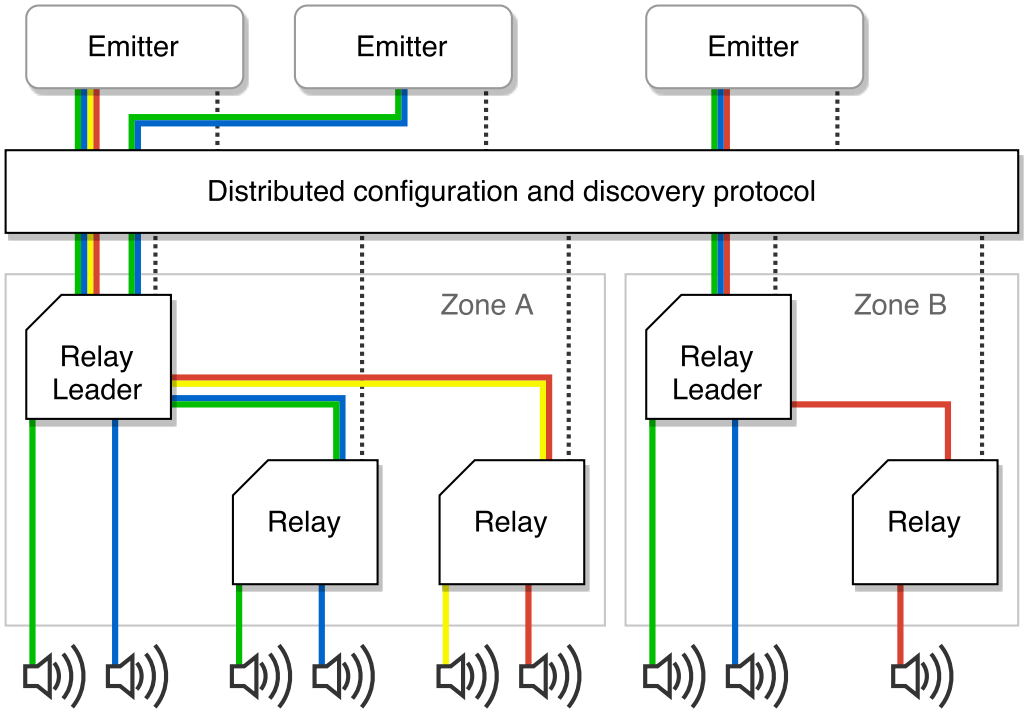

Figure 2: Overal system architecture. Emitters provide different channels of audio streams to zone leaders directly. Configuration is achieved through a distributed protocol.

formance as the final implementation should run on small computers and/or embedded systems. The code for both relays and emitters currently only runs on Linux, but the only platform specific component (audio playback) is well isolated to allow porting later on if the need arises. Furthermore, the only third party dependencies included in the implementation are *libsndfile*[2] for reading WAV files, *msntp*[3] for SNTP (Mills, 2006) support, *Alsa*[4], audio playback on Linux, and *mDNS* discovery, on Linux this is provided through Avahi[5].

2.2.1 Discovery and relay management

Relays use *mDNS*, a standard protocol provided by Apple for zeroconf network discovery on LAN. After the initial discovery, relays exchange information about their local state. Following this process, replication takes place so that new relays are brought up to date with the state of the entire system.

As clock synchronisation is important for audio playback and as the entire system is

[2]https://github.com/erikd/libsndfile “A C library for reading and writing sound files containing sampled audio data”.

[3]https://github.com/snarfed/libmsntp “A full-featured, compact, portable SNTP library”.

[4]http://alsa-project.org “Provides audio and MIDI functionality to the Linux operating system”.

[5]http://avahi.org “A system which facilitates service discovery on a local network”.

designed to run on LAN where upper bounds for delta time for packet transmissions is low, timed reliable atomic broadcasts are used to solve the problem of consensus throughout the system. Each message sent is accompanied by an ID and time stamp. All messages are delivered and processed in order at $time\_stamp + fixed\_delay$, as described by Hadzilacos and Toueg (1994).

The configuration of each relay is stored in a series of unordered sets describing relays, zones, connected emitters and active streams. Each of these four data structures is an atomic independent unity. Modifying a relay does not directly affect another entity in the data structure and operations which would be non atomic (such as moving a relay to a new zone) can be broken up into independent atomic commands which can be directly decided upon for *consensus* using *atomic broadcasts*.

The communication protocol used to achieve configuration and data transfer follows a custom data layout and is not final. The generic protocol follows a simple structure: 3 byte header containing *command ID* (1 byte) and *packet size* (2 bytes) followed by arbitrary data to be handled based on the received *command ID*. The data is serialized as JSON[6] for configuration commands exposed to emitters and the distributed user interface or as a custom opaque binary format when performance is important (transferring stream data, stream metadata or real time continuous debugging) or when the command is intended to be used only to transfer data between relays. Internal commands that are transmitted via atomic broadcasts contain an additional header with a 4 byte *sequence ID* and an 8 byte *timestamp* specified in microseconds.

To achieve synchronization, each relay broadcasts a synchronization phase every few minutes on its logical clock. During this phase, clock rate offsets are computed using the SNTP protocol between and all neighbours. The local logical clock of each node is then updated to match the average value read.

All the communication is implemented over TCP to ensure packets are delivered in the same order they are sent. The communication protocol that the interface uses is a connectionless request-response protocol. A request arrives at one of the relays, it is then broadcast to all other using *atomic broadcasts* to ensure consensus and after the request is delivered by the *atomic broadcast* it is processed and a response is sent back to the interface and the TCP connection is closed. In contrast, the audio data protocol is a connected one, where the TCP socket is maintained throughout a session.

A final separate connected command protocol exists for debugging. This is persisted throughout the applications lifetime and updates the state of each relay and logs to a debugging user interface at a rate of 20 updates/second. A more in depth description of

[6]http://json.org “A lightweight data-interchange format”.

the debugging interface can be found in Appendix B.

2.2.2 Audio processing loop

The first implementation of the prototype relied heavily on asynchronous threads to handle distinct phases of the audio processing. Playback and mixing would take place on the main thread, reading input streams would fill in the audio buffers on one thread per data source, forwarding data would be handled in a separate thread per target relay and processing configuration commands would be done on their own threads as well. This implementation proved to be difficult to synchronise and unpredictable. Furthermore, the testing hardware only has one execution core. As such the parallel threading model fell apart.

The current implementation does the main processing on the same thread sequentially and maintains a second thread open for the command protocol. This second thread is predominantly idle and blocks whilst listening for TCP connections.

The main audio thread runs at a frame rate defined by the size of the hardware audio buffer. *Alsa* submits fixed chunks of audio data to the sound card of a predefined size known as *periods*. Furthermore, *Alsa* provides a fixed buffer in software where multiple frames can be provided ahead of time. The default settings for all tested hardware used a 256 frame *period* and a 1024 frame buffer. In this setup at 44100Hz, audio would have a consistent delay of 6ms across all relays and the application would be limited to 172.2 frames/second but could occasionally drop safely to 43 frames/second.

For each audio frame, available stream data must be read into the mixing ring buffers, stream data must be forwarded to other relays if possible and 256 frames of audio signal must be produced for mixing. Additionally, this main loop also processed events produced by the command thread (such as responding to a mute command) or discovery events produced by the *mDNS* service. As such, these are the phases of the main loop, in order: poll and read stream sockets, poll and process events, forward data, mix, wait for buffer availability, write audio to hardware.

2.2.3 Mixing

Each relay implements a single mixer that stores and processes all audio data. Most other elements of the relay are build around the mixer.

The mixer contains a collection of data queues, implemented as ring buffers, to store all input streams arriving to the relay. Each ring buffer contains PCM data for a single mono channel, along with metadata about what that channel represents. This design was

chosen as it can handle audio data being delivered at different rates and caches up to 2 seconds of audio ahead of playback if possible. This allows both real time streams and cached streams to coexist, and can play individual streams at different delays. Furthermore, each stream to be played at the best possible latency and packet loss within a stream to not affect other signals. The final benefit of storing mono channels separately is that it allows mixed signals to be easily rerouted.

Three handles are used to manipulate the data for each ring buffer. The buffer, along with these handles are illustrated in Figure 3.

1. **The playback handle.** For every audio frame the mixer provides a fixed amount of samples for playback and advances the playback handle.

2. **The forward handle.** Every audio frame, the mixer provides as many samples as possible for each channel, starting from the forward handle.
   These samples are rerouted by the leader relay to other relays in the same zone.
   The forward handle is ignored for non-leader relays.

3. **The write handle.** Incoming audio data from a stream is written to this handle.

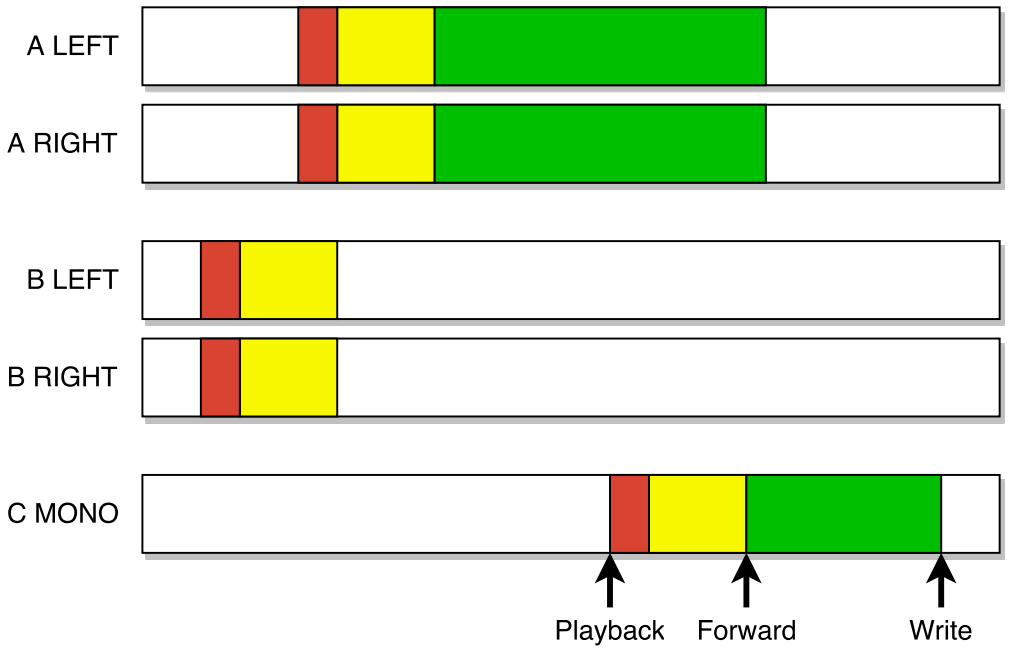

Figure 3: A typical buffer layout. Red: audio data needed for the next frame; yellow: data forwarded to other relays; green: data that cannot be mixed and forwarded yet.

The current version of the mixer only supports a single mixing pass, which is designed to be an equalization step. For testing purposes, the mixer only does signal amplification. Full equalization support is to be implemented at a later stage.

2.3 Interface development

Whilst some parts of the system are configured programmatically, the user has to be able to adjust certain properties in parallel. These could be properties that, by their nature, can not be estimated by the system or properties that the user might want to overwrite. Such distributed environment brings its own set of specific challenges and opportunities to any control interface:

1. The interface can not expect its local copy of the state to be valid. Consensus with the rest of the nodes in the system is required for any change to persist.
2. The interface is by nature a DUI. Any change to the properties of the system made by either an other instance of the control interface or programmatically by the other nodes should be reflected in the interface in a timely manner.

2.3.1 Actions

The interface empowers the user to manipulate the system to some degree. The primary actions a user can perform using the interface are:

Assign Emitters to Zones Each zone can play back one or multiple sources. The user is able to see and select these for each zone known to the system, as shown in Figure 4. This action is likely to be performed frequently and is therefore a central part of the interface.

Assign Relays to Zones This action is only required when setting up the system or physically relocating the speakers. The user is able to change which zone a relay and its speakers are assigned to, as show in Figure 5.

Map Speakers This covers mapping the speakers in a zone to the channels of the source signal as shown in Figure 6. Changing the physical location of the speakers might require the user to set which speaker is supposed to play which channel of the source signal. In the simplest case the user might want to swap left and right speakers in a stereo setup. However, in order to support more complex setups such as 5.1 surround, the interface visually models the physical setup, from where he can assign the individual speakers.

Other Furthermore, the interface provides the opportunity to alter settings such a volume and some metadata, such as names and description of the zones and speakers.

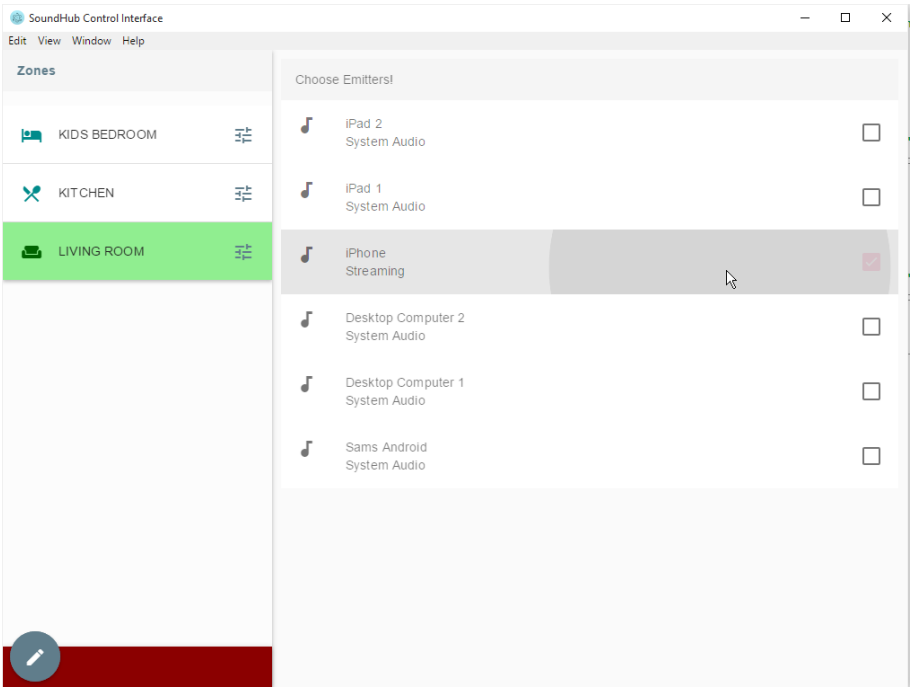

Figure 4: The user selects a zone from the left panel and is subsequently presented with a list of available emitters. He can select one or multiple.

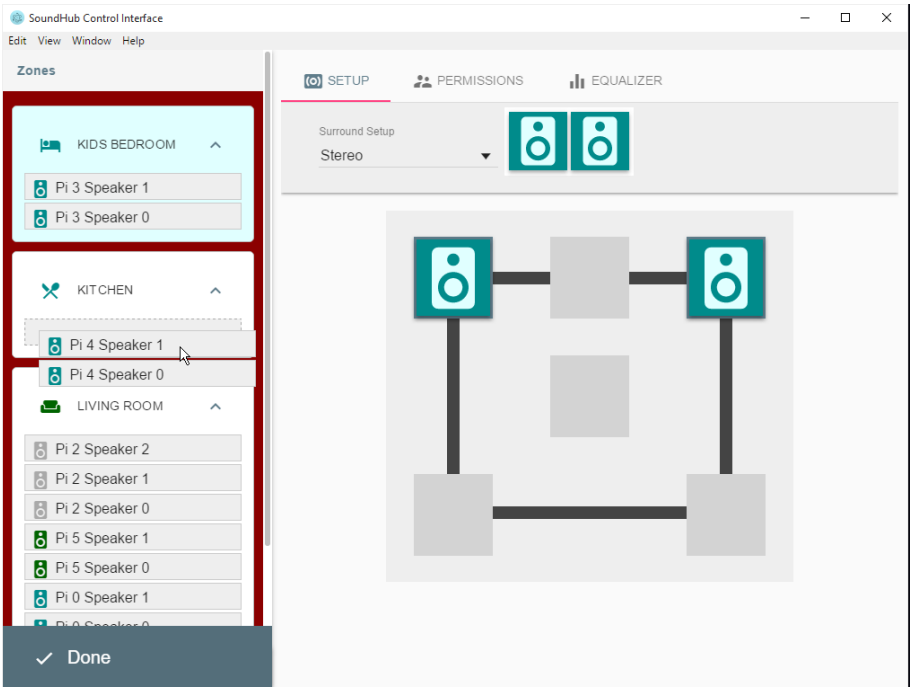

Figure 5: The user entered *edit mode*, where he can alter the zone configuration. He adds a relay with its two speakers to the zone *Kitchen*. It was previously assigned to the zone *Kids Bedroom* .

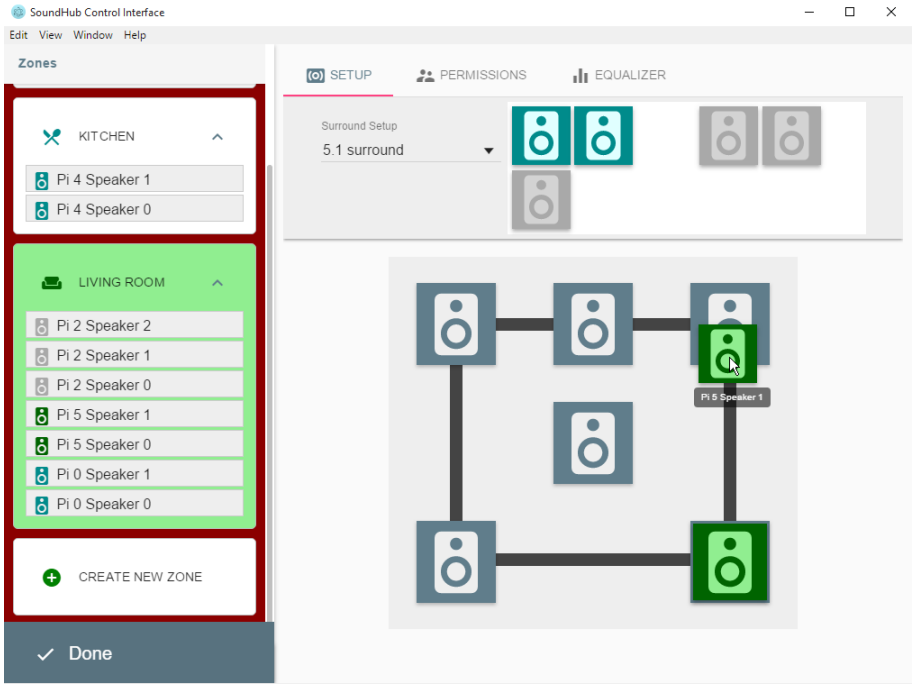

Figure 6: From the top panel the user chooses this zone to be a 5.1 surround setup. Notice that this zone has multiple relays. All unassigned speakers are listed in the top, from where they can be dragged into the configuration view and assigned a channel. Also, notice that the grey relay has 3 speakers, a rare, but supported case.

2.3.2 Implementation

The configuration interface is implemented using Electron[7], a framework for building cross platform desktop GUI applications. In its basic form, it provides a window in which the developer can implement a traditional web application using HTML, CSS and JavaScript on top of Node.js[8]. Furthermore Angular.js[9] is used to simplify the development. It provides some tools to extend the HTML vocabulary, primarily through data binding, for taking care of most DOM manipulations. Finally, in order to further simplify the implementation and in order to achieve a consistent design language throughout the application, Angular Material[10] is utilised. Angular Material is a implementation of Google's Material Design Specification[11] and provides us with a set of UI components.

The application follows the Model–View–ViewModel (MVVM) architectural pattern (Smith, 2009), as shown in Figure 7.

[7]http://electron.atom.io A framework for developing desktop applications.
[8]https://nodejs.org A JavaScript runtime, originally server-side only.
[9]https://angularjs.org A web application framework by Google.
[10]https://material.angularjs.org A UI framework.
[11]https://design.google.com A design language by Google.

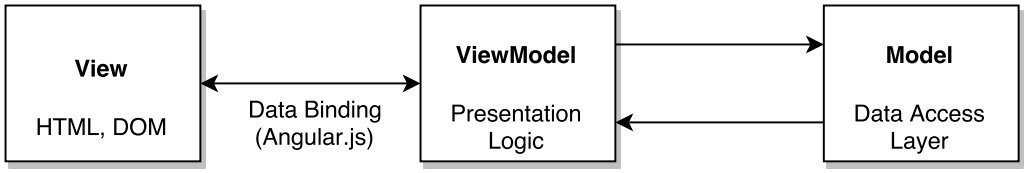

Figure 7: The Model–view–viewmodel. Angular.js provides the data-binding between the DOM and presentation logic.

Data Access Layer (DAL) All data comes from either a locally stored JSON[12] file or is received from the other nodes of the system. The JSON file stores a representation of the entire system and guarantees a complete global state and maps directly to the model, whilst the communication with the other nodes in the system only provides local states from the respective node or a complete state in a different representation. The data received from the other nodes always overwrites the data in the JSON file. Changes in the model are broadcast to other nodes. The DAL hides the complexity of discovery, network communication, file I/O and conflict solving to the rest of the modules, allowing for easy getting and setting of the system's properties. Network sockets are provided by the Node.js *net* module and for discovery Bonjour[13] is used.

Presentation Logic The presentation logic is divided into scopes. The major visual elements in the interface, the *zone view*, the *emitter view* and the *surround view* each represent a scope with its own controller. This pattern is provided by Angular.js, which takes care of the underlying logic. The logic for these scopes consist of:

1. Declaring the structures for data binding and getting data from the DAL.
2. Reacting on UI events and altered data from the binding.
3. Persist changes through the DAL.

2.4 Validation

The performance of the system was often measured during development to identify pain points within the system and to validate the choice of hardware. As such, reading the data stream, forwarding the data stream, mixing and playback of the data stream is measured, as well as overall time spent per frame and time spent waiting for hardware sound buffers to become available. These time measurements are made manually in code. All collected measurements are expressed in microseconds and have a granularity dependent on the hardware clock (in our observations, this granularity was under 3 microseconds for all

[12]http://json.org "A lightweight data-interchange format".
[13]https://github.com/watson/bonjour "A Bonjour/Zeroconf protocol implementation in JavaScript"

hardware used). Furthermore, the real time difference between local clocks on the relays is measured as part of the implemented synchronization algorithm.

The performance data is streamed from each relay to a debug interface using a secondary debugging protocol to along with other diagnostics. The data was then saved for further analysis.

Apart from performance metrics, sound quality was also relevant in order to evaluate the hardware. Although no formal measurement or analysis of this factor was made, our continuous use of the relays permits us to discuss this aspect.

3 Results

The collected profiling data shows promising results for the current version of the prototype. Presented here is a summary of the performance statistics collected from a *Raspberry Pi 1 B rev 2*, the lowest specification hardware available for testing. A *Raspberry Pi 2 B* and a Linux distribution running on a virtual machine were also used and show similar or better statistics, but the choice to focus on the lowest grade hardware is more representative for determining both minimum system requirements and optimization requirements.

Table 1 illustrates the measurements of a *leader* in a system configuration with one zone containing a total of two relays. As the main audio loop is determined by the hardware audio buffer, its duration can vary from platform to platform. The metrics presented in Table 1 are expressed in percentages of the duration of a frame.

Table 1: Average time spent per frame

Measurement	Time (mean)	Time (max spike)
Reading stream input	1.4%	5.5%
Forwarding stream	3.5%	12.9%
Playback	0.2%	0.3%
Other (config, clock sync...)	0.2%	0.5%
Idle	94.7%	N/A

The only measurement which differs for non-leader relays is the time spent forwarding stream data, which drops to 0 when no data needs to be forwarded (the buffer handle for forwarding is just moved forward). In larger configurations, again, forwarding is the only measurement that changes, increasing linearly with the number of nodes.

Moreover, we have encountered occasional spikes in data forward time which extend well beyond the duration of a single frame. These outliers have been excluded from the statistics presented above.

In terms of memory use the relay application used under 8 MB of RAM with input buffers preallocated for 32 distinct channels of audio input.

Finally, clock synchronisation was accurate to within 6ms and no audible difference between audio streams was perceived.

4 Discussion

This section presents our final assessment of the prototype and of the proposed goals.

4.1 System design review

Overall, the design of the hardware architecture was flexible enough during development to simulate any reasonable speaker configuration or use case. The model supports broadcasting multiple streams within a same zone, as well as broadcasting the same stream within multiple zones by temporarily linking zones together. The protocol can be further extended such that zone leaders forward only individual streams to other zones to avoid linking the zones together. This flexibility allows us to achieve goals 2 and 3 regarding playback from multiple sources and and simultaneous playback.

We have learned during development that this design is not only flexible in terms of what it allows users to do, but it is also versatile in terms of empowering the interface design. The model does not enforce any particular work flow in terms of defining how users configure the system, and this decision can be left entirely to the interface.

The current version of the network architecture stores individual relays in separate data structures from zones, where relays store the IDs of zones, but not the other way around as described in Section 2.2.1 (Discovery and relay management). Originally this was not the case, a tree structure was maintained storing a root node with the entire system with zone children and further down relay children. The original system, although holding the same information turned out to be more difficult to synchronise as commands that would modify zones were not atomic and caused relays to be changed. Transitioning to a linear data storage, where there is no dependency from zones to relays allowed all operations on the database to be atomic and made satisfying *consensus* and data replication an easier task.

The chosen design does have one major drawback, it requires a non-standard com-

ponent be implemented in order to communicate with the audio system. Emitters are required as an entry point for audio data. Admittedly, emitters can themselves provide a standard interface and they can be implemented as audio drivers for all platforms that need to use the system. Furthermore, hardware implementations of emitters can be provided to allow input from closed platforms (such as vinyl players or TVs). This drawback requires that users install additional software to playback audio and that development effort must be allocated further on to support a wide variety of platforms, unlike systems like Bluetooth which can provide an audio output via a standardised protocol. It is believed that users are willing to install these drivers as they also would be required to install the configuration interface in some form to be able to prepare the system for use. Regardless, this drawback is a problem that affects goal 1 in regards to ease of setting up.

The performance measurements taken indicate no immediate changes need to be done to the architectures, but they do show some concern in regards to the time it takes for a relay to forward audio data to its neighbours. The time spent writing data is very high, especially considering it more than doubles reading data from the same network. To solve this concern, the system could benefit from an optimization pass over data forwarding, as well as additional testing.

On certain occasions data forwarding can spike for almost half a second, much longer than the duration of a single frame. This known bug may be the cause of longer writes, and there is a high chance fixing it will bring the write time in line with the reads. If that assumption is wrong, the topology of data transmission can be restructured to have two layers of redirection of audio data. This would increase total delay but solve forwarding time for large zones.

The difference between logical clocks measured in real time was always under 6ms and thus achieved goal 6 by hitting our upper threshold of 25ms. Furthermore, the continuous resynchronisation achieved by repeating synchronisation phases and interpolating between the average clock rates worked as expected to achieve goal 7 and compensate for drift (Mills, 2006; Sommer & Wattenhofer, 2009). The synchronised logical clock worked correctly with *atomic broadcasts* and no problems in regards to *consensus* were observed (Hadzilacos & Toueg, 1994).

The ultimate goal of a complete multi-zone audio system is to manage all audio transmission requirements of a house. That includes input and recording sound, not just output. Although this report focuses almost exclusively on the playback of sound, it is trivial to model audio recordings such that the system does not need to be modified. An emitter would be placed at the entry point of the recording source and the data would be sent to a relay connected to an arbitrary audio processing unit, instead of a speaker. As this pro-

cess is modelled using the same components used for playback, all the findings regarding audio playback should also apply for recording.

4.2 Hardware choice

The performance measurements indicate that relays running on the lowest grade Raspberry Pi are capable of powering the network in regards to both processing and and playing back audio data transmitted over the network. As the system was idle 94% of the time, this further suggests the power requirements of relays are low and the main factor determining energy use would be audio playback itself. These findings suggest tests should be carried out with the relays running on less powerful embedded systems in order to drive down the cost of a finished product.

There is no reason to believe at this point that the forwarding audio issues are caused by the limitations of the tested hardware, as the same behaviour was observed during development when testing from a virtual machine with two 2.4 GHz cores and 2 gigabytes of RAM. Furthermore, we know it is possible for relays themselves to transmit the audio data across the network even in scenarios where more relays are used (SONOS, Inc, 2016). As a contingency plan, developing a hub dedicated to audio forwarding would be a solution to this problem (SAMSUNG, 2013; Control4 Corporation, 2016), yet it does not seem required.

A major concern with the tested hardware is the audio quality. Although not measured, a clearly audible noise floor was always present in the playback signal while the overall amplitude was also low. We were unable to find any software solutions to eliminate the noise floor (using different audio drivers and different playback techniques with those drivers in regards to buffering, interleaving data, sample rates, dithering…). The final hardware choice should be influenced primarily by the *digital to analogue converter* (DAC) as other performance criteria would be easy to satisfy.

A final consideration in regards to the hardware observed during testing was our reliance on 3.5mm audio jacks for output. We have encountered various TV sets or home cinema systems that use optical cables or custom connectors instead and could not be used directly with our prototyping hardware. This implies that either multiple types of hardware relays need to be developed to accommodate for different audio connections, or that relays would need to incorporate more ports.

4.3 Control Interface

Tesoriero, Lozano, Vanderdonckt, Gallud, and Konstan (2012) discuss the potential of DUI-specific features "polluting" the interface. A designer should ask the question of which elements in the interface are required to be seen by all users. Not in all cases it makes sense to let the other instances of the interface be barely a mirror image. The interface presented in this report circumvents such issues mostly by only synchronising the state of the underlying model, not the state of the view.

4.3.1 Collaboration

Whilst the interface manages to synchronise with all running instances of itself in the network, it fails to communicate any changes made explicitly to the user. According to Tesoriero et al. (2012), a change in the state of the system should have the interface attempt to inform the user about it, so he does not "miss it". This is to prevent a conflict between the actual state of the system and the perceived state. Possible ways to achieve this would be through visual animations or notification messages in the interface.

4.3.2 User Account Management

The interface allows any instance to change any parameter of the state. This amount of control comes at a cost. Users might be interested in not having others be able to control or even see the settings of a specific *zone* or *emitter*. For this to be an option, the system lacks two key attributes, as initially described in goal 4 (account management).

Firstly, it can not identify users. An account management system would have to be integrated. A hierarchy of users would be required in order to administrate the permissions, allowing some users to have control over the whole system including the permissions itself. Secondly, there is no concept of "ownership" in the current model. Assigning a set of users to each *zone* and *emitter* would be required in order to implement such feature.

Nevertheless, as development of the prototype advances, we are reevaluating the goals. Multiple users have specifically asked for this feature during the formative interviews, but we have seen no need for it during development. Account management clashed with the idea that users can configure the system from any device capable to run the interface and introduced extra steps in our UX design. Further UX testing should verify this requirement.

4.4 Bias

Unfortunately, this project only partially achieved its established goals. One of the goals have not yet been implemented: 5 (control over equalization). In this scenario, equalization settings are currently ignored. Furthermore, there is a high degree of uncertainty in regards to goal 8 (supporting gaming scenarios). Although the playback latency is low, it has not yet been measured in a gaming scenario. Playback of game audio is impossible in the system until an emitter that emulates a virtual audio output on a desktop computer is implemented. The fact that these three goals were not completed means some of the results will likely change in the future. For example, the performance metric regarding audio mixing should increase after EQ is implemented, as that step would then require computing a couple extra Fast Fourier Transforms for mixing.

The chosen performance metrics do not offer a complete picture of the quality of the system. They do not describe the user experience in a direct way. They were chosen as a main data point as most identified goals are either binary and can be verified easily if the implementation was successful or not, or they are expressed by a clear metric, such as delay. Additional validation techniques should complement these metrics in order to verify and validate the interpretations made based on performance data.

The tests were all carried out in an uncontrolled network with varying traffic conditions. Although this may be representative of a home network, it means that irregularities in transfer speeds could be caused by third parties on LAN. This situation may have influenced the obtained results.

4.5 Future work

The relays, emitters and control interface are still in active development. The prototype presented here shows that the architectural model is viable and provides the necessary groundwork for the next phase of development. To complete our analysis of the prototype, we must establish which goals need to be carried over to this next phase and what changes in the development approach must be made based on the knowledge gained from this prototype:

1. Emitters as audio drivers for desktop are a high priority as they inhibit testing testing with a class of applications.
2. Improve upon data forwarding.
3. Verify and iterate upon the interface.
4. Implement and test audio recordings within the system.

5. Add EQ support in the mixer.

The current basis for the software architecture will continue to be used, and achieving these goals should allow users to use a prototype running on Raspberry Pis without experimenter supervision.

4.5.1 Interface Verification

At the current state, the interface's primary goal is to exemplify the idea behind it and to allow some degree of control over the system apart for the debugging interface. Further work on accessing its quality as a User Interface in terms of usability contains performing user studies. These should be focused around the aspect of usability through the System Usability Scale (Brooke et al., 1996). This would add value to the discussion of goal 1 (ease of use and setup) and be a requirement for starting any further iteration.

5 Conclusion

As this thesis shows, most of the identified goals were achieved. A prototype of a multi-zone audio system architecture was shown to be a viable solution to accomplish these goals. The system was able to run on the designated testing hardware as well as it was shown that lower performance hardware with a higher quality sound card would be preferred for future iterations. Finally, this thesis establishes the goals required for the next iteration of development.

References

Aberer, K., Alima, L. O., Ghodsi, A., Girdzijauskas, S., Haridi, S., & Hauswirth, M. (2005). The essence of p2p: a reference architecture for overlay networks. In *Peer-to-peer computing, 2005. p2p 2005. fifth ieee international conference on* (pp. 11–20).

Balakrishnan, H., Kaashoek, M. F., Karger, D., Morris, R., & Stoica, I. (2003). Looking up data in p2p systems. *Communications of the ACM*, *46*(2), 43–48.

Brooke, J., et al. (1996). Sus-a quick and dirty usability scale. *Usability evaluation in industry*, *189*(194), 4–7.

Burg, J., Romney, J., & Schwartz, E. (2012). *Digital sound and music - concepts applications and science*. Retrieved 2016-06-07, from `http://digitalsoundandmusic.com/curriculum/`

Control4 Corporation. (2016). *Catalog.* Retrieved 2016-06-07, from `http://www.control4.com/solutions/catalog/multi-room-audio`

Correia, M., Neves, N. F., & Veríssimo, P. (2006). From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, *49*(1), 82–96.

Haas, H. (1972). The influence of a single echo on the audibility of speech. *Journal of the Audio Engineering Society*, *20*(2), 146–159.

Hadzilacos, V., & Toueg, S. (1994). *A modular approach to fault-tolerant broadcasts and related problems* (Tech. Rep.). Cornell University.

Lamport, L., & Melliar-Smith, P. M. (1985). Synchronizing clocks in the presence of faults. *Journal of the ACM (JACM)*, *32*(1), 52–78.

Lamport, L., et al. (2001). Paxos made simple. *ACM Sigact News*, *32*(4), 18–25.

Lua, E. K., Crowcroft, J., Pias, M., Sharma, R., & Lim, S. (2005). A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, *7*(2), 72–93.

Melchior, J., Mejías, B., Jaradin, Y., Van Roy, P., & Vanderdonckt, J. (2013). Improving duis with a decentralized approach with transactions and feedbacks. In *Distributed user interfaces: Usability and collaboration* (pp. 17–25). Springer.

Mills, D. L. (1991). Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, *39*(10), 1482–1493.

Mills, D. L. (2006). Simple network time protocol (sntp) version 4 for ipv4, ipv6 and osi.

Özsu, M., & Valduriez, P. (1991). *Principles of distributed database systems*. Prentice Hall.

SAMSUNG. (2013). *Samsung wireless multiroom audio system hub.* Samsung Electronics America. Retrieved 2016-06-07, from `http://www.samsung.com/us/video/home-audio/WAM250/ZA`

Smith, J. (2009). Patterns-wpf apps with the model-view-viewmodel design pattern. *MSDN magazine*, 72.

Sommer, P., & Wattenhofer, R. (2009). Gradient clock synchronization in wireless sensor networks. In *Proceedings of the 2009 international conference on information processing in sensor networks* (pp. 37–48).

SONOS, Inc. (2016). *Sonos hifi wireless speakers and home audio systems.* Retrieved 2016-06-07, from `http://sonos.com/`

Tanenbaum, A., & van Steen, M. (2007). *Distributed systems: Principles and paradigms*. Pearson Prentice Hall.

Tesoriero, R., Lozano, M., Vanderdonckt, J., Gallud, J. A., & Konstan, J. A. (2012). distributed user interfaces. , 2719–2722.

Turek, J., & Shasha, D. (1992). The many faces of consensus in distributed systems. *Computer*, *25*(6), 8–17.

Villanueva, P. G., Tesoriero, R., & Gallud, J. A. (2013). Revisiting the concept of distributed user interfaces. In *Distributed user interfaces: Usability and collaboration* (pp. 1–15). Springer.

Appendices

A Formative interviews

This section describes an open ended interview carried out in order to define the requirements of a multi-zone audio system. The interview focused on establishing the required features of the system and identifying the usage patterns that needed to be supported. The following questions were used as guidelines:

- Where and how do you listen to music?
- What services do you use for music? (What about physical sources? CDs?)
- What else do you listen to (apart from music)?
- What devices capable of playing sound do you use? (both input and output)
- (The concept of multi-zone audio should be introduced here)
- What would you like to control about this? Most common interactions with the system.
- Is there something you wish your audio system did, but it doesn't do?
- What concerns do you have about such a system?

B Debugging interface

A low level debugging interface was created to monitor multiple relays across a network. This was distinct from the final user interface as it needed to be modified quickly whenever relay code changes took place. This debug interface is written in C++ using IMGUI[14] and reuses most of the relay code base for communication. The interface can show several relays side by side simultaneously along with their status and a set of periodically updated properties. Only a single instance of the debug interface could be open on the same network where relays were active.

Figure 8 shows the interface and the type of data it collected. Both the last sent value and the average of the last 10 entries are shown in the interface. Furthermore, certain logs are transmitted from all relays to the application and preserved.

[14]https://github.com/ocornut/imgui "Bloat-free Immediate Mode Graphical User interface for C++ with minimal dependencies."

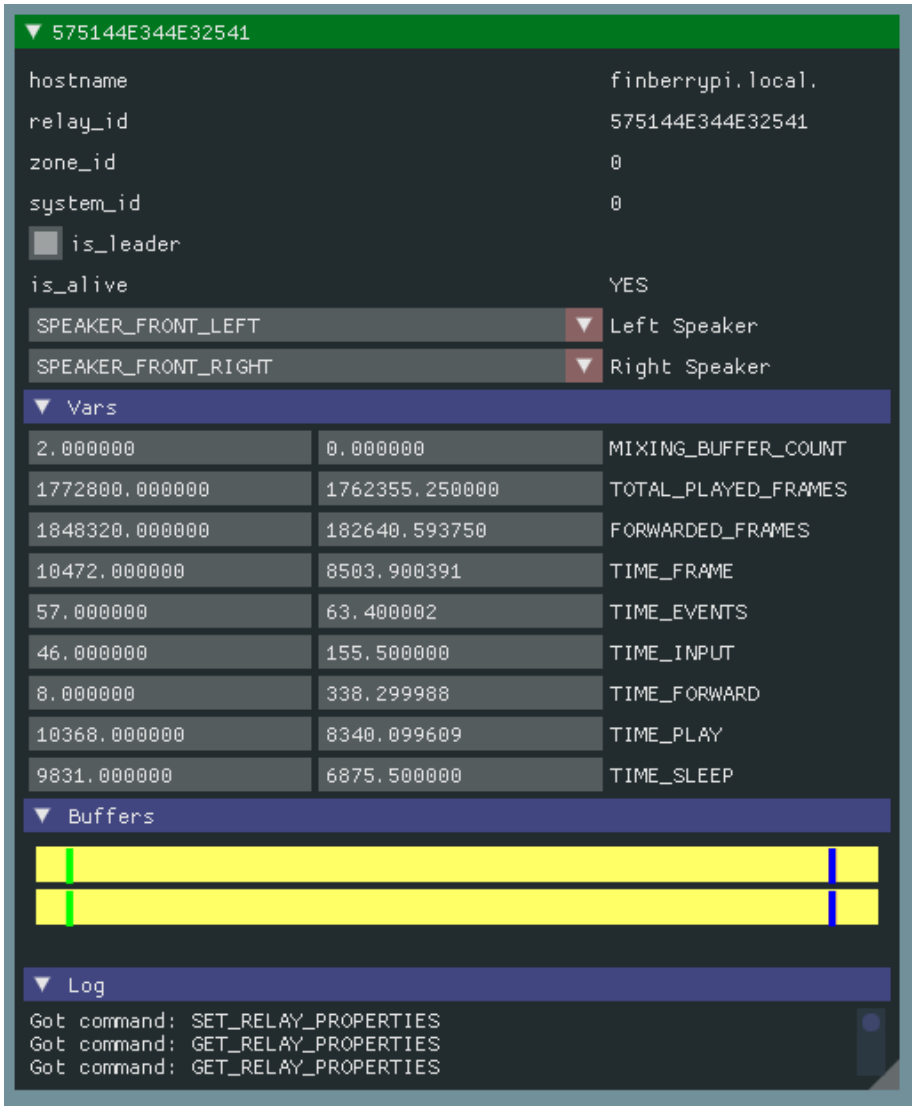

Figure 8: Debug interface. Shows major configuration properties of a relay. Allows the manipulation of these basic properties. Shows performance and debugging statistics. Shows mixing handles (green playback, blue write, red forward). Here red is not visible as the entire buffer was forwarded. Shows a subset of the logs produced by the relay.