Networked Microcontrollers for Distributed Audio Spatialisation

Thomas Albert Rushton

Sound and Music Computing, 2023-06

Master's Project

Typset with pdfTeX π-2.6-1.40.24 and Latexmk 4.77

Department of Architecture, Design and Media Technology
Aalborg University
https://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Networked Microcontrollers for Distributed Audio Spatialisation

Theme:
Scientific Theme

Project Period:
Spring Semester 2023

Project Group:
n/a

Participant(s):
Thomas Albert Rushton

Supervisor(s):
Stefania Serafin
Romain Michon (external)

Copies: 1

Page Numbers: 61

Date of Completion:
May 25, 2023

Abstract:

Systems for spatial audio typically demand large numbers of loudspeakers and audio hardware capable of serving many output channels. Centralised systems of this sort are inflexible, and, due to their reliance on specialist audio hardware and software, costly, with a high barrier to entry. Recent decades have seen increasing interest in both audio spatialisation and the transmission of audio over computer networks. Advancements in low-cost microcontroller platforms with support for networking and audio processing, may facilitate a decentralised approach to audio spatialisation systems. Based on one such platform, this thesis describes the development of a modular, scalable system of distributed audio processors with applications to spatial audio. Though faced by significant technical challenges, the system demonstrates interesting initial perceptual results. Findings are commensurate with a capability, with further development and research, to disrupt and democratise the fields of spatial and immersive audio.

Contents

Preface

This project began life as an internship assignment on the *Emeraude* research team at Inria (*Institut national de recherche en sciences et technologies du numérique*) in Villeurbanne, France, under the supervision of itinerant Faust evangelist, Romain Michon. He tasked me with expanding on a proof-of-concept JackTrip microcontroller client that a previous team of interns had created, and I spent a couple of months wrestling with the Teensy platform, JackTrip and Linux audio in general, and gazing at Wireshark captures. The result was a distributed Wave Field Synthesis implementation based on JackTrip and Faust, which was presented at the 2022 *Programmable Audio Workshop* (also held in Villeurbanne). A paper detailing the work was submitted to the 2023 *Sound and Music Computing* conference and very kindly accepted at peer review; it will be published in the proceedings of the conference later in 2023.

I will not claim to be more than a neophyte where much of the content of this thesis is concerned — particularly with regard to the low-level specifics of network transmission and computing hardware. That being said, the work of research has invariably focused on its engineering aspects, rather than its creative applications, so the former is where my efforts must lie in terms of documentation; I hope that the reader will bear with me as I make what best I can of my understanding.

Acknowledgements

My sincerest thanks to Stéphane Letz for sharing his unrivalled knowledge of JACK and Faust; to Dan Overholt and Peter Williams for their support with the vagaries of working with microcontrollers; and to Shawn Silverman for taking the time to walk me through how ethernet works on Teensy. Thanks also to Romain Michon and Stefania Serafin for their patient supervision.

Copenhagen, May 25, 2023

Thomas Albert Rushton
<trusht21@student.aau.dk>

Chapter 1

Introduction

Spatial and immersive audio techniques have been the beneficiaries of significant research interest over the past few decades. The more recent explosion in virtual and augmented reality technologies, and *object-based* audio techniques, has led to an acceleration in interest in the creation of virtual sound fields via approaches such as Wave Field Synthesis (WFS) and Higher Order Ambisonics (HOA)[1–4]. These techniques call for the deployment of large numbers of loudspeakers, and centralised, *in situ* installations of dedicated hardware and software. The costs associated with such installations have seen them relegated to the reserve of concert venues, cinemas, and institutions with the means to purchase and operate monolithic systems of this sort.

Recent advancements in embedded computing mean that there now exist an assortment of small, low-cost devices with support for audio digital signal processing (DSP). These devices are easily programmable, open source, and may provide support for communication over ubiquitous computer networking equipment and protocols. A network of such devices could be used to *distribute* the problem of audio spatialisation, potentially lowering the barrier of entry to what is otherwise a comparatively exclusive branch of audio research.

The work that this thesis describes is an exploration of the possibility of achieving such an outcome. As such, it is concerned primarily with the transmission of digital audio signals throughout a computer network. It is meaningful, then, to reflect on the nature of such signals, a selection of their properties most pertinent to this work, and their representations within computer systems and networks.

1.1 Digital Audio Signals

In a digital audio system with sampling rate F_s, an audio signal y is composed of samples $y[n]$, each representing the amplitude of the signal at a given point in time t, where $t = n/F_s$. For arithmetical convenience, sample amplitudes are typically

considered to be floating point numbers, constrained to the range $y \in [-1, 1]$ (see Figure 1.1). It is in this form that samples are typically handled during the processing stage of a DSP algorithm, but the true underlying representation is concealed, and matters such as numerical resolution and precision abstracted away.

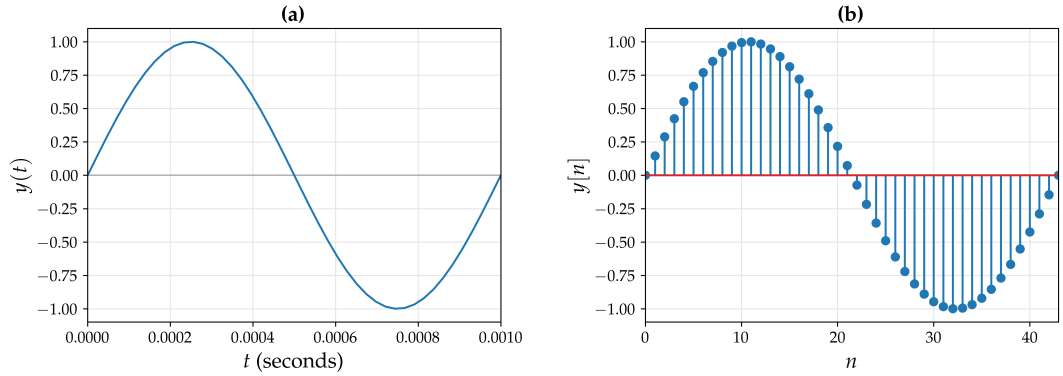


Figure 1.1: An audio signal (1 kHz sine wave); (a) in continuous time; (b) sampled at intervals of $1/Fs$ seconds, with $F_s = 44\,100$ Hz.

Samples undergo format conversion at various stages during processing, such as from an integer pulse code modulation (PCM) filetype to a stream of floating point audio samples in a digital audio workstation (DAW), or from a floating point audio stream in an audio device driver to an integer stream to be handled by a hardware codec. For the most part a user, and even a developer of audio software, need not concern themselves with the rudiments of sample representation and conversion; as shall be shown, however, under certain circumstances these digital fundamentals must be dealt with directly.

1.1.1 Numerical Representation

Commonly-encountered sample formats include 16 and 24 bit integer, and 32 bit floating point. Broadly speaking, more bits afford greater resolution in terms of sample amplitudes that can be represented, with ramifications for dynamic range and signal to noise ratio, as well as for storage and throughput. Integer formats offer comparatively poor resolution at low amplitudes due to the mismatch between the linear distribution of their values versus the logarithmic nature of sound intensity. Floating-point formats address this by having a logarithmic distribution of available values; the IEEE standard for single precision (i.e. 32-bit) floating point numbers[5] dictates that precision is greatest around zero, with around half the available numbers lying in $[-1, 1]$.[1] For brevity's sake, and for its pertinence later in the text, consider the 16-bit case, and a single 16-bit integer audio sample:

[1]Actually, given available negative exponents from -126 to -1, and a fraction of 23 bits for each exponent, around 49.2%. The remainder consists of numbers in $\pm[1, 3.4 \times 10^{38}]$, plus space reserved

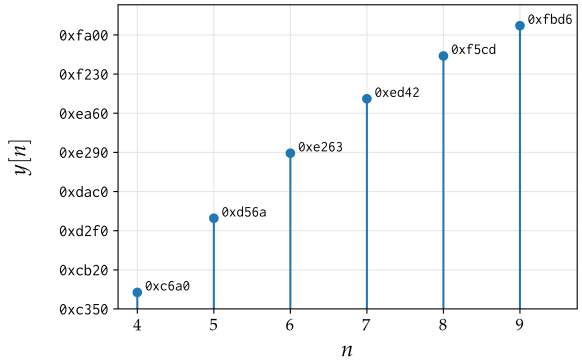


Figure 1.2: Detail of Figure 1.1 (b). Sample amplitudes converted to 16 bit hexadecimal values.

Listing 1.1: A 16-bit audio sample, binary representation

```
11010101 01101010
```

Separating the bits into two groups of eight is reflective of the fact that the eight-bit *byte* is typically the unit of transmission in computer systems. Grouping the bits in this way points to their expression in hexadecimal format, transforming the bytes into more easily-digestible morsels of two digits:

Tighten up the wording around unit of transmission?

Listing 1.2: A 16-bit audio sample, hexadecimal representation

```
d5 6a
```

A number of this kind may also been seen represented (in C and C++ code for example) as `0xd56a` with '`0x`' indicating that the number to follow is in base sixteen. Sixteen bits grant access to $2^{16} = 65{,}536$ distinct amplitude values for each sample; what listing 1.2 tells us is that, in decimal terms, this sample should take the 54,634$^{\text{th}}$ amplitude value[2]. See Figure 1.2 (and the sample at $n = 5$) for an illustration of this.

1.1.2 Storage and Transmission

Listings 1.1 and 1.2 hint at the property of *endianness*, which is to say the order in which a number's component bits and bytes appear[6]. The above representations mirror the left-to-right nature of western written language and numbers, being *big endian* at the levels of both bit and byte, with the most significant bit (and most significant byte, both abbreviated *MSB*) appearing first.

Paraphrasing Cohen[6], if digital audio signals could be transmitted in their

for representing infinity and *non-numbers*.

[2]For convenience and explicitness elsewhere in the text, decimal equivalents to hexadecimal numbers will be indicated with subscript 10, e.g. 54634_{10}.

entirety, i.e. the unit of transmission was *an audio signal*, then endianness would not matter. Practically speaking, however, in order to be handled by software or hardware, or transmitted over a network, audio signals must be broken down into temporally-distributed blocks, those blocks into samples, samples into bytes, and bytes into bits; correct endianness must be observed with respect to differing computer architectures, file formats, and transmission protocols.

The PCM WAV file format, for example, dictates that the least significant byte of each sample is stored first — little endian — but that bit order should be big endian[7]; thus the audio sample in listing 1.2 should be stored in a `.wav` file as `0x6ad5`. The ethernet standard for communication over local area computer networks[8] calls for something akin to the opposite: octets (bytes) are sent *"top to bottom"*, but each octet is transmitted with its least significant bit first.

Section in intro on spatial audio?

Chapter 2

Analysis

In this chapter an overview is given of the problem space. The topic of delivering data, and audio signals in particular, over computer networks is addressed, with particular focus on the user datagram protocol for network transmission. Suitable hardware platforms for implementing a low-cost distributed audio system are described, as are audio spatialisation algorithms, with an emphasis on wave field synthesis. Prior art in distributed audio systems is considered, and aims for the development of a new system, taking advantage of current consumer-grade microcontroller technology, described.

2.1 Networked Audio

The transmission of audio data has been a topic of research interest since the earliest days of computer networking as it is recognised today, i.e. over packet-switched networks whereby data to be transmitted is grouped into packets, each consisting of a header and a payload. Voice transmission over ARPANET was being conducted as early as 1974[9] and the first standard for voice communication over packet-switched networks — the Network Voice Protocol (NVP) — was released in 1977[10].

With its references to 'calling', 'ringing', and termination of a connection as a 'goodbye', it is clear that the NVP standard was intended for digital telephony. Indeed, research on networked audio was primarily interested in telephony well into the 1990s, focusing on real-time voice communication over wide area networks (WAN) with efforts centring on *quality of service* (QoS), particularly with regard to the perennial issues of latency, packet loss, and network jitter — inconsistencies in the rate of packet transmission[11, 12]. Work at this time dealt with streams of compressed audio data, and speech coding algorithms to overcome the deleterious effects of dropped packets over unreliable network paths and low-bandwidth connections.

Whereas the priority for digital telephony, and later voice over IP (VoIP) systems, is intelligibility, for musical purposes fidelity, and the use of uncompressed audio

signals, is of greater concern. With the increasing availability of high-speed internet connections in the late 1990s came research into transmitting uncompressed audio data over the internet[13, 14]. Work of this sort was spearheaded by the *SoundWIRE* project, developed by researchers at McGill University and the Centre for Computer Research in Music and Acoustics (CCRMA) at Stanford University, and took the form of a wide variety of experiments with high quality audio over both WAN and local area networks (LAN). These experiments included LAN-based real-time musical performances[13], concert streaming over WAN[13, 14], and sonification of QoS via a distributed digital waveguide dubbed the *Network Harp*[13, 15].

2.1.1 Protocols and Systems

VoIP research in the 1990s focused on matters such as audio codecs and data compression[12, 16], seeking a compromise with the *best-effort* nature of internet service. The SoundWIRE project, in search of high audio quality, turned its attention directly to the fundamental transport layer protocols of the Internet Protocol suite: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Doing away with data compression removed a layer of computational overhead inherent to VoIP, and Chafe et al. characterised SoundWIRE and associated work as taking a "simplified approach"[13] to networked audio, emphasising the importance of delivering multichannel audio of at least CD quality (16 bit, 44.1 kHz) with as little latency as possible.

SoundWIRE experiments included using TCP for unidirectional transmission such as concert streaming. TCP is in fact a bidirectional protocol, but its *connection-oriented*, one-to-one design, whereby networked entities establish a connection via a 'handshake', following which they exchange packets of data, allows for mechanisms that guarantee packet ordering and provide protections against packet loss[17, 18]. These mechanisms mean that, at the expense of increased latency, quality of service, and thus audio fidelity, is ensured; ideal for a remote concert scenario. The strict one-to-one nature of TCP clearly places limits on its applicability to distributed computing, however.

UDP by comparison is a *connectionless* protocol, providing no guarantees on the integrity of the stream of network data, but equally none of the computational, or indeed temporal, overhead that such guarantees introduce. A network entity can send UDP packets to a network address irrespective of whether any other entity with that address exists. Further, *many-to-many* (multicast) and *one-to-many* (broadcast) modes of transmission are possible via address spaces reserved as part of the internet protocol standard[19]. Via UDP, SoundWIRE was able to run as a distributed digital waveguide over a WAN spanning ~4500 km[13].

From the SoundWIRE project emerged *JackTrip* [20, 21], a hybrid system that couples a TCP handshake with data transmission over UDP, thus sidestepping the

overhead of TCP packet flow control. Rather than rely on TCP's built-in mechanisms for stream integrity, JackTrip supplements UDP with a number of optional buffering strategies that aim to tailor its use to operation over local versus wide area networks. In this sense it is more flexible than TCP, but in effect JackTrip moulds UDP transmission into something akin to the connection-oriented model of TCP, and, in its 'hub server' mode, into a kind of *multiple one-to-one* design — multicast transmission is not possible.

UDP has emerged to be the protocol of choice for platforms enabling remote musical collaboration, serving as the basis for systems such as NetJACK[22], part of the JACK Audio Connection Kit (a cross-platform audio host), Jamulus[23], Soundjack[24], and other jamming-focused platforms, plus more recent entrants to the arena of networked audio and music such as Elk OS, which operates on a closed-source, but ultimately UDP-based, system[25]. It even serves as a critical component of proprietary systems such as Dante (Digital Audio Network Through Ethernet)[26].

AoE in the Audio Industry

In parallel with the work being carried out by researchers such as those developing SoundWIRE, JackTrip and NetJACK, audio industry bodies were taking an interest in networked audio. Prominent amongst these bodies were the IEEE (Institute of Electrical and Electronics Engineers) and AES (Audio Engineering Society) standards groups, and companies like Audinate, the creators of the Dante system. Traditional large-scale audio systems such as those used in broadcast, concert venues and recording studios rely on the installation of unwieldy systems of analogue hardware and cabling, with many potential points of failure. Seeking literally to lighten the load posed by "hundreds of kilograms"[27] of analogue cabling in analogue audio installations, in the 2000s industry entities were looking to high speed ethernet as a means to simplify the provision of high-quality, multichannel audio in industry settings.

Dante, with its promise of low-latency, highly-multichannel audio over ethernet, and device synchronisation via Precision Time Protocol (PTP), has become the de facto standard in this area[27][1]. In 2011, IEEE released the Audio Video Bridging (AVB, IEEE 802.1) standard, and AES67 followed in 2013; these open technical standards describe operation at layers below TCP and UDP in the hierarchy of network technology, and provide frameworks for interoperability between AoE and AoIP systems, including mechanisms for device discovery and synchronisation. Being standards, and not implementations in themselves, it is then up to manufacturers to

[1]Bakker et al. refer to Dante as an "open" system. This is perhaps true in the sense that companies can incorporate the Dante system into their own products under licence from Audinate, but, from the perspective of the research community, Dante is very much a closed-source initiative.

implement their recommendations in their products. (AES67, for example, has in fact been implemented in Dante[27].)

The proprietary nature of Dante means that, despite the intriguing nature of its guarantees of device synchronicity to the sub-microsecond range, it is not a serious target for further research. Two interrelated factors preclude the further consideration of AVB and AES67 in this work: 1) their reliance on network protocols such as PTP that are not supported by ubiquitous networking equipment; 2) the high cost of devices that do provide such support. Ultimately, if an accessible, low-cost solution is sought, attention must be turned back to the transport layer, and UDP.

2.1.2 Anatomy of a Datagram

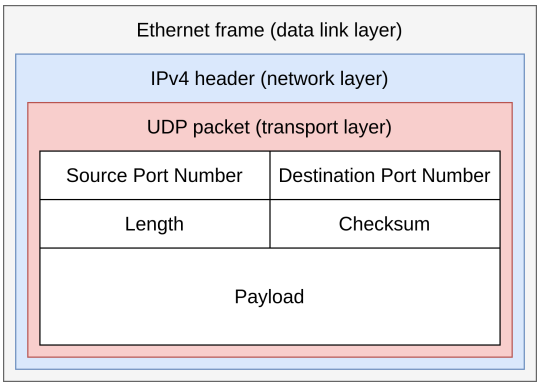


Figure 2.1: Structure of an ethernet frame containing a UDP packet.

A UDP packet consists of data encoded an 8-bit integer format. Being a transport layer protocol, a UDP packet is in fact preceded in an ethernet frame by information relating to lower layers in the hierarchy of network topology: the network layer and data link layer. The structure of a UDP packet (within an ethernet frame) is shown in Figure 2.1.

Listing 2.1: Network capture: ethernet frame containing a UDP packet

```
       00 01 02 03 04 05 06 07  08 09 0a 0b 0c 0d 0e 0f

0000   00 00 0c 9f f5 16 d8 80  83 80 8c b7 08 00 45 00   ..............E.
0010   00 2a 83 ba 40 00 40 11  0a 09 ac 1e fe df 01 01   .*..@.@.........
0020   01 01 ed 3f 22 b8 00 16  01 74 48 65 6c 6c 6f 2c   ...?"....tHello,
0030   20 77 6f 72 6c 64 21 0a                            world!.
```

The ethernet frame in listing 2.1 was generated using the Netcat (nc) command line

utility[2], and captured with Wireshark network packet analysis software[3]. Four-digit numbers to the left indicate the hexadecimal number of the byte at the beginning of the corresponding line. The central block of two-digit hexadecimal numbers are a representation of the bytes in the ethernet frame, labelled above by byte position. The block of characters to the right are the same data represented with ASCII encoding.

Bytes `0x0000` to `0x000d` make up the ethernet header, consisting of the media access control (MAC) addresses of the destination and source; the final two bytes of the ethernet header, `0x0800`, indicate that this is an *internet protocol version 4* frame. Bytes `0x000e` to `0x0021` are the IPv4 header; this contains information about the internet protocol part of the packet, such as its length in bytes — `0x002a` (42_{10}) — and the source and destination IP addresses, encoded as groups of four-bytes. The source IP address, for example, is `0xac1efedf`, a 32-bit encoding of the more familiar-looking `172.30.254.223`.

Beginning at byte `0x0022` is the UPD header. This contains the source port (`0xed3f`, 60735_{10}), destination port (`0x22b8`, 8888_{10}), the length of the UDP part of the frame (`0x0016`, 22_{10} bytes) and ends with a checksum, which can be used to verify the integrity of the packet[4]. The packet payload begins at byte `0x002a`, and consists of bytes corresponding with the ASCII characters `Hello, world!`, plus `0x0a`, the line feed (LF) character, captured and sent by Netcat when the user hit the return key.

Netcat takes data supplied to a computer's standard input stream, in this case characters entered in a terminal emulator, and uses this data as the payload for the packet to be transmitted. The payload of a UDP packet can of course consist of any data which can be appropriately byte-encoded, such as a stream of audio samples, or audio control data such as MIDI or OSC messages.

Ethernet frames, and UDP datagrams by extension, are subject to size limitations. The maximum transmissible unit (MTU) of a transport medium is the limit on the size of a packet that can be sent without fragmentation, i.e. being split into multiple subpackets. The theoretical limit on datagram size is 65535 bytes[17] — this is the largest number representable by the two bytes allocated to the 'Total Length' field of the IPv4 header — however in practice the data link layer imposes a basic limit of 1500 bytes on the payload of an ethernet frame[8].

2.2 Hardware Platforms

The notion of taking a distributed approach to DSP is reliant on the identification of a suitable supporting hardware platform. For a distributed audio application, the ideal computing platform should be small, inexpensive, plus easily and rapidly

[2] `https://nc110.sourceforge.io/`

[3] `https://www.wireshark.org/`

[4] In the remainder of this work it is assumed that transmission over LAN is unlikely to result in packet corruption. The UDP checksum is not used, nor discussed any further.

Platform	Processor	Memory	Price
Teensy 4.1[6]	ARM Cortex-M7 600 MHz	1 MB SDRAM	~€32
Daisy Seed[7]	ARM Cortex-M7 480 MHz	64 MB SDRAM	~€28
ESP32-LyraTD-MSC[8]	Xtensa LS6 240 MHz	8 MB PSRAM	~€52
Bela[9]	ARM Cortex-A8 1 GHz[10]	512 MB DDR3	~€209

Table 2.1: Comparison of selected embedded audio development platforms. Prices as of May 2023.

programmable; of course, it should also provide audio and networking capabilities.

Recent years have seen the emergence of a number of small, low-cost, open-source platforms for embedded development, perhaps best known amongst these being Arduino[5]. Though support for audio is limited via official Arduino models, a number of audio-specific, Arduino-like systems have been developed, principal amongst these being various ESP32 models, Daisy Seed, and the Teensy family of microcontrollers. Also worthy of consideration is the Bela platform; though not a true microcontroller (being based on the Beaglebone single-board computer, and running an opeating system) it is a small-footprint audio-focused device suitable for embedded applications. Platforms of this kind that also possess networking support may also fall under the category of *Internet of Things* (IoT) devices.

A comparison of selected devices can be found in table 2.1. Bela is significantly more powerful than the microcontroller systems, but it is commensurately costly. Daisy Seed is well-appointed with memory (which is important for DSP algorithms featuring long delay-lines, for example), but does not feature ethernet support. Teensy4.1, and the selected ESP32 device support networking via their repsective ethernet add-ons, but the latter's CPU is under-powered. Though lacking in memory, Teensy's processor and low price make it the most suitable candidate to support a distributed, networked audio implementation.

2.3 Approaches to Audio Spatialisation

Audio spatialisation is the practice of distributing sound in space. In terms of the reproduction of primary sound sources, i.e. sound captured by microphones, recorded as digital audio files, or synthesised in real-time, spatialisation can be achieved simply by delivering those primary sources to secondary sound sources, i.e. loudspeakers, placed at arbitrary positions relative to a listener. Such an approach is,

[5] https://www.arduino.cc/

[6] https://www.pjrc.com/store/teensy41.html

[7] https://www.electro-smith.com/daisy/daisy

[8] https://www.espressif.com/en/products/devkits/esp-audio-devkits

[9] https://shop.bela.io/products/bela-starter-kit

[10] https://beagleboard.org/black

of course, inflexible; loudspeakers being stationary entities by-and-large, the idea of moving a sound source is physically impractical, and, that of locating a sound source between loudspeaker positions physically impossible. Taking advantage of auditory cues, however, and the nature of the propagation of sound, it is possible to suggest the presence of primary sources at arbitrary locations, independent of the secondary source distribution. The motivation behind audio spatialisation, then, is to create (or indeed *re*create) sonic environments for creative and immersive purposes, such as for virtual reality experiences, in cinematic settings, for music production or art installations, to give but a handful of examples.

A number of techniques exist for what is termed *sound field synthesis*[28, 29], all of which essentially take the form of applying some manner of *driving function* to an input audio signal to generate an appropriate driving signal to be delivered to a secondary sound source in the listening environment[28]. For a loudspeaker at position $\mathbf{x} = \begin{bmatrix} x & y & z \end{bmatrix}^T$, the driving signal $\hat{D}(\mathbf{x}, \omega)$ can be expressed, in the frequency domain, as the product of the input signal, $\hat{S}_{\text{in}}(\omega)$ and the driving function $D(\mathbf{x}, \omega)$:

$$\hat{D}(\mathbf{x}, \omega) = \hat{S}_{\text{in}}(\omega) \cdot D(\mathbf{x}, \omega), \tag{2.1}$$

where f is time frequency and ω denotes radian frequency, $\omega = 2\pi f$. Moving to the time domain, the multiplication of the input signal and driving function changes to a convolution, and equation (2.1) becomes:

$$\hat{d}(\mathbf{x}, t) = \hat{s}_{\text{in}}(t) * d(\mathbf{x}, t), \tag{2.2}$$

where t denotes time.

An early sound field synthesis approach, referred to as an 'acoustic curtain'[30] entailed placing microphones in one space, such as a concert auditorium, and, in another space, loudspeakers at locations corresponding to those of the microphones. In this case, there are as many input signals $\hat{S}_k(\omega)$ as there are microphones, and the driving function reduces to a 'pass-through' of each signal to its corresponding loudspeaker, i.e. $D_k(\mathbf{x}, \omega) = 1$.

The acoustic curtain is a relatively rarely-deployed technique (though similar recreation of 'real' sound fields is still conducted, albeit typically by way of ambisonic recording and reproduction) and sound field synthesis is most often concerned with the distribution and movement of artificial or arbitrary sound sources. Commonly-employed approaches to sound field creation can be grouped into two broad categories: amplitude- and time-based panning techniques, and physical sound field recreation approaches.

Periphony

The former, *periphonic*, types are perhaps more familiar to the layperson and encompass stereophony and surround-sound systems, consisting of secondary sources in a

horizontal planar arrangement equidistant to the listening position. These techniques exploit interaural level difference (ILD) cue, i.e. the difference in perceived amplitude relative to the listener's ears[30–32], to encourage the listener to localise sound to a position on the circumference of an arc or circle around the listening position. This is achieved by weighting the amplitudes of signals sent to the secondary sources, creating a "phantom" sound source that may appear to emanate from a position between loudspeakers. For systems of this sort, the driving function is a constant scalar value, or, for a moving phantom source, a time-varying function that returns a scalar value. Such periphonic approaches can extend to three dimensions in the case of vector base amplitude panning (VBAP)[31], which uses trios of speakers to position phantom sources on the surface of a sphere.

Time-based panning effects, by contrast, make use of the interaural time difference (ITD) cue to give the impression of a phantom source located toward the loudspeaker producing the signal at the earliest time[31, 32]. Thus the driving function for a time-based panning system is a delay of the form:

$$d(\mathbf{x}, t) = \delta(t - \tau), \tag{2.3}$$

where τ is the duration of the delay.

The effects of ILD and ITD cues may transfer to headphone-based listening, in which case, rather than periphonic, they become a sort of *in-head* localisation[28]. It is worth mentioning that these cues vary in their effectiveness with frequency and, excepting the case of headphone-based listening, intersect with cues related to listener's torso[32] and the head-related transfer function[33, 34].

Periphonic approaches (again, headphones excepted) are subject to the phenomenon of an ideal listening position, or *sweet-spot*[29, 32], that is a listening position away from which the spatialisation effect is significantly degraded. As such, these techniques are not suited to subjection to multiple listeners, nor to immersive auditory experiences permitting the participant to move freely about their environment. Additionally, phantom sources are inherently bound to the periphery on which they reside; there is no authentic way to model a phantom source at greater (or lesser) distance, though using reverberation and amplitude cues may offer a satisfactory perceptual impression.

Physically-Inspired Techniques

Physical approaches fall into two main types: wave field synthesis (WFS) and ambisonics (and higher-order ambisonics — HOA). Rather than directly manipulating sound localisation cues, these types seek to trigger those cues indirectly by synthesising a sound field as if it had been created by 'true' acoustic sources, as opposed to loudspeakers. In the case of ambisonics, the sound field is decomposed into 'spherical harmonics', spatial functions described by linear sums of directional components of

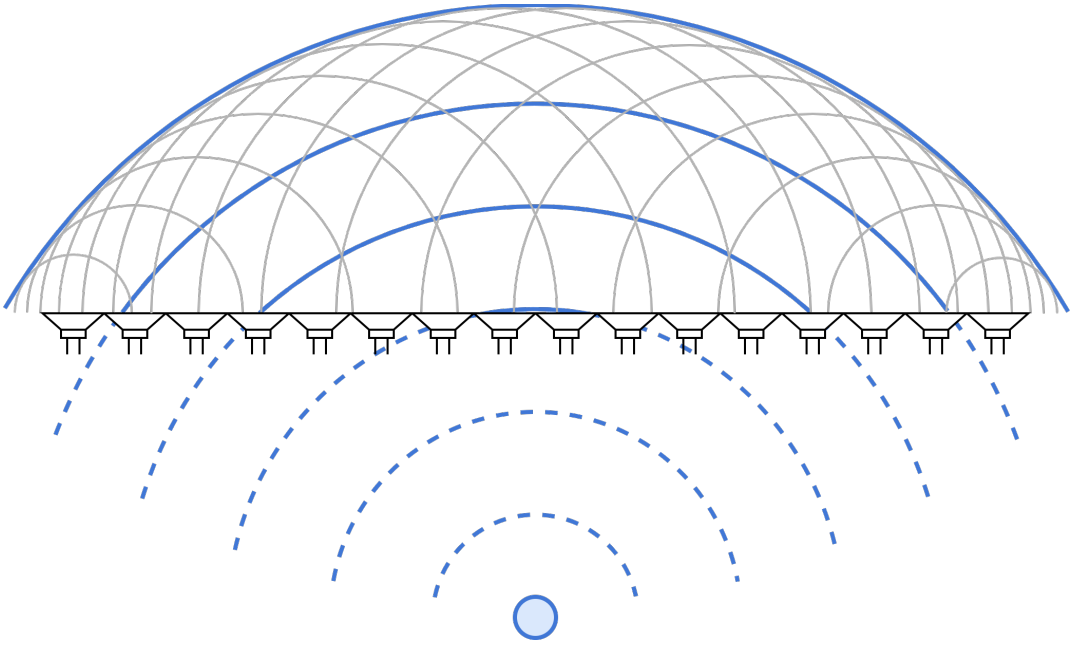


Figure 2.2: *Holophony*. Huygens' principle states that the propagation of a wavefront can be recreated by a collection of secondary point sources. The bottom of the figure represents a virtual sound field, and the top a real sound field, separated by a row of secondary point sources (loudspeakers). The small blue circle represents a virtual sound source and the blue dashed arcs are virtual wavefronts associated with that sound source; the grey arcs are wavefronts produced by the array of secondary point sources; the solid blue arcs represent the propagation of a reconstructed wavefront in the real sound field.

increasing order[29]. Spherical and plane waves can be reproduced, corresponding with virtual point sources and sources at 'infinite distance' respectively.

Ambisonics, like periphonic approaches, suffers from a sweet-spot effect which exacerbates with attempts to reproduce sounds of higher frequency. The ideal listening area can be broadened by implementing ambisonics at higher order, increasing the density of the distribution of secondary sources. Doing so obviously has ramifications for the physical complexity of an ambisonics installation, and, since higher-order spherical modes must be calculated, in terms of computational demands placed on the system.

2.3.1 Wave Field Synthesis

WFS, reminiscent of the acoustic curtain, is based upon Huygens' principle, originating in the field of optics, which states that a propagating wavefront can be recreated by a distribution of secondary point sources[1, 35, 36] (see Figure 2.2). It is variously termed a form of *acoustic holography* or *holophony*[28, 37]. Effectively, by timing the reproduction of an input signal at an array of secondary sources, a wavefront associated with a virtual sound source can be synthesised. To simulate distance cues,

a filter can be applied to model losses to the virtual medium of acoustic propagation. The principle assumes a continuous array of secondary sources but of course in practice it is necessary to use a discrete array of loudspeakers, which, much as is the case with HOA, has ramifications for spatial resolution; to mitigate the issue of spatial aliasing, whereby sounds of higher frequency cannot be recreated unambiguously[38], secondary sources should be placed very close together. Consequently, to serve a large listening area, many speakers, and thus many audio channels, are required.

Via appropriate timing of the delivery of a primary sound source to the secondary source array, it is possible to synthesise virtual sound sources, plane waves, and *focused* sound sources, corresponding with concave, flat, and convex synthesised wavefronts respectively; the latter, dependent on the location of the listener, appear to emanate from within the real sound field, rather than its virtual counterpart.

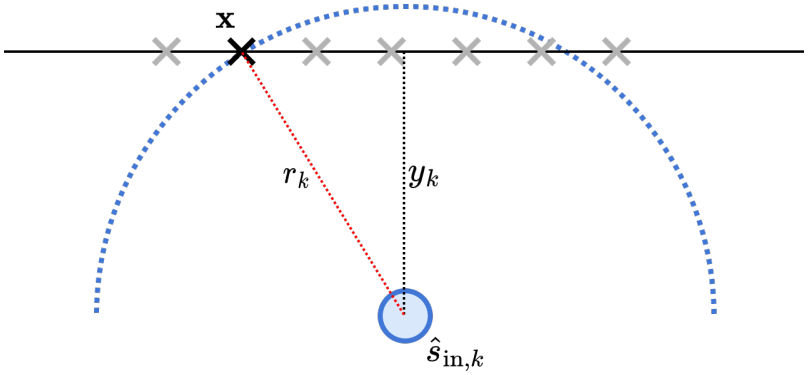


Figure 2.3: The driving signal for the WFS secondary source at position $\mathbf{x}$, for virtual primary source $\hat{s}_{\text{in},k}$, is dependent on the distance r_k of the primary source from the secondary source. This corresponds with a propagation delay via the simulated medium of propagation, coupled with a filter describing losses to that medium.

Focusing on the former kind, however, for m virtual sources, the time-domain driving signal $\hat{d}$ for the secondary source at $\mathbf{x}$ may be expressed as:

$$\hat{d}(\mathbf{x}, t) = \sum_{k=0}^{m-1} \hat{s}_{\text{in},k} * d_k(\mathbf{x}, t), \tag{2.4}$$

where the driving function d_k is[28]:

$$d_k(\mathbf{x}, t) = \frac{y_k}{r_k} f(t) * \delta\left(t - \frac{r_k}{c}\right). \tag{2.5}$$

The *WFS prefilter*[28] $f(t)$ is a function that aims to simulate the absorption of energy into the simulated medium of acoustic propagation. The delta function δ has the effect of delaying the prefilter, and thus $\hat{s}_{\text{in},k}$, by the time of propagation for a medium with propagation speed c (typically modelled as 343 m/s for sound in air). The components of the driving function are depicted in Figure 2.3.

2.4 Distributed Computing

In the broadest terms, a distributed system is *"a collection of independent entities that cooperate to solve a problem that cannot be individually solved"*[39]. In turn, the term *distributed computing* simply describes a system of computation that is distributed in space[40].[11] At a low enough level, this effectively describes *any* computer system, such systems being composed of individual entities — processors, memory, input and output devices, et cetera — all acting in cooperation.

At a higher level — that of a computer network, for example — why take a distributed approach to computation? As Kshemkalyani and Singhal describe[39], a variety of rationales exist for taking such an approach; adapting these to the notion of distributing an audio spatialisation algorithm across a computer network, most relevant are:

Scalability Particularly so if taking advantage of a network protocol that supports multicast or broadcast transmission. Under a unicast model, such as that employed by JackTrip, streams of audio data are duplicated on a per-client basis; under such a model, at some point all available network bandwidth will be exhausted. An ideal multicast networked audio system entails there being just one stream of audio data, plus perhaps a stream of control data, for all clients to consume. Dependent on the application, the server in such a system may not need to be aware of how many clients are connected; by a similar token, clients exist in isolation, and fulfil their task with no dependency on their peers on the network.

Modularity Closely related to scalability, modularity entails *extensibility*. This is something that centralised audio spatialisation systems either lack entirely, or possess only at great cost in terms of hardware, and even then only if the hardware supports extension via daisy-chaining, for example. The matter of expense anticipates:—

Improved cost/performance ratio A modular, scalable system can be constructed to meet the size that circumstances require, with the minimum amount of redundancy. If it becomes desirable to scale the system up, this can be achieved by small increments rather than by expensive leaps.

Kshemkalyani and Singhal also vaunt *enhanced reliability* as motivation for distributed computing. This may indeed hold true for systems where the failure of a single node can be compensated for by increased work on the part of the remaining nodes until such time as the failed node can resume operation or be replaced; it does not for the sort of system under consideration here. Indeed, this points toward two

[11] There is a certain linguistic symmetry here with respect to spatial audio, but that's about as far as the parallel goes.

significant drawbacks of distributed systems in general: increased complexity and a proliferation of potential points of failure. Nodes in a distributed computational system must be served at the very least with power and access (e.g. over a network) to the data that they require in order to operate. This entails the provision of cables, perhaps batteries, and physical connections that may be subject to wear-and-tear or misuse. Additional concerns surround the programmability of such a system, which is the other side to the coin of modularity; integrity, in terms of all nodes possessing up-to-date instructions for operation, may be difficult to ensure.

2.5 Distributed Audio Systems

The notion of taking a distributed approach to audio processing is by no means unprecedented; a selection of prior work in distributed DSP and audio spatialisation, plus systems incorporating microcontrollers and single-board computers is detailed below.

2.5.1 State of the Art

Applications of SoundWIRE to what its creators termed *Internet Acoustics*[15] obviously represent a case of distributed audio processing. These include implementations such as a network reverberator[41], or *"transcontinental echo chamber"*[13], plus the aforementioned *Network Harp*. Experiments of this sort were intended initially as sonifications of QoS — a characteristic of network systems that is difficult to represent in graphical or textual form due to the ephemeral nature of phenomena such as jitter and packet loss — but stand as fascinating applications in their own right of digital audio in the age of computer networking. Subsequent work on JackTrip has focused on optimising networked audio less for sound processing or as a creative tool in itself, and more in service of the social and communal aspects of music participation and appreciation in a networked world; these are topics that came to the fore in computer music research during the COVID 19 pandemic[42, 43].

Lago[44] proposed a UDP-based system for real-time distributed audio processing taking the form of a network of general purpose computers. A server sent packets of audio data to be processed by a collection of clients, which would then return processed audio to the server to be combined and used for output. Since clients were not to be used directly for output, synchronisation was not important, but Lago identifies the timing or hardware based interrupts for audio and network processing as being of great importance to a distributed real-time implementation. Though an interesting exploration of approaching certain difficulties of distributed computing, DSP in particular, and ambitious for its time (2004), arguably the need for such a system has been obviated by advances in computer processing power over the succeeding two decades.

A digital music production system of networked Beagleboard single-board computers was demonstrated by Gabrielli et al.[45] Another ambitious project, particularly since it relied on wireless communication, the authors describe an interesting way of measuring transmission round-trip times.

Exploring the possibilities of burgeoning network technology in the early 2010s, Lopez-Lezcano set out to build a UDP-based 'network sound card' to support a networked WFS system[46]. The aim was to replace otherwise expensive high channel-count conventional audio interfaces, which receive audio over the MADI protocol, with something more cost-effective. Ultimately the devised system was not used for audio spatialisation, but facilitated networked musical performance, and it stands an example of the results that can be achieved by using 'raw' UDP data for audio transmission, rather than an established protocol or system.

In addition to Gabrielli et al., implementations on IoT-like devices include Chafe and Oshiro's port of JackTrip to the Raspberry Pi single-board computer for further internet acoustics, plus distributed spatialisation systems such as those described by Devonport and Foss[47] and Belloch et al.[36] The latter two address aims closely aligned with the work described here, but are based on costly computing platforms. Devonport and Foss achieved high synchronicity via AVB; Belloch et al. used a GPU-based hardware platform, which is perhaps unsuited to its task, and report client synchronisation to the millisecond range — likely not sufficient for timing-critical audio spatialisation effects.

Also of interest is the OTTOsonics[48] project; its emphasis on a fully-costed, do-it-yourself alternative to conventional spatial audio systems is pertinent to this work, though it diverges in its use of AVB, and associated hardware for audio transmission.

2.5.2 Challenges

Time, especially when dealing with the fine margins posed by real-time audio processing, represents the principal source of difficulty in a distributed audio setting.

Jitter refers to fluctuations in the rate of transmission or processing. In a networked audio setting, jitter gives rise to a situation whereby the arrival of audio data does not correspond with the moments at which it is needed. in a naive implementation, this may result in a recipient either halting processing until it receives the expected data, or simply continuing without any data. In either case, the result is likely to be disruption of the integrity of the audio signal in the form of audible discontinuities.

Clock drift arises as an inevitable consequence of no source of time in a system of computation being perfectly uniform, and no two sources of time being identical. The timing of a computer system is typically governed by a crystal oscillator, the accuracy of which is affected by factors such as ambient temperature, and potentially computational load on the system it governs[49] Relative drift (sometimes, within the diagnostic parts of JackTrip for example, referred to as *skew*), is the difference in

clock rates between two or more systems. Whereas jitter is a short-term phenomenon, clock drift typically takes effect over a longer timescale. As two distinct systems of time move in and out of phase with each other over the longer term, drift may indeed give rise to jitter.

In studio and professional audio settings, devices may be synchronised via an authoritative clock source such as word clock, or, in a networked setting, via PTP or lower-resolution network time protocol. In the absence of such an authoritative source, e.g. over a wide area network, or if using hardware that does not support such measures, buffering strategies are typically employed, coupled with delay-locked loops[50] and resampling[51].

2.6 Research Questions

The primary focus of this thesis is an exploration of employing a network of hardware modules to act as a co-ordinated networked-audio system with a view to supporting a distributed approach to audio spatialisation techniques. It is hoped that in selecting a low-cost hardware platform, using ubiquitous computer networking technologies, and keeping the overall design as simple as possible, the work here could lead to a lowering of the barrier to entry of the creation of large scale spatial and immersive audio installations. The first research question, then, is:

> **Research Question 1**
> *How can a network of microcontrollers be used to create a distributed audio system suitable for managing scalable installations for spatial and immersive audio?*

Adjacent to the bare bones of implementation is the performance of the devised system:

> **Research Question 2**
> *How can such a system be created and configured such that it maintains inter-client synchronicity? What are the effects of loss of synchronicity on distributed spatialisation algorithms?*

2.6.1 Prior Work

Before beginning work on this project, a distributed WFS system was developed, based on the Teensy 4.1 microcontroller and JackTrip protocol. This consisted of an audio application running on a general purpose computer, delivering audio over

JackTrip to a collection of hardware modules, and WFS control data over multicast UDP. Strategies were developed for mitigating the effects of jitter and clock drift, and these serve as the basis for work on the project described in this thesis.

The prior system was not formally evaluated, but, anecdotally, provided support for the holophonic effect of primary-source WFS. These strategies were initial efforts, based on a limited understanding of the problem-space, and called for further development. It was deemed, also, that the implementation was constrained by the more opinionated aspects of the JackTrip system.

A paper documenting the work on the JackTrip-based approach can be found in appendix A.

Chapter 3

Development

To address the research questions posed in section 2.6, a networked audio system was developed. Being a system distributed across distinct computing platforms (a general purpose computer; a network of microcontrollers), and software elements serving a variety of purposes (server and client instances for transmission and reception of networked audio and control data, plus a digital signal processing algorithm), it is important to consider each of these elements in detail. In the sections to follow, these elements are described; finally an overview of the devised system is provided.

3.1 The Networked Audio Server

Prior work having been predicated on the development of a JackTrip client for the Teensy platform, a principal aim of conducting further work on the server part of the networked audio system was to establish whether improvements could be found by addressing certain shortcomings of JackTrip. JackTrip, running in hub server, is a hybrid system, based on both TCP and UDP; clients initiate a connection with a JackTrip server by way of a TCP handshake, whereby client and server exchange the port numbers that will subsequently be used for audio transmission over UDP. TCP is, as described in section 2.1.1, a connection-based, one-to-one protocol, so the JackTrip connection model enforces a sort of pseudo-connectionfulness on the otherwise connectionless UDP. The result is a system which permits only unicast UDP transmission, and, for multiple clients, must send a duplicate of the outgoing stream of audio datagrams to each connected client. A JackTrip server creates two instances of its `UpdDataProtocol` class per client: a sender and a receiver, each of which is a thread of execution[20]. Modern computer systems may be able to contend quite contentedly with the operation of large numbers of threads, and bandwidth over a LAN may be plentiful, but ultimately a unicast system does not meet the requirement of scalability (see section 2.4).

With a view to exploiting the UDP multicast capabilities of NetJACK, initial

efforts centred on attempts to create a minimum-viable NetJACK client for the Teensy platform. Beyond making initial contact with a NetJACK server, however, these did not bear fruit. In the absence (to the best of the author's knowledge) of readily-available documentation of the NetJACK protocol, work on a Teensy-based client relied upon reverse engineering the JACK2 codebase, running a debuggable instance of `jackd` and scrutinising Wireshark captures for clues. Ultimately, although representing the promise of a multicast networked audio server, work on NetJACK was abandoned for two main reasons: 1) NetJACK was created as a general-purpose system, with functionality beyond the requirements of the proposed implementation; 2) JACK has not served as a truly cross-platform audio host for a number of years; although `jackd` runs on Mac OS X systems, not since OS X 10.14 has JACK been compatible with Mac's CoreAudio API, thus any server implementation based upon NetJACK would not be operable on a modern Mac operating system.[1]

Of course, being inherently tied to JACK as its audio host, the latter rationale applies also to the JackTrip-based approach. This being the case, attention was turned to the design of a bespoke multicast networked audio system.

3.1.1 Designing a Networked Audio Protocol

To take a true no-protocol approach, such as that described by Lopez-Lezcano[46], is not without its drawbacks. Ironically, for a no-protocol system to work, the rubric that ensures successful data encoding, transmission, and decoding must be understood implicitly by all members of the network; i.e. there *is*, in fact, a protocol, but that protocol is an unspoken contract. Dependent on the intended application, and if assumptions can be made about matters such as bit resolution, this may be a sensible design choice, but to make the new protocol somewhat flexible and future-proof, a simple packet header was devised. Its structure is given in listing 3.1.

Listing 3.1: Packet header structure

```
struct PacketHeader {
    uint16_t SeqNumber;
    uint8_t BufferSize;
    uint8_t SamplingRate;
    uint8_t BitResolution;
    uint8_t NumChannels;
};
```

The resulting six-byte header is comprised, then, of a two-byte (unsigned 16 bit integer) packet sequence number, to be incremented by the sender, plus four further

[1] For more information, see Stéphane Letz's proposed design for a successor to the defunct CoreAudio/JACK bridge `https://github.com/jackaudio/jack-router/blob/main/macOS/docs/JackRouter-AudioServerPlugin.md` (accessed 08/05/2023).

bytes describing the structure of the audio data in the packet. Of course, commonly-encountered sampling rates, and buffer sizes greater than 255, cannot be represented by unsigned eight-bit integers, so these are backed up by enumerations inspired by those used by JackTrip[2]:

Listing 3.2: Supporting enumerations for packet metadata

```
enum BufferSizeT {
    BUF8 = 3,
    BUF16,
    BUF32,
    BUF64,
    BUF128,
    BUF256,
    BUF512,
    BUF1024,
    BUF2048,
    BUF4096
};
```

```
enum BitResolutionT {
    BIT8 = 1,
    BIT16,
    BIT24,
    BIT32
};
```

```
enum SamplingRateT {
    SR22,
    SR32,
    SR44,
    SR48,
    SR88,
    SR96,
    SR192
};
```

Thus a buffer size of 16 is represented by the number 4, which is the appropriate power of two to use in converting between the enumeration and the number with which it corresponds. Similarly, a bit resolution of 16 is associated with the number 2, which is of course the number of bytes in a 16-bit integer.

For well-formed packets, `BufferSize` could in fact be inferred from the size of the packet (minus its header), divided by `NumChannels` and `BitResolution`, the latter treated as the number of bytes per sample (as per the enumeration value). To permit scope for the detection of malformed packets, however, the expense of an additional byte in the header was deemed a reasonable one. The sequence number will wrap around every 65,536 packets, and is intended as a means for a recipient to identify the occurrence of packet loss.

3.1.2 Server Design

The networked audio server was written in C++ using utility classes provided by the JUCE framework for the development of audio applications.[3] The server is encapsulated as a class called `NetAudioServer`, which can be incorporated into any JUCE-based audio application; initial development was conducted on a basic console application, and later work targeted a DAW plugin supporting as a consolidated audio server and wave field synthesis controller (see section **??**).

`NetAudioServer` expects to receive blocks of multichannel audio from an audio

[2] `https://github.com/jacktrip/jacktrip/blob/v1.6.8/src/AudioInterface.h#L56`
[3] JUCE 7.0.5 `https://github.com/juce-framework/JUCE`

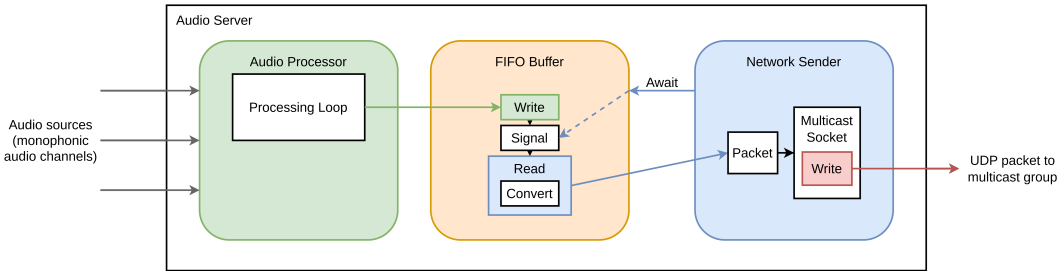


Figure 3.1: Overview of operation of the networked audio server. The network sender awaits notification of readiness to read samples from a first-in-first-out buffer of audio samples. The audio processor receives audio channels from a multichannel source (e.g. a DAW); at each iteration of its processing loop, it writes samples to the FIFO; upon write-completion, the FIFO sends a signal to the network sender that a block of samples is ready. Samples are converted to the appropriate format, and bundled into a UDP packet which is then written to the network.

application's main processing loop. It sets up *sender* and *receiver* execution threads, and assigns a network *socket* to each; a socket is essentially a numerical identifier for an *"endpoint for [network] communication"*[52] to which a type, such as `SOCK_STREAM` for TCP or `SOCK_DGRAM` for UDP, can be assigned. To avoid potentially blocking the processing thread with networking operations, upon receiving an audio block the server writes it to an intermediate buffer — a first-in-first-out (FIFO) structure — and signals to the sender thread that an audio block is ready for transmission. The sender thread, which has been awaiting such a signal, then requests samples from the FIFO; these are stored as contiguous channels of 32 bit floating point samples and converted when requested to the bit resolution specified in a packet header created when `NetAudioServer` is initialised. Upon receiving the requested samples, the sender thread writes these to its socket, which has been configured to connect to a UDP multicast group.

Listing 3.3 shows an example network capture of an outgoing UDP audio packet:

Listing 3.3: Network capture of a UDP audio packet

```
       00 01 02 03 04 05 06 07  08 09 0a 0b 0c 0d 0e 0f

0000   01 00 5e 04 e0 04 a0 36  bc d0 aa 18 08 00 45 00   ..^....6......E.
0010   00 62 8b b5 40 00 01 11  63 1a c0 a8 0a 0a e0 04   .b..@...c.......
0020   e0 04 39 f9 a3 56 00 4e  66 2b df 1c 04 02 02 02   ..9..V.Nf+......
0030   3f 7a 40 7a 41 7a 42 7a  43 7a 44 7a 45 7a 46 7a   ?z@zAzBzCzDzEzFz
0040   47 7a 48 7a 49 7a 4a 7a  4b 7a 4c 7a 4d 7a 4e 7a   GzHzIzJzKzLzMzNz
0050   2c 8c ff 88 43 86 05 84  46 82 00 81 44 80 00 80   ,...C...F...D...
0060   45 80 ff 80 46 82 06 84  41 86 02 89 29 8c db 8f   E...F...A...)...
```

Bytes `0x0000` to `0x0029` comprise the ethernet, IPv4, and UDP headers, including the destination address: at position `0x001e`, the bytes `0xe004e004`, or `224.4.224.4`,

a valid (and unassigned) UDP multicast address from the second ad-hoc address block as specified in the IANA multicast address assignment guidelines[19]. The six subsequent bytes are the header inserted into the packet by `NetAudioServer`. In listing 3.3 these are:

- `0xdf1c`: a sequence number of 57116_{10};
- `0x04`: buffer size 4_{10} corresponding with `BUF16`;
- `0x02`: sampling rate 2_{10} corresponding with `SR44`;
- `0x02`: bit resolution 2_{10} corresponding with `BIT16`;
- `0x02`: 2_{10} audio channels.

Audio data begins at byte `0x0030`. Since the header indicates that there are two channels of 16-bit audio, and a buffer size of 16 (samples, or rather frames of two channels worth of samples), it is clear that the data for channel 1 encompasses the 32 bytes from `0x0030` to `0x004f`, and channel 2 the remaining bytes.

Here, channel 1 is a test signal, a unit amplitude-increment unipolar sawtooth wave, i.e. a signal whose amplitude starts at zero, and increments by 1 at each sample until it reaches the maximum value that a signed 16-bit integer may take — 32767_{10} — at which point it wraps around to zero and repeats. This test signal serves two important purposes. First, its impulse-like behaviour once every 32767 samples (roughly 0.74 s) is useful for taking basic synchronicity measurements, e.g. involving connecting two clients' audio outputs to an oscilloscope. Second, this numerically-predictable signal served as a means to inspect the integrity of the audio server algorithm, and to identify the appropriate endianness for transmission. Inspecting the first sixteen samples of the first audio channel (grouped by sample for legibility, see listing 3.4), it is evident that the amplitude values increment on a per-sample basis, and, since it is the first byte that increases with each sample, that samples are transmitted little endian. This proved to be the appropriate endianness for the system under development.

Listing 3.4: Extract of samples from the unipolar sawtooth wave

```
3f 7a   40 7a   41 7a   42 7a   43 7a   44 7a   45 7a   46 7a
```

Mention numerical range of signed 16-bit integers?

The receiver thread polls its socket for traffic reaching the multicast group from clients, and uses the presence of an incoming packet stream as proof that a client is connected.

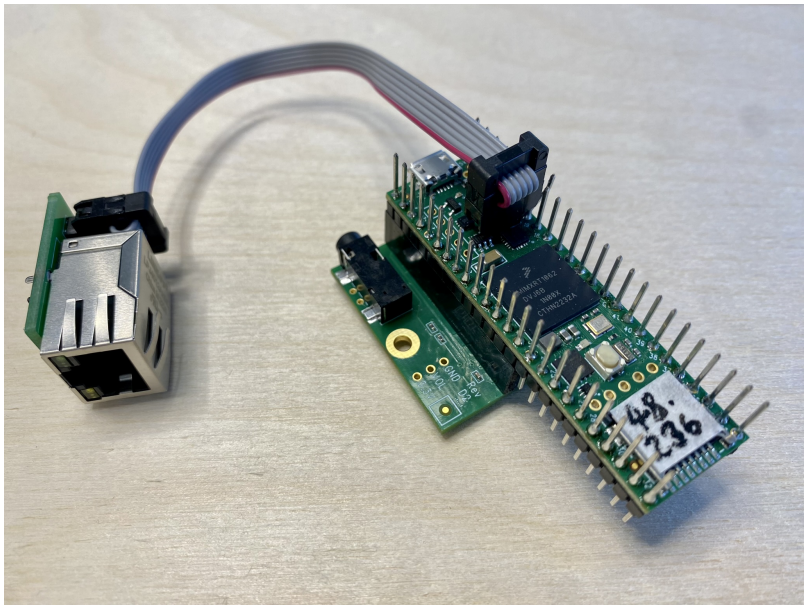


Figure 3.2: A hardware module consisting of Teensy 4.1 microcontroller (labelled with the last two bytes of its serial number-derived IP address), connected via headers to an audio shield and via ribbon cable to an ethernet shield.

3.2 The Networked Audio Client

Unlike the networked audio server, which runs on a general purpose computer and has access to threads of execution, which it can use to conduct related but separate tasks that rely on some central resource (the FIFO buffer alluded to above), the client implementation is designed to operate on a microcontroller platform that has no operating system, and no native notion of threads[4].

The task of the network audio clients is threefold in nature:

1. to retrieve packets of audio data from the UDP multicast group;
2. to send a stream of audio data back to the multicast group, primarily to announce their connectivity;
3. to maintain, as far as possible, synchronous operation with the server, and (by extension) each other.

To address the first two requirements, the client sets up a socket, which it uses to both read from and write to the UDP multicast group.

[4]There is in fact a non-core library, *TeensyThreads*, that provides thread-like functionality. It was experimented with during development, but found to be incompatible with the interrupt-driven nature of the Teensy audio and networking libraries.

The client was created as a C++ class named `NetJUCEClient`, an implementation of the Teensy audio library class `AudioStream`. `AudioStream` descendents must implement a method named `update()`; this method is called at each audio hardware interrupt, and is where an audio library class should perform operations on the current audio buffer. Due to the regular timing of calls to `AudioStream::update`, and the correspondence between an audio buffer and a network audio packet, it was tempting to use that method, and thus the underlying audio interrupt as an opportunity to handle networking operations too. This proved unreliable, however; conflicts between the audio and network interrupts caused the Teensy to crash sporadically.

Instructions to read from and write to the network were moved to a method, `NetJUCEClient::loop`, which is called from the top level loop of the Teensy program. Although not called at regular intervals — Teensy's `loop()` function is itself executed from the body of a non-terminating `while` loop — tests indicated that the time between iterations lay on the order of tens of microseconds, far lower than the audio interrupt interval.

Separating audio and networking operations to `update()` and `loop()` methods respectively, the two sets of operations were linked by way of an intermediate buffer, similar to the FIFO used on the server. Thus the client attempts to receive packets from, and send a packet to, the multicast group on each call to `loop()`, with incoming packets written to the intermediate buffer:

Listing 3.5: loop method of the networked audio client implementation

```
void NetJUCEClient::loop() {
    receive();

    checkConnectivity();

    send();

    adjustClock();
}
```

The client also performs a periodic check for the presence of the server (and potentially other clients — see section 3.2.1), and, as described in section 3.2.1, makes adjustments to its audio clock.

On the audio interrupt, the client reads from the intermediate buffer to produce samples for audio output. It also takes samples reaching its audio inputs (e.g. arriving from other Teensy audio library classes), and adds those to a packet to be sent to the multicast group at the earliest possible subsequent call to `NetJUCEClient::loop`:

Listing 3.6: `update` method of the networked audio client implementation

```
void NetJUCEClient::update(void) {
    doAudioOutput();

    handleAudioInput();
}
```

3.2.1 Synchronicity with the Server

Due to the influence of clock drift and transmission jitter, and since the clients constitute a distributed system, with no direct knowledge of each other and no authoritative source of time, their third task posed, without doubt, the greatest challenge.

A two-pronged strategy was developed for addressing server-client and inter-client timing discrepancies:

Jitter Compensation Similar to the approach taken in prior work[53], clients monitored the difference between the write and read positions to their intermediate audio buffer, using a delay-locked loop to keep this difference within an interval of one audio buffer. This was achieved by way of setting thresholds for the read-write difference, and adjusting the read-position increment if it fell beyond those thresholds; increasing the increment if the difference exceeded the high threshold; decreasing it should the difference fall short of the low threshold. This in turn entailed employing a fractional read-position, and interpolating around the read-position to achieve an appropriate sample value; essentially a form of adaptive resampling. For this purpose a cubic Lagrange interpolator was used; sample values for the interpolator were converted from their 16-bit signed integer representation to floating point numbers, interpolation conducted, and the resulting value rounded to the nearest integer for output.

Clock Drift Compensation In the absence of an authoritative source of time, clients were set up to infer the difference in rate between their own internal clock and that of the server by comparing the rate of packet reception from the network to their internal audio interrupt rate. This was achieved by taking the ratio, over thirty-second intervals, of packets written from the network to the intermediate buffer to blocks read from the intermediate buffer for audio output. This ratio was then used to calculate appropriate divisors to apply to the 24 MHz master clock generated by a crystal oscillator on the Teensy, adjusting the audio clock's phase locked loop to produce an adjusted audio sampling rate. The aim of this approach was to minimise reliance on the adaptive resampler described above, and ultimately encourage all clients to run at the same audio rate as the server.

An Unexpected Source of Jitter

Jitter does not arise solely during the journey from machine to machine across a network. It was found that, using the default development machine's default audio host (ALSA), the timing of iterations of JUCE's audio processing loop, and thus signals to `NetAudioServer`'s sender thread, was very uneven, i.e. subject to a high degree of jitter. Though the average rate of execution was commensurate with the selected sample rate and audio buffer size, individual audio buffer intervals differed, in microsecond terms, by up to an order of magnitude. This inconsistent audio timing was sufficient to cause transmission jitter that overwhelmed the client-side strategy for jitter compensation. The result was consistent audible distortion due to rapid fluctuations in the client's audio buffer read-position increment.

Switching from ALSA to JACK as the audio host resolved this issue, but since JACK *uses* ALSA as its underlying audio host, this phenomenon was difficult to account for. A detailed description lies beyond the scope of this report, but essentially JACK uses *direct memory access* to map the underlying audio device into memory[5], and seemingly does so in a more effective manner than the driver for the sound card on the development machine. It is not known at the time of writing whether using ALSA with an external audio interface (a potential barrier to entry) would have improved the situation.

This incident serves to illustrate that, beyond the concerns associated with effective software design, a system such as the one described here, for which the timing of operations is critical, is susceptible to the whims of a variety of supporting hardware and software systems — things that in a more conventional audio system can, to a greater degree, be taken for granted. The topic of whether to attempt to compensate for this sort thing in software or impose strict requirements on a future end-user in terms the audio host they should use (something that would almost certainly vary with the user's operating system), remains one for future research.

A Note on the Client-Server Dichotomy

In principle, there is no meaningful impediment to the so-called clients acting as servers in their own right; a change of port number is all that would be required for each client to receive traffic from (and send traffic to) not just the server, but also from (and to) all other clients. Consequently, in code for both the server and clients, networked audio entities in the system are referred to as *peers* (see, e.g. Figure 3.5). In practice, and for the particular application — distributed spatial audio — considered here, receiving UDP packets from sources other than the designated server is not only unnecessary, but also manifestly inefficient. With Chafe et al.'s *Internet Acoustics* work in mind, it is likely that there are some very interesting uses for a *promiscuous*

[5]Credit goes to Stéphane Letz, a prominent contributor to the JACK project, for providing this bit of context.

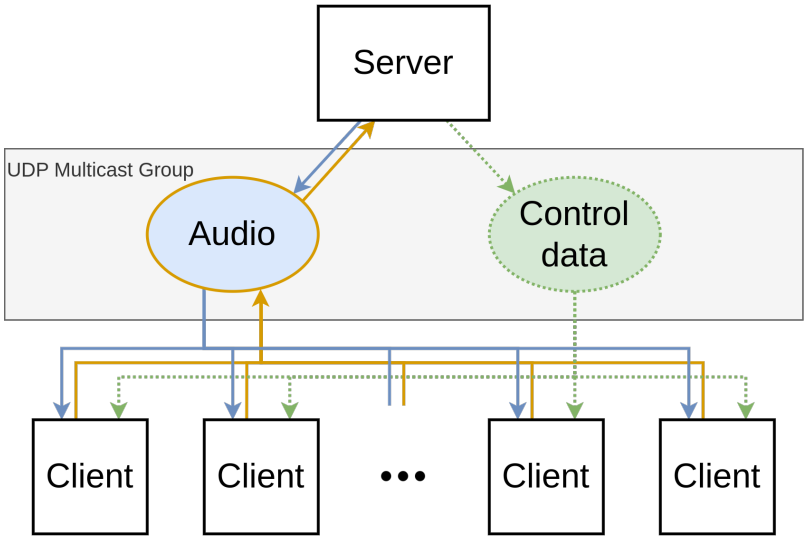


Figure 3.3: Architecture of the networked audio system. The server sends audio and control data to a UDP multicast group on distinct port numbers; clients listen for this data, and send an audio stream back to the multicast group with which the server can register their presence on the network.

network of audio peers, but, for now, such applications lie beyond the scope of this work.

3.3 The Spatialisation Algorithm

With some minor modifications, e.g. the possibility to specify speaker spacing parametrically, the WFS algorithm from[53] was reused. This algorithm was written in Faust, a domain-specific programming language for audio synthesis and signal processing[6], and compiled to a C++ class compatible with the Teensy audio library via Faust's `faust2teensy` utility[54].

Implementing Huygens' principle in a digital audio system entails applying delays to an audio signal representing a virtual sound source based on the intended position of that sound source with respect to the secondary point sources. To limit the computational burden placed on the hardware modules, specifically with regard to memory, the length of the delay lines was reduced by discarding the longitudinal component of r_k, leaving only the relative inter-speaker delay. Additionally, a simplified WFS prefilter was employed, using the distance from each virtual sound source to each secondary point source to an inverse square law mapping for frequency-independent amplitude loss to the medium of propagation, and the cutoff of a two-pole lowpass filter. Adopting a modified version of equation (2.5), the driving

[6] `https://faust.grame.fr/`

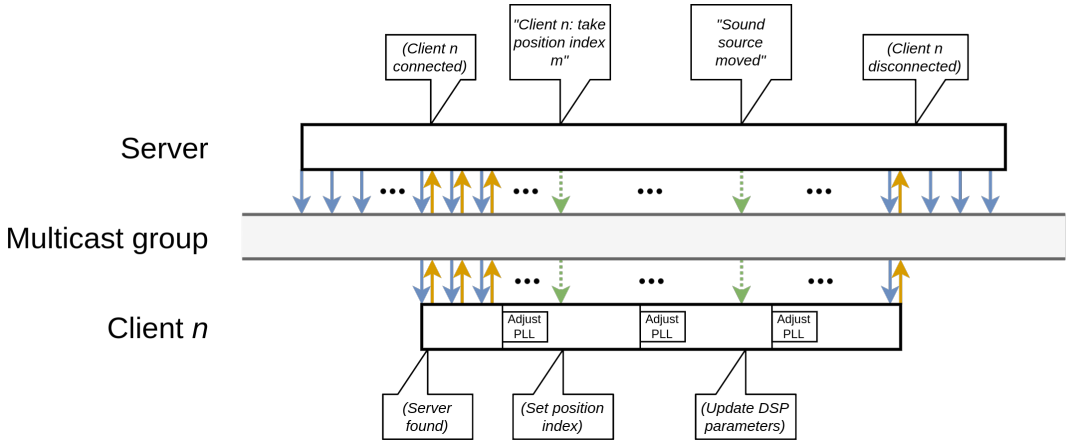


Figure 3.4: Example timeline of client-server interaction. Blue arrows indicate audio data being sent from the server to the multicast group; orange arrows indicate audio data being sent from the client back to the multicast group; green arrows represent control data.

function becomes:

$$d_k(\mathbf{x}, t) = f(t, r_k) * \delta\left(t - \frac{r_k - y_k}{c}\right). \tag{3.1}$$

In implementation, the prefilter was tuned by ear using Faust's `fi.lowpass` function[7]; a thorough treatment of the simulation of distance effects in WFS stands as a topic for future work.

Modularity and Maximum Delay

The reduction in the maximum delay length represented by the subtraction of the longitudinal distance component in equation (3.1) is essential for the viability of the system. As capable a platform as Teensy 4.1 is, as described in section 2.2, it is limited in terms of memory. This in turn places limits on the lengths of delay lines that it can compute, a matter exacerbated if there are many such delays to consider, such as in the case of a WFS implementation with multiple virtual sound sources. Each hardware module must compute two delay lines for each virtual source, one for each of its output channels, the maximum length of which (depending on the position of a given module in the speaker array) corresponds, after removal of the longitudinal component, of the width of the speaker array. It was observed that, for eight virtual sources and eight hardware modules, the maximum speaker spacing permissible lay around 0.425 m, corresponding with a speaker array of maximum width $15 \times 0.4 = 6$ m, equating to a maximum delay of ~17 ms or approximately 795 samples at a sampling rate of 44.1 kHz. The matter has not been rigorously tested, but nonetheless the presumption is that this places significant limits on the modularity of the system.

[7] `https://faustlibraries.grame.fr/libs/filters/#filowpass`

It is hoped that future versions of the Teensy platform, or developments in other microcontroller platforms, overcome this manner of limitation.

Controlling the WFS Algorithm

Parameter values are delivered to the Faust algorithm in the form of OSC messages. OSC control data, describing virtual sound source positions, speaker spacing, and informing clients of their position in the speaker array, is bundled into UDP packets and delivered by the server to the multicast group for all clients to consume.

Listing 3.7: Network capture of a UDP control data packet

```
       00 01 02 03 04 05 06 07  08 09 0a 0b 0c 0d 0e 0f

0000   01 00 5e 04 e0 04 a0 36  bc d0 aa 18 08 00 45 00   ..^....6......E.
0010   00 44 54 23 40 00 01 11  9a ca c0 a8 0a 0a e0 04   .DT#@...........
0020   e0 04 39 f9 a3 57 00 30  4d 60 23 62 75 6e 64 6c   ..9..W.0M`#bundl
0030   65 00 00 00 00 00 00 00  00 01 00 00 00 14 2f 73   e............./s
0040   6f 75 72 63 65 2f 30 2f  78 00 2c 66 00 00 3d 1b   ource/0/x.,f..=.
0050   5f a2                                             _.
```

Listing 3.7 demonstrates an example control data packet, an OSC bundle containing one message. This message has address `/source/0/x`, indicating that it refers to the x-coordinate of the zeroth sound source, providing a value in the form of a 32-bit floating point number, `0x3d1b5fa2`, approximately 0.038_{10}.

3.4 System Overview

Being composed of multiple hardware and software components, it is worthwhile to summarise the nature of these components and the interactions between them.

3.4.1 Hardware Setup

The network audio server runs on a general purpose computer. During the development and testing of this project, that computer was an ASUS G513R Notebook PC, with an AMD Ryzen 7 6800H processor with a clock speed of 3.2 GHz. For the majority of development, the computer's internal sound card was used; for testing and evaluation, it was connected to a Steinberg UR44C USB audio interface in the hope that this would provide better reliability in terms of audio interrupt timing.

The computer was connected via CAT6 ethernet cable to an eight-port ethernet switch (D-Link DGS-108GL). For evaluation, and to support a total of eight network audio clients, this switch was daisy-chained to an additional switch (D-Link DES-1008D). Teensy 4.1 hardware modules, assembled as per Figure 3.2, were connected

via CAT6 ethernet cables to available ports on the ethernet switches. Hardware modules were powered by a combination of a seven-port USB hub, plus, for the eighth module, a USB mains socket. The two audio outputs of each hardware module were connected to M-Audio BX5 speakers.

3.4.2 Software System

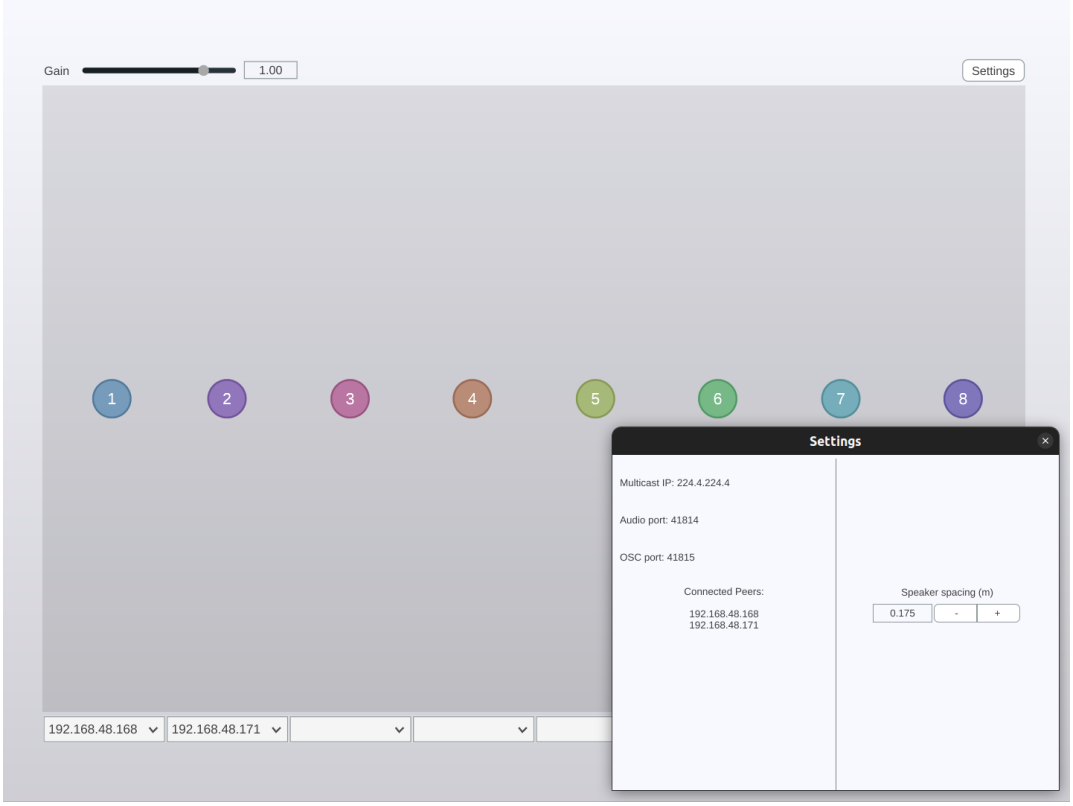


Figure 3.5: User interface for the WFS controller DAW plugin, with modal settings window visible. The interface consists of an X/Y control surface, with eight nodes representing the locations of sound sources in a virtual sound field. Dropdown menus at the bottom of the interface correspond with hardware module positions in the loudspeaker array. The settings window facilitates setting the speaker spacing, and shows a list of connected network peers.

Server-side, the software system consists of a VST plugin running in Reaper digital audio workstation software[8]. The plugin comprises the networked audio server, receiving monophonic audio sources in the form of audio or instrument tracks in the DAW, plus a control data server, commanded either by parameter automation via the DAW, or manually via a graphical user interface (see Figure 3.5). The audio and control data servers send streams of UDP packets to a UDP multicast group.

[8]https://www.reaper.fm/

Client-side software connects to the multicast group and reads UDP packets containing audio and control data from the server. These streams are delivered to the Faust-based WFS algorithm, with audio streams processed according to the control parameters of virtual sound source positions and speaker spacing. The WFS algorithm produces driving signals for each of the two output channels of the hardware module on which it is running. Additionally, the client-side networked audio client returns a stream of audio data to the multicast group, to be consumed by the server.

Code for the server and client software components can be found at `https://github.com/hatchjaw/netjuce` and `https://github.com/hatchjaw/netjuce-teensy` respectively[9].

[9]At the time of writing both codebases are undocumented, but the key details of implementation lie in each repository's `src/` directory.

Chapter 4

Evaluation

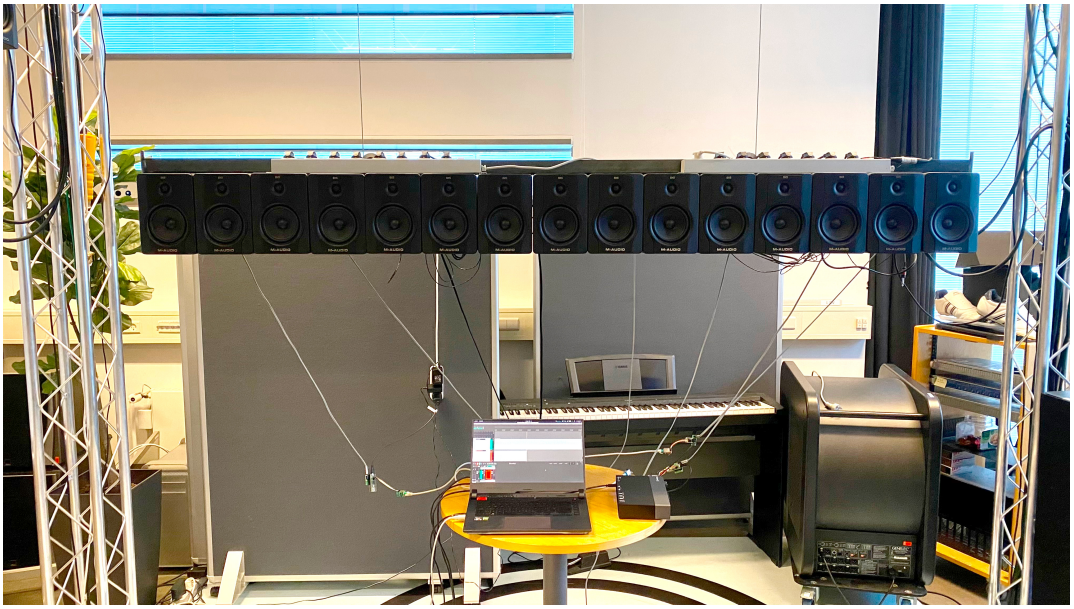


Figure 4.1: System configuration for technical and perceptual evaluation. Eight hardware modules connected to fifteen loudspeakers — seven of the modules produced output for two loudspeakers each; the final module used only its first output channel.

Possessing technical underpinnings, but ultimately being designed to serve immersive auditory ends, it was important to consider the performance of the system described and developed in chapter 3 in terms of both its technical capabilities and the quality of the perceptual effects it was able to support. The success of the system as platform for audio spatialisation techniques is contingent on it being composed of effective solutions to the challenges posed by distributing audio processing across a local area network. It makes limited sense, however, as a technical exercise in isolation; the subjective assessment provided by listeners exposed to its output may help

identify the most critical aspects of the technical implementation and guide future development.

4.1 Technical Evaluation

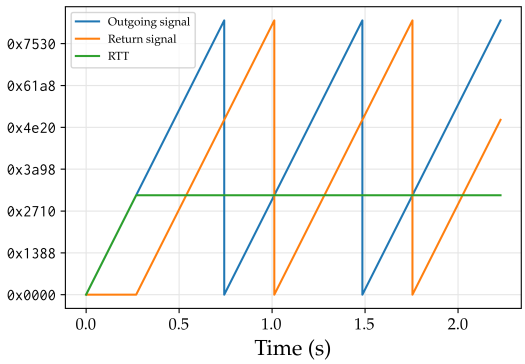


Figure 4.2: Illustration of the use of a test signal, a unipolar sawtooth wave, to measure round trip time. Subtracting the return signal from the outgoing signal gives the time (in samples) between transmission and reception.

Of most pressing technical concern is the matter of synchronicity between the hardware modules. To assess this, a similar approach was taken to that found in [53] and [45]:

Round Trip Time To measure transmission round trip time (RTT), the server transmitted a unipolar sawtooth wave of unit amplitude increment to the multicast group, and each client, upon receiving that signal simply returned it immediately to that same group to be consumed by the server. At the server side, the return signal, x_{ret}, was subtracted from the outgoing signal, x_{out}, at the time of reception, with round trip time found as:

$$\text{RTT} = \begin{cases} (x_{\text{out}} + \max_{\text{int16}}) - x_{\text{ret}}, & x_{\text{out}} < x_{\text{ret}}, \\ x_{\text{out}} - x_{\text{ret}}, & \text{otherwise}, \end{cases} \tag{4.1}$$

where $\max_{\text{int16}}$ is the maximum value representable by a signed 16-bit integer, `0x7fff` (32767_{10}).

The vagaries of buffering at the level of the server's audio driver aside, the resulting value should provide an accurate impression of the number of samples that elapse between transmission and reception (see Figure 4.2). Since there is one source of transmission, for multiple clients, comparing RTT offers a means to assess inter-client synchronicity. Server-to-client latency cannot be measured in this way,

but that can be inferred to be around half of, and, of course, certainly not greater than, the RTT.

Clock Drift/Skew A unipolar sawtooth wave of unit amplitude increment was generated on the clients, subtracted from the incoming sawtooth wave from the server, and the difference (found similarly to equation (4.1)) returned to the multicast group for consumption by the server. Under ideal conditions, the incoming signal and the one being generated on a given client, while not necessarily (and almost certainly not) synchronised, should be out of phase by some constant value; if this value changes then some relative drift has occurred between server and client. While not of direct relevance to synchronicity, the client-side clock-adjustment strategy described in section **??** was designed to minimise the reliance on the adaptive resampling approach that it complements; low (and ideally *no*) drift would be indicative of the effectiveness of that strategy.

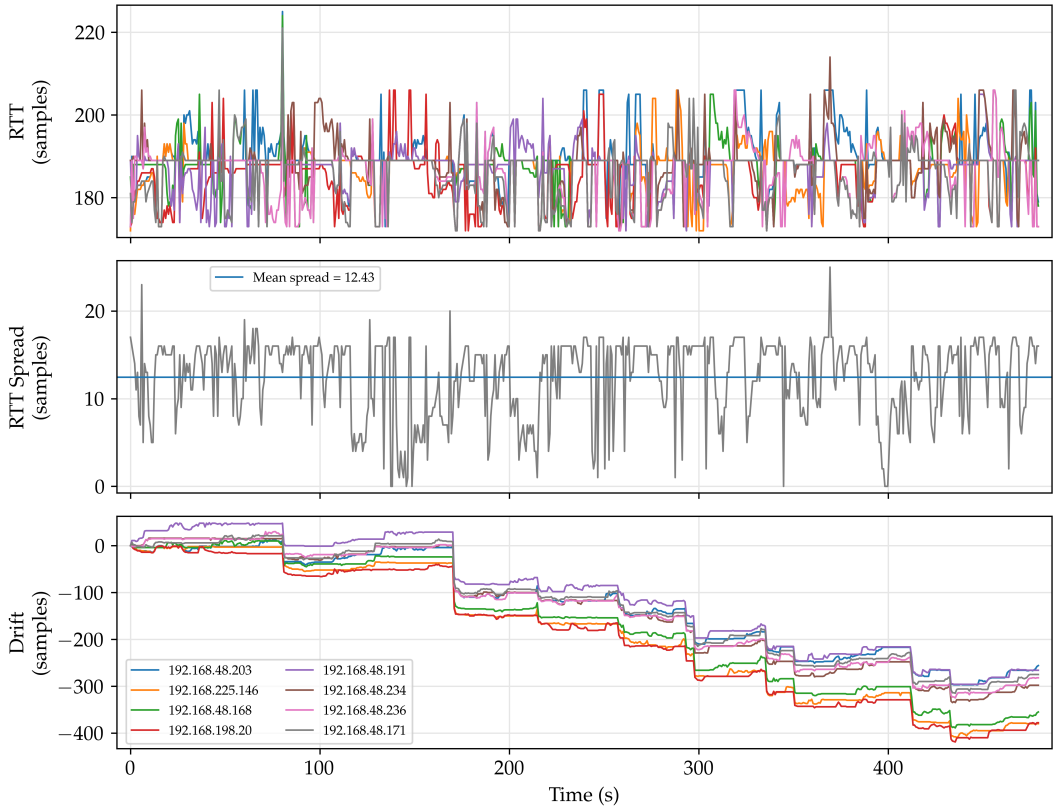


Figure 4.3: Round-trip time, round-trip time spread, and clock drift measurements for eight networked audio clients, for a networked audio session of eight minutes' duration. Audio buffer size, 16 samples.

Initial RTT and relative drift measurements for eight clients are shown in Figure 4.3.

Mean RTT spread, describing the average temporal interval over which clients were distributed over the course of the test, is promising, the 12.43 sample interval corresponding to approximately 282 µs. RTT, and thus maximum latency, is clustered around a respectable 190 samples (~4.3 ms).

That visual clustering, coupled with the apparent tendency for RTT spread to lie at around 16 samples (i.e. precisely one buffer), suggest, however, a certain over-aggressiveness in the resampling strategy, perhaps resulting in a polarisation of clients to the temporal extremes of the interval between their audio interrupts. What Figure 4.3 does not show, and, given the short timescales involved, is not easily represented in such a diagram, is the rate of relative inter-client movement, i.e. the rate of change of asynchronicity. Cursory, subjective assessment of the system's audible output revealed that, given the rapid rate of relative movement between clients, in this state it would not stand up to perceptual testing.

Transmitting a signal consisting of white Gaussian noise (WGN) to the clients and delivering this to their audio outputs without further processing — seeking, essentially, to sonify QoS — an aggressive phasing, or time-varying comb-filter effect was clearly audible. This effect is visualised in Figure 4.5(a); ideally (and subject to the frequency of the response of the microphone used) an ambient recording of a white noise source would correspond with a magnitude spectrogram exhibiting equal intensity across the frequency range at all times; clearly, though, there are regions of greater and lesser intensity, and these regions shift and change rapidly over time. In addition to the above, tests involving the reproduction of signals containing steady-state harmonic content revealed obtrusive audible artefacts.

A buffer size of 16 samples had been selected in an attempt to minimise the duration of the window of inter-client synchronicity, and to maximise the number of channels that could be transmitted over the network, subject to restrictions posed by the MTU (see section 2.1.2). Recalling, however, that previous work[53] had employed a 32-sample audio buffer, equivalent measurements were taken for the larger buffer size, the results of which are depicted in Figure 4.4 and Figure 4.5(b).

Again, visually, there is an apparent clustering in the RTT recordings, with clients spending large periods separated by around one buffer's worth of samples (~726 µs), seemingly often grouped at either extreme of the interval of one audio buffer. The mean RTT spread, ~626 µs, is comparable with results from prior work, but, disappointingly, slightly less favourable. Importantly, however, and as demonstrated in Figure 4.5(b), the rate of relative inter-client temporal movement was much improved by the switch to a 32-sample buffer. Although exhibiting similar visual striations to the spectrogram for the test at 16 samples, fluctuations occur less frequently, and seemingly more gradually. Indeed, subjectively-speaking, the disruption caused by the phasing effect that afflicted the 16-sample buffer implementation was significantly reduced, as was the presence of audible artefacts affecting harmonic signals. Thus it was the version of the system employing a buffer

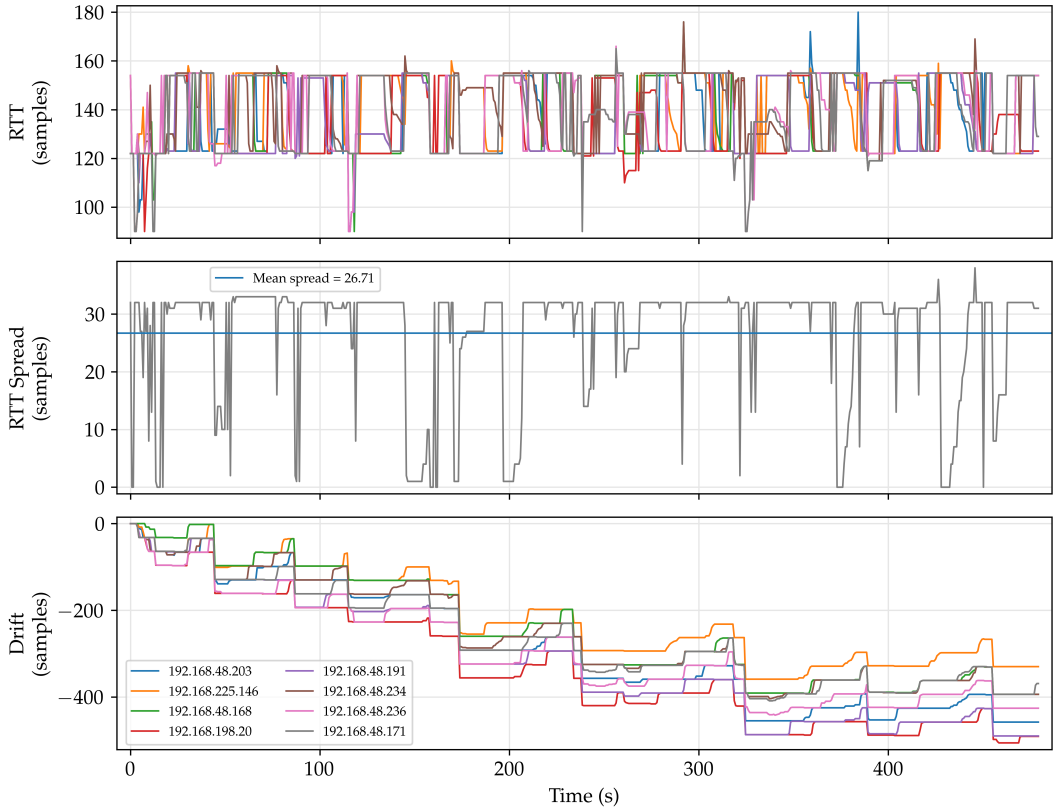


Figure 4.4: Round-trip time, round-trip time spread, and clock drift measurements for eight networked audio clients, for a networked audio session of eight minutes' duration. Audio buffer size, 32 samples.

size of 32 samples that was exposed to perceptual evaluation.

Clock drift measurements in Figures 4.3 and 4.4 exhibit comparable trends. Increasing negative drift over time is indicative of the clients running faster than the server. Visually, there is evidence that client clocks adjust to approximate parity with the server for periods of time, perhaps falling slightly slower (e.g. the drift plot in Figure 4.3, between 100 and 160 seconds), but periodically demonstrate large negative steps. These leaps are far from desirable, and suggestive of there being significant room for improvement in the devised strategy for PLL adjustment.

4.2 Perceptual Evaluation

The WFS system was subjected to an informal perceptual evaluation, in which participants were presented with a virtual sound source at various locations and asked to indicate, on a digram of the virtual sound field, the point at which they estimated the sound had emanated from. The informality of the experiment arose in

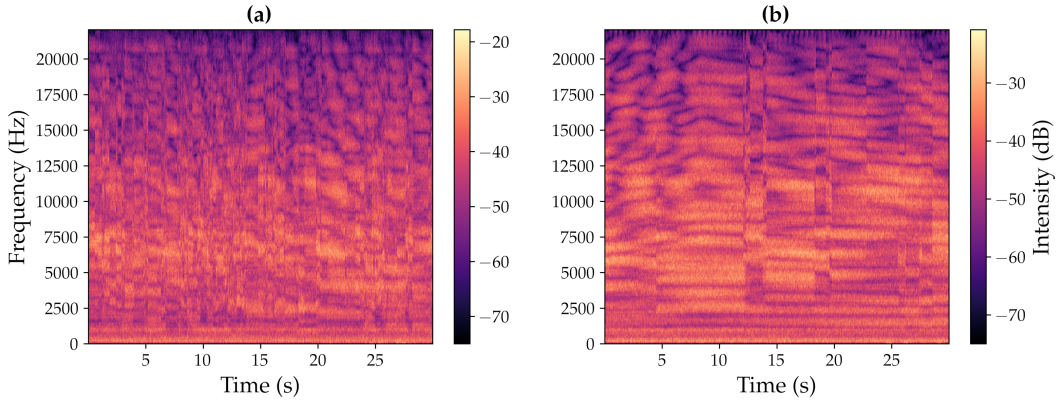


Figure 4.5: Magnitude spectrograms of ambient, monophonic recordings of a reproduction of white Gaussian noise by a group of eight networked audio clients driving an array of fifteen loudspeakers spaced at intervals of 0.175 m. Capacitor microphone placed ~2 m from the speaker array. Audio buffer (and thus network packet) size (a) 16 samples; (b) 32 samples.

part as a consequence of the listening environment not being acoustically treated, and there being sources of ambient sound in the laboratory in which the WFS system was installed. Furthermore, the speaker array (Figure 4.1) consisting of fifteen speakers, but each hardware module producing two audio output channels, the second channel of the right-most module was not used; for eight modules, however, the WFS plugin assumed a virtual sound field spanning sixteen speakers, thus it was possible to position a virtual sound source horizontally beyond the rightmost extent of the speaker array. Ultimately the aim of the experiment was to draw some preliminary, guiding conclusions as to the effectiveness of the distributed WFS system in triggering listeners' localisation cues, its technical and installation shortcomings notwithstanding.

In terms of the design of the auditory stimulus, it was felt that listeners would be most comfortable localising a naturalistic sound. Rather than use blasts white noise as in [32, ch. 6], but wishing to minimise the potential effects of frequency-dependent localisation interference due to spatial aliasing, a broadband stimulus was selected in the form of a close-mic recording of a snare drum. The recorded sample was repeated three times in succession at intervals of 0.125 s, and, again in the interests of adding a natural quality to the sound, with slight variations in amplitude (the second iteration of the sample was played marginally quieter than the first; the third slightly louder).

Participants were given a brief description of the system under evaluation, and informed that they should expect to hear sounds that appeared to emanate from 'behind' the speaker array, from which they stood at a distance of 2 m. Eight different virtual source positions were specified via automation of the x (lateral) and y (longitudinal distance) components of the position of a node in the WFS plugin interface. The range of the x component corresponded with the distance from the

driver of the leftmost speaker to the centre of the driver of the missing sixteenth loudspeaker; drivers lay at intervals of 0.175 m, giving a horizontal axis spanning 2.625 m. Longitudinal position was mapped to a range from 0 m (i.e. lying directly on the speaker array) to 10 m 'behind' the array.

For each position, the auditory stimulus was sounded, and repeated at the participant's request. Details of the source positions for each test, and the responses given by eight participants, are displayed in Figure 4.6.

As can be seen, although far from perfect, and with some significant outliers (e.g. the position reported by the fifth participant for test **(h)**), certain trends do appear to emerge from the results. Firstly, responses seem to loosely track the intended positions, with reported positions most closely corresponding with intended ones for virtual source locations lying close to the speaker array. Indeed, tests **(b)**, **(d)**, and **(g)** exhibit the lowest mean error values between the intended and reported positions. The results for tests **(c)** and **(h)**, exhibit the greatest mean error, and ambiguity regarding the lateral position of distant sound sources is to be expected; as the distance of a sound source from the listener increases, $r_k - y_k$ tends towards zero, and thus the ITD (and ILD) also approaches zero; thus, with increased distance the wavefront produced by a sound source (be it a real sound source or one synthesised under ideal conditions) approximates more and more closely a plane wave. In any case, despite this inherent, physical ambiguity, there is (visually at least) a tendency amongst the results toward the lateral location of the most longitudinally distant intended virtual source positions. Particularly for test **(c)**, participants seem to have had greater difficulty in estimating the depth of the virtual sound field; this may simply be as a function of their developing a familiarity with that aspect of it over the course of what was only a brief experiment.

Participants were asked for any anecdotal observations they had, based on their experience of the experiment. One participant noted, for the first position in particular, that the amplitude variations between the snare drum strikes gave the impression of a sound source that was advancing upon the listening position; for the lack of any visual cue as to the position of the sound source, this is a reasonable conclusion to draw; it did not, however, ultimately prevent them from reaching a decision with regard to their estimate for the position of the sound source. Another, likely hearing the time-varying comb-filter effect, asked whether the "phasing" they were hearing was intentional. A third, also perceiving a similar phenomenon, suggested that they felt that the sound sources were moving. Finally, a participant with prior experience working with WFS systems, remarked that the distance effect (i.e. the WFS prefilter) was perhaps a little extreme, and not altogether realistic.

4.3 Discussion

The temporal clustering and polarisation seen in Figures 4.3 and 4.4 is indicative of two points for concern with regard to technical implementation: the read-write difference threshold strategy may be insufficiently forgiving, forcing the read position into deleteriously fluctuating increment changes in response to periods of jitter. and without a master clock to indicate to each client the beginning of each output audio block, even with clock rates perfectly aligned, there is nothing to guarantee agreement of the timing of audio interrupts at the client side.

To avoid sudden, large or 'unrealistic' clock adjustments, clients assess the drift ratio as assessed via the ratio of network packet transmission to reception and, if it lies beyond an arbitrary threshold, simply resets the clock to the default 44.1 kHz. Resets of this sort may account for the large steps seen in the drift plots in Figures 4.3 and 4.4. It is clear that this strategy could be significantly improved upon.

The phasing effect noted by one participant is a disappointing consequence of the approach taken to combating jitter and keeping the clients close, temporally, together, and as close to the server as possible. It is clear that the current approach is, at best, too aggressive to be viable for high-quality audio output. Furthermore, an unpitched sound source such as a snare drum, though audibly susceptible to the time-varying comb-filter effect described, masks other artefacts caused by phenomena such as rapid fluctuations in the clients' buffer read position increment, and sudden, comparatively large audio clock adjustments.

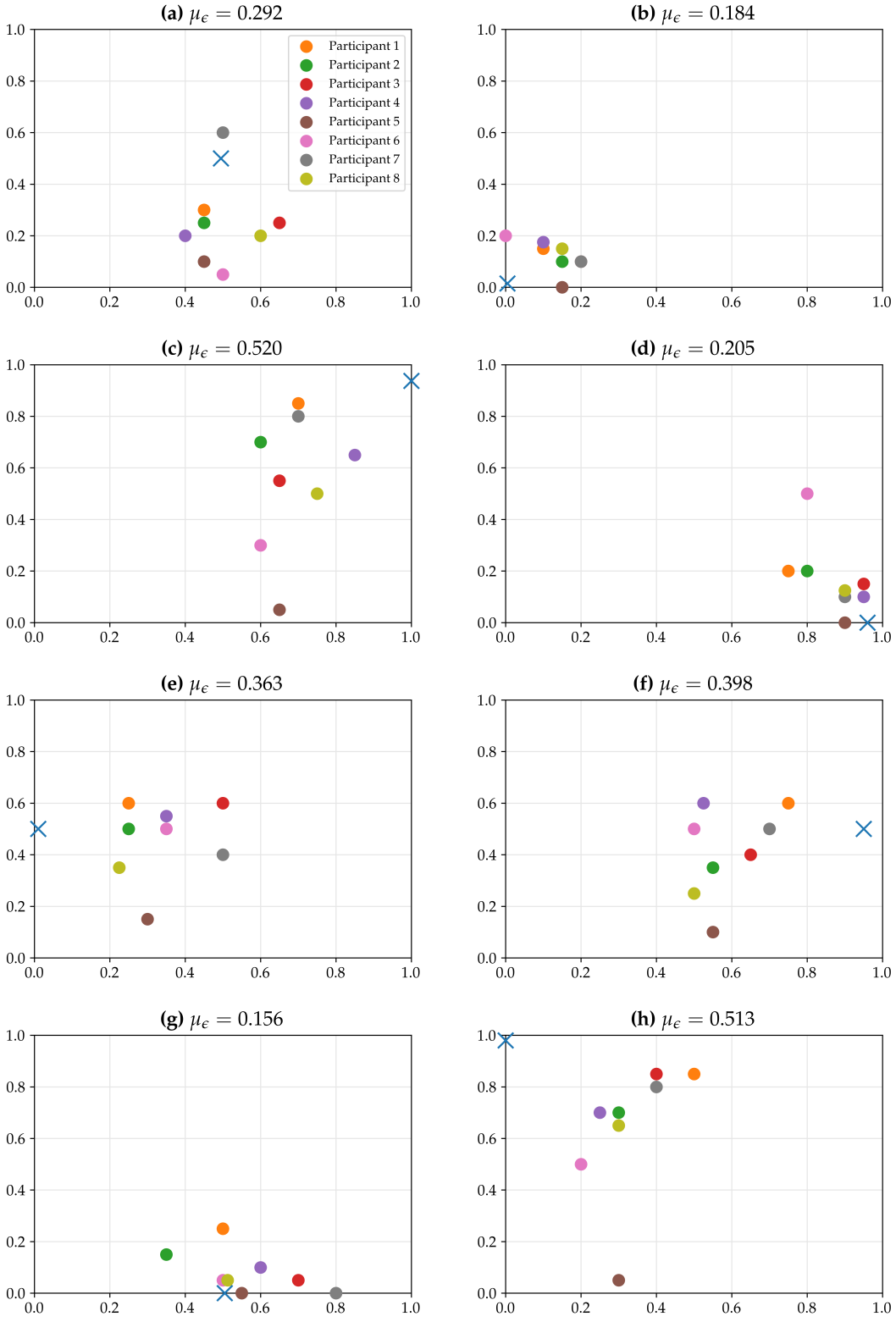


Figure 4.6: Results of the perceptual evaluation. Lateral (horizontal axis) and longitudinal (vertical axis) components are normalised to $[0, 1]$. For each plot, the horizontal axis (i.e. longitudinal component equalling 0) corresponds with the location of the speaker array. Each plot shows the intended position of the virtual sound source as specified by parameters to the WFS plugin interface (blue cross) and estimated sound source positions as reported by participants (coloured dots). Each plot is labelled with the mean euclidean error μ_ϵ between intended position and reported positions. Legend in plot **(a)** applies to all plots.

Chapter 5

Conclusion

This thesis has described an exploration of certain fundamental aspects of digital audio, transmission of audio over computer networks, distributed computing and audio spatialisation. A networked audio system was developed, strategies devised for addressing challenges presented by distributed computing, and a distributed spatial audio system was developed and deployed. Evaluation of the system exposed the extent of the technical challenges that confront it in its current form, and shed light on opportunities for further development. In an informal setting, perceptual testing revealed that it may, with heavy provisos, offer performance sufficient to support timing-critical sound field synthesis techniques.

It is pertinent at this point to consider the research questions stated in section 2.6, and what exploring those questions has revealed.

Research Question 1 It has been shown that a system of discrete computational entities can communicate effectively over an ethernet network, exchanging audio and control data in a scalable fashion via a UDP multicast group. Though facing significant technical challenges, an approach to the foundational requirements of such a system has been demonstrated, with encouraging results with regard to spatial audio applications.

Research Question 2 Timing discrepancies between entities in the networked audio system give rise to audible artefacts such as time-varying comb-filtering. Loss of synchronicity is difficult to measure and compensate for in real-time and the extent of the resulting audible disturbances is unpredictable, but, depending on the nature of the sound sources being dealt with, certainly perceptible. The developed strategies for mitigation of asynchronicity are *best-effort* in nature, and call for further refinement, or replacement with more sophisticated techniques.

5.1 Future Work

Given its potential to disrupt the present situation with regard to spatial audio installations, or complement it with a modular approach that could serve more flexible and perhaps creative ends, it is hoped that scope will exist to develop this work further. A number of technical challenges remain, and questions with regard to the fundamental, low-level characteristics of the various components of the devised system stand unanswered. From the level of audio hardware and driver software, to audio host and audio buffer behaviour, to network QoS and the performance of network switches, to the behaviour of the Teensy platform, and the processor and audio codec that it is built around, plus its software libraries — much that is typically taken for granted in the development of embedded systems, and audio and networking systems, calls for deeper investigation.

Clock-sharing Each Teensy generates a clock signal to deliver to its audio shield. In a locally distributed setting, this is quite redundant. It may be possible to generate a single, authoritative clock on one Teensy and deliver that to all others in the system. If this can be achieved, it would remedy the issue of clock drift, leaving only jitter to be addressed.

Further Audio Spatialisation Techniques A basic, linear, distributed wave field synthesis algorithm has been demonstrated, implementing primary virtual sources. WFS is capable of producing other types of sources, supporting nonlinear speaker arrays, three-dimensional arrays, and more faithful models of energy loss due to absorption. There are many further avenues to pursue, including optimising any DSP algorithms to best exploit the capabilities of what is, in the shape of the Teensy, a very powerful platform. Further, and only given cursory treatment here, higher order ambisonics remains as a worthy target for implementation in future work.

Bibliography

[1] A. J. Berkhout, D. de Vries, and P. Vogel. "Acoustic control by wave field synthesis". In: *The Journal of the Acoustical Society of America* 93.5 (May 1993), pp. 2764–2778. URL: https://asa.scitation.org/doi/abs/10.1121/1.405852 (visited on 12/15/2022).

[2] Jens Ahrens, Rudolph Rabenstein, and Sascha Spors. "The Theory of Wave Field Synthesis Revisited". In: Audio Engineering Society, May 2008. URL: https://www.aes.org/e-lib/browse.cfm?elib=14488 (visited on 12/15/2022).

[3] Jerome Daniel, Sebastien Moreau, and Rozenn Nicol. "Further Investigations of High-Order Ambisonics and Wavefield Synthesis for Holophonic Sound Imaging". In: *Audio Engineering Society Convention* 114 (2003).

[4] Matthias Frank, Franz Zotter, and Alois Sontacchi. "Producing 3D Audio in Ambisonics". In: *Audio Engineering Society Conference: 57th International Conference: The Future of Audio Entertainment Technology–Cinema, Television and the Internet* (2015).

[5] *IEEE Std 754™-2019 (Revision of IEEE Std 754-2008) IEEE Standard for Floating-Point Arithmetic*. Tech. rep. 2019.

[6] D. Cohen. "On Holy Wars and a Plea for Peace". In: *Computer* 14.10 (Oct. 1981), pp. 48–54.

[7] *Multimedia Programming Interface and Data Specifications 1.0*. Tech. rep. 1991. URL: https://www.mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/Docs/riffmci.pdf.

[8] *IEEE Standard for Ethernet (IEEE Std 802.3™-2018 (Revision of IEEE Std 802.3-2015)*. Tech. rep. IEEE, 2018. URL: https://ieeexplore.ieee.org/document/8457469/ (visited on 05/11/2023).

[9] Henning Schulzrinne. *Voice communication across the Internet: A network voice terminal*. Tech. rep. University of Massachusetts at Amherst, Department of Computer and Information Science, 1992.

[10] D. Cohen. *Specifications for the Network Voice Protocol (NVP)*. Tech. rep. RFC0741. RFC Editor, Nov. 1977, RFC0741. URL: https://www.rfc-editor.org/info/rfc0741 (visited on 05/15/2023).

[11] V. Hardman et al. "Reliable Audio for Use Over the Internet". In: *Proceedings of INET*. 1995. URL: https://web.archive.org/web/20160103083922/http://www.isoc.org/inet95/proceedings/PAPER/070/html/paper.html (visited on 12/15/2022).

[12] Vicky Hardman, Martina Angela Sasse, and Isidor Kouvelas. "Successful multiparty audio communication over the Internet". In: *Communications of the ACM* 41.5 (May 1998), pp. 74–80. URL: https://dl.acm.org/doi/10.1145/274946.274959 (visited on 12/19/2022).

[13] Chris Chafe et al. "A Simplified Approach to High Quality Music and Sound Over IP". In: *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)*. 2000.

[14] Aoxiang Xu et al. "Real-Time Streaming of Multichannel Audio Data over Internet". In: *Journal of the Audio Engineering Society* 48.7/8 (July 2000), pp. 627–641. URL: https://www.aes.org/e-lib/online/browse.cfm?elib=12056 (visited on 01/13/2023).

[15] Chris Chafe, Scott Wilson, and Daniel Walling. "Physical model synthesis with application to Internet acoustics". In: *2002 IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 4. May 2002, pp. IV–4056–IV–4059.

[16] Thierry Turletti. "The INRIA videoconferencing system (IVS)". In: *ConeXions* 8 (Jan. 1995).

[17] Flávio Luiz Schiavoni, Marcelo Queiroz, and Marcelo M Wanderley. "Alternatives in network transport protocols for audio streaming applications". In: *ICMC*. 2013.

[18] Fahad Taha AL-Dhief et al. "Performance comparison between TCP and UDP protocols in different simulation scenarios". In: *International Journal of Engineering & Technology* 7.4.36 (2018), pp. 172–176.

[19] David Meyer, Michelle Cotton, and Leo Vegoda. *IANA Guidelines for IPv4 Multicast Address Assignments*. Request for Comments RFC 5771. Internet Engineering Task Force, Mar. 2010. URL: https://datatracker.ietf.org/doc/rfc5771 (visited on 05/19/2023).

[20] Juan-Pablo Cáceres and Chris Chafe. "JackTrip: Under the Hood of an Engine for Network Audio". In: *Journal of New Music Research* 39.3 (Sept. 2010), pp. 183–187. URL: https://doi.org/10.1080/09298215.2010.481361 (visited on 12/15/2022).

[21] Juan-Pablo Cáceras and Chris Chafe. "JackTrip/SoundWIRE Meets Server Farm". In: *Computer Music Journal* 34.3 (2010), pp. 29–34. URL: https://www.jstor.org/stable/40963030 (visited on 12/15/2022).

[22] Alexander Carôt, Torben Hohn, and Christian Werner. *Netjack – Remote music collaboration with electronic sequencers on the Internet*. Jan. 2009.

[23] Volker Fischer. *Case Study: Performing Band Rehearsals on the Internet With Jamulus*. 2015. URL: https://jamulus.io/PerformingBandRehearsalsontheInternetWithJamulus.pdf.

[24] Alain Renaud, Alexander Carôt, and Pedro Rebelo. "Networked Music Performance : State of the Art". In: *AES 30th International Conference*. Jan. 2012.

[25] Luca Turchet and Carlo Fischione. "Elk Audio OS: An Open Source Operating System for the Internet of Musical Things". In: *ACM Transactions on Internet of Things* 2.2 (Mar. 2021), 12:1–12:18. URL: https://doi.org/10.1145/3446393 (visited on 01/19/2023).

[26] *What is Dante? | Audinate | Dante Pro AV Networking*. URL: https://www.audinate.com/meet-dante/what-is-dante (visited on 12/20/2022).

[27] Ron Bakker, Andy Cooper, and Atsushi Kitagawa. *An introduction to networked audio*. White Paper. Yamaha Commercial Audio Team, 2014.

[28] Jens Ahrens. *Analytic Methods of Sound Field Synthesis*. T-Labs Series in Telecommunication Services. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. URL: https://link.springer.com/10.1007/978-3-642-25743-8 (visited on 05/13/2023).

[29] Rozenn Nicol. "Sound Field". In: *Immersive Sound*. Routledge, 2017, pp. 276–310.

[30] Tim Ziemer. "Wave Field Synthesis". In: *Psychoacoustic Music Sound Field Synthesis*. Vol. 7. Cham: Springer International Publishing, 2020, pp. 203–243. URL: http://link.springer.com/10.1007/978-3-030-23033-3_8 (visited on 05/17/2023).

[31] Ville Pulkki. "Virtual Sound Source Positioning Using Vector Base Amplitude Panning". In: *J. Audio Eng. Soc* 45.6 (1997), pp. 456–466. URL: http://www.aes.org/e-lib/browse.cfm?elib=7853.

[32] E. N. G. Verheijen. "Sound reproduction by wave field synthesis". In: (1998). URL: https://repository.tudelft.nl/islandora/object/uuid%3A9a35b281-f19d-4f08-bec7-64f6920a3821 (visited on 05/18/2023).

[33] Giovnni De Poli and Davide Rocchesso. "Physically based sound modelling". In: *Organised Sound* 3.1 (Apr. 1998), pp. 61–76. URL: http://www.journals.cambridge.org/abstract_S1355771898009182 (visited on 02/10/2022).

[34] Matthias Geier, Jens Ahrens, and Sascha Spors. "Object-based Audio Reproduction and the Audio Scene Description Format". In: *Organised Sound* 15.03 (Dec. 2010), pp. 219–227. URL: http://www.journals.cambridge.org/abstract_S1355771810000324 (visited on 01/17/2023).

[35] Rolf K Mueller. "Acoustic holography". In: *Proceedings of the IEEE* 59.9 (1971), pp. 1319–1335.

[36] Jose A. Belloch et al. "On the performance of a GPU-based SoC in a distributed spatial audio system". In: *The Journal of Supercomputing* 77.7 (July 2021), pp. 6920–6935. URL: https://doi.org/10.1007/s11227-020-03577-4 (visited on 12/20/2022).

[37] A. J. Berkhout. "A Holographic Approach to Acoustic Control". In: *Journal of the Audio Engineering Society* 36.12 (Dec. 1988), pp. 977–995. URL: https://www.aes.org/e-lib/browse.cfm?elib=5117 (visited on 05/18/2023).

[38] F Winter, J Ahrens, and S Spors. "A Geometric Model for Spatial Aliasing in Wave Field Synthesis". In: *Proc. of German Annual Conference on Acoustics (DAGA)* (2018).

[39] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, Mar. 2011.

[40] Leslie Lamport and Nancy Lynch. "Distributed Computing: Models and Methods". In: *Formal Models and Semantics*. Elsevier, 1990, pp. 1157–1199. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780444880741500238 (visited on 05/14/2023).

[41] Chris Chafe. "I am Streaming in a Room". In: *Frontiers in Digital Humanities* 5 (2018). URL: https://www.frontiersin.org/articles/10.3389/fdigh.2018.00027 (visited on 12/16/2022).

[42] Marina Bosi et al. "Experiencing Remote Classical Music Performance Over Long Distance: A JackTrip Concert Between Two Continents During the Pandemic". In: *Journal of the Audio Engineering Society* 69.12 (Dec. 2021), pp. 934–945. URL: https://www.aes.org/e-lib/browse.cfm?elib=21542 (visited on 12/16/2022).

[43] Matteo Sacchetto, Antonio Servetti, and Chris Chafe. "JackTrip-WebRTC: Networked music experiments with PCM stereo audio in a Web browser". In: *Web Audio Conference WAC-2021*. 2021.

[44] Nelson Posse Lago. *Distributed Real-Time Audio Processing*. 2004.

[45] Leonardo Gabrielli et al. "Networked Beagleboards for wireless music applications". In: *2012 5th European DSP Education and Research Conference (EDERC)*. Sept. 2012, pp. 291–295.

[46] Fernando Lopez-Lezcano. “From Jack to UDP packets to sound and back”. In: *Proceedings of the Linux Audio Conference*. Vol. 2012. 2012.

[47] Sean Devonport and Richard Foss. “The Distribution of Ambisonic and Point Source Rendering to Ethernet AVB Speakers”. In: *5th International Conference on Spatial Audio*. 2019.

[48] Manu Mitterhuber, Rojin Sharafi, and Enrique Tomás. *Ottosonics*. URL: https://tamlab.kunstuni-linz.at/projects/ottosonics/ (visited on 05/04/2023).

[49] Hicham Marouani and Michel R. Dagenais. “Internal Clock Drift Estimation in Computer Clusters”. In: *Journal of Computer Networks and Communications* 2008 (May 2008), e583162. URL: https://www.hindawi.com/journals/jcnc/2008/583162/ (visited on 12/15/2022).

[50] Fons Adriaensen and Alcatel Space. “Using a DLL to filter time”. In: *Linux audio conference*. 2005.

[51] Fons Adriaensen. “Controlling adaptive resampling”. In: *Linux Audio Conference*. 2012.

[52] *socket(2) - Linux manual page*. URL: https://man7.org/linux/man-pages/man2/socket.2.html (visited on 05/24/2023).

[53] Thomas Albert Rushton, Romain Michon, and Stéphane Letz. “A Microcontroller-based Network Client Towards Distributed Spatial Audio”. In: *Sound and Music Computing Conference (SMC-23)*. 2023.

[54] Romain Michon et al. “Real Time Audio Digital Signal Processing With Faust and the Teensy”. In: *Sound and Music Computing Conference (SMC-19)*. Malaga, Spain, May 2019. URL: https://hal.archives-ouvertes.fr/hal-03153709 (visited on 12/15/2022).

REMOVE THE TODO LIST

Appendix A

Prior Work

At the time of writing, the paper *A Microcontroller-based Network Client Towards Distributed Spatial Audio*[53], which is to feature in the proceedings of the 2023 *Sound and Music Computing* conference, has not yet been published. For the reader's convenience, it is included here in the pages to follow.

A MICROCONTROLLER-BASED NETWORK CLIENT TOWARDS DISTRIBUTED SPATIAL AUDIO

Thomas Albert Rushton
Aalborg University
A. C. Meyers Vænge 15
2450 Copenhagen, Denmark
trusht21@student.aau.dk

Romain Michon
Univ Lyon, Inria,
INSA Lyon, CITI, EA3720
69621 Villeurbanne, France
michon@grame.fr

Stéphane Letz
Univ Lyon, GRAME-CNCM,
INSA Lyon, Inria, CITI, EA3720
69621 Villeurbanne, France
letz@grame.fr

ABSTRACT

Audio spatialisation techniques such as wave field synthesis call for the deployment of large arrays of loudspeakers, typically managed by dedicated audio hardware. Such systems are typically costly, inflexible, and limited by the computational demands and high throughput requirements of centralised, highly-multichannel digital signal processing. The development of a distributed system for audio spatialisation based on *Audio over Ethernet* represents a potential easing of the infrastructural burdens posed by traditional, centralised approaches.

This work details the development of a networked audio client, supporting the popular JackTrip audio protocol, and running on a low-cost microcontroller. The system is applied to the case of a wave field synthesis installation, with a number of client instances forming a distributed array of signal processors. The problems of client-server latency, and interclient synchronicity are discussed and a mitigative strategy described. The client software and hardware modules could support large scale audio installations, plus serve as self-contained interfaces for other networked audio applications.

1. INTRODUCTION

The past few decades have seen a rapid increase in interest in audio spatialisation techniques. In the commercial sphere, surround-sound systems and networked home entertainment platforms have become commonplace. In academia, techniques such as Wave Field Synthesis (WFS) and Ambisonics [1–3] have received considerable treatment, and their applications for *object based* approaches to audio personalisation [4, 5] continue to draw research interest.

Audio spatialisation systems tend, however, to be monolithic *in-situ* installations, requiring dedicated, multichannel hardware, and bringing with them the costs and inflexibility associated with specialist equipment. The recent emergence of a raft of low-cost, small form-factor consumer microcontrollers with dedicated audio functionality, presents an opportunity to explore other approaches to audio spatialisation, incorporating flexibility of application, distributed computing, and taking advantage of the benefits of networked audio.

Taking one such microcontroller platform and using WFS as a proof-of-concept implementation, this project seeks to establish how a distributed, networked array of microcontrollers can be used to support a low-cost, modular implementation of an audio spatialisation system. To stand as a viable alternative or complement to established approaches, such a system demands reliable transmission of audio over a network protocol, low latency, and high synchronicity. Interoperability with other audio systems is also a priority. A credible implementation could serve as the basis for large-scale systems, other spatialisation techniques, and a variety of practical and creative applications of networked audio.

2. BACKGROUND

Networked audio has been a ubiquitous part of digital communication for many years. As the availability of high-speed internet connections has improved, so have opportunities for real-time network audio transmission for activities such as videoconferencing and musical performance. Since its earliest days, networked audio research has focused on the challenges posed by transmission across networks where data integrity is not guaranteed [6]. Packet loss and *jitter* — inconsistency in the rate of packet arrival — are perennial issues, and incorporating redundancy and a forgiving buffering strategy, while minimising latency, have always entailed striking a fine balance [6, 7].

While early work on Audio over Ethernet (AoE) centred on communication [6, 8], by the late 1990s research was underway with respect to how audio in general, and music in particular, could be created and transmitted over computer networks in real time [9, 10]. In 1999, UDP-based concertcasts of compressed audio were being facilitated by the *SoundWIRE* system [9], featuring buffering strategies designed to overcome the jarring effects of packet loss on networked musical performance, albeit at the expense of latency amounting to *"a number of seconds"* [9]. Suffice it to say, no definitive solution has been found to the problems of latency and jitter, and they remain active topics of research [11, 12].

Networked audio presents a unique challenge amongst modes of real-time network communication due to the short timescales and fine margins at play [9]. Though packet loss can be monitored relatively easily, jitter, no less deleteri-

ous to the quality of an audio signal under transmission, may occur suddenly and sporadically, and can be difficult to monitor and quantify. Perhaps as a by-product of these difficulties, pioneering work on AoE featured creative applications of networked audio; Chafe et al. [13, 14] essentially treated the internet as a resonant medium, using the network as a digital waveguide with their SoundWIRE-based *Network Harp*. This approach served as a way of *sonifying* jitter, thus providing an intuitive, auditory quality of service (QoS) measure.

2.1 Protocols for Networked Audio

In principle, the transmission of audio signals across a computer network is subject only to identifying an appropriate network transport layer protocol, and establishing a suitable scheme for encoding and decoding audio data at the points of delivery and reception. If signal integrity is a higher priority than minimising latency, or if high QoS cannot otherwise be ensured (e.g. over a Wide Area Network), Transmission Control Protocol (TCP), with its guarantees on packet ordering and retransmission of lost packets, may be a good choice. If low-latency is of greater importance, or if working with a Local Area Network (LAN), User Datagram Protocol (UDP), which provides no such guarantees, but exhibits none of the computational overhead associated with ensuring the integrity of the packet stream, may be preferable [15]. Further, TCP is a connection-oriented protocol and thus supports strictly one-to-one communication; UDP, on the other hand, is *connectionless*, and supports one-to-many, and many-to-many *multicast* communication.

Networked audio brings with it certain domain-specific requirements, however; it is sensible to include metadata in audio packets, such as the sampling rate and bit resolution. A variety of more-or-less opinionated protocols and systems — typically UDP-based but with audio-specific features — have been developed to provide specific support for AoE, of which a selection are described in the paragraphs to follow.

Before proceeding, it is worthwhile to introduce the JACK Audio Connection Kit, (JACK [1]) a cross-platform sound server that acts as a layer between audio applications and the underlying audio host. [2] JACK provides, amongst other things, an API for making arbitrary connections between the input and output ports of audio applications and devices.

Developed as part of the JACK suite, Netjack [3] is a multicast system for distributed musical performances [16]. Netjack operates on a centralised model, with one machine, running JACK on a traditional audio host, acting as the server, and clients running on JACK's *NET* host.

Other JACK-based systems include Zita-njbridge, [4] a command line, multicast audio client intended for use over local networks, and JackTrip [17, 18], a successor to the SoundWIRE project, described as *"a [...] system that supports multi-machine network [audio] performance over best-effort Internet"* [17]. Zita-njbridge uses adaptive resampling at the receiver to compensate for the lack of a common clock between network nodes [19]. JackTrip provides a selection of buffering strategies to protect against jitter [17], but supports only unicast transmission.

There is the proprietary Dante [20], which uses Precision Time Protocol (PTP) and specialist hardware to achieve synchronicity between clients, the Audio Video Bridging (AVB, IEEE 802.1) protocol for highly-synchronised audio (and video) over ethernet, plus a plethora of rehearsal and jamming-focused platforms [21–23], in which JackTrip also features [12].

Amidst this multitude of systems it may be tempting, then, to pursue a *'No-protocol'*, or 'raw-UDP' solution such as in [24]. Such an approach is certainly an option, but systems of this sort are by nature *sui generis*, and unlikely to find use beyond their specific application. The prospect of interoperability via building atop an accessible, widely-used platform, potentially opens the door to creative applications and interactions with existing tools.

2.2 Distributed, Networked Audio Systems

Clearly the idea of taking a distributed approach to real-time audio and music creation is not without its precedents. Of particular interest here are projects concerned with dividing the computation of some signal process or audio synthesis amongst multiple network nodes. These include the aforementioned Network Harp [13] project and related work on *internet acoustics*, plus ongoing use of JackTrip for purposes such as distributed reverberation, again taking advantage of a computer network's ancillary nature as a system of delay lines [25].

But why pursue a distributed approach to audio spatialisation? As described in section 1, centralised systems tend to be single-purpose installations reliant on costly, specialist equipment, such as highly multichannel audio interfaces. Such systems cannot easily be extended, and dedicated, niche control software runs a high risk of obsolescence. A well-designed distributed system could be modular and scalable, reprogrammable to support a variety of applications, generic and trivial to upgrade.

Distributed JackTrip networks have been implemented in such a fashion, with clients running on Raspberry Pi single-board computers [26]. A networked 'music studio' of wirelessly networked Beagleboard single-board computers has been demonstrated [27]. Indeed, Belloch et al. created a distributed WFS implementation [28], albeit running a non-generic system on a GPU-based single-board hardware platform.

Centralised approaches may, however, offer advantages in terms of reliability — distributed systems may have more potential points of failure — and synchronicity — distributed audio systems are subject to timing discrepancies, as each node in such a system has its own clock. The latter point is particularly pertinent in the case of timing-critical applications such as spatial audio; Belloch et al. relied on an external Network Time Protocol (NTP) clock to synchronise the nodes in their WFS implementation.

[1] `https://jackaudio.org/` (This and all other URLs last accessed 23/01/2023.)

[2] For Linux, the host is typically ALSA (Advanced Linux Sound Architecture); on Mac, Core Audio; ASIO (Audio Stream I/O) on Windows.

[3] `https://github.com/jackaudio/jackaudio.github.com/wiki/WalkThrough_User_NetJack2`

[4] `https://kokkinizita.linuxaudio.org/linuxaudio/`

2.3 Wave Field Synthesis

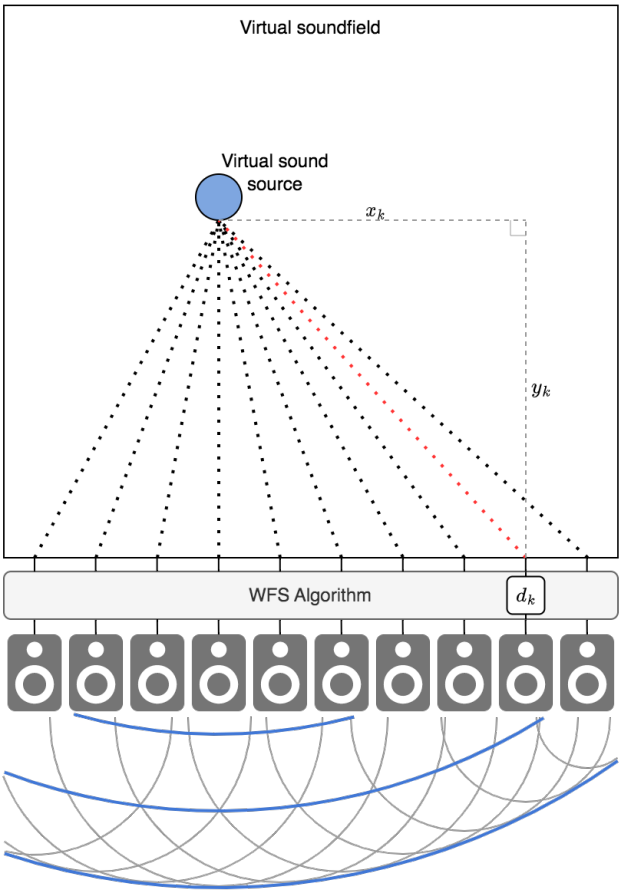


Figure 1: The basic principle of WFS. A sound source is situated within a virtual soundfield; dotted lines indicate simulated distances from an array of secondary point sources. x, y position and delay d indicated with respect to the kth secondary point source. With appropriate timing, a wavefront can be recreated suggesting the position of the sound source, were it physically present.

Wave Field Synthesis is a form of acoustic holography [1], or *holophony* [3], based on Huygens' principle, which states that the propagation of a wavefront can be recreated from a collection of secondary point sources, such as an array of loudspeakers [1, 28, 29]. Whereas more commonplace spatialisation techniques, such as stereo panning and surround-sound, use amplitude-based rules to give the impression of a sound source emerging from a given location, WFS, in its most basic form, composes wavefronts from secondary point sources via fine control of delay lines (see Figure 1).

Given the simulated position of a sound source relative to a speaker array, the equivalent time for sound to reach each secondary point source can be computed, and used to set an appropriate sample delay d for the kth secondary point source:

$$d_k = \frac{F_s}{c}\sqrt{x_k^2 + y_k^2}, \tag{1}$$

where c is the speed of sound and F_s the system sampling rate.

The above describes a basic, two-dimensional approach to WFS, consisting of primary (virtual) point sources and a linear distribution of secondary point sources (loudspeakers). WFS supports the generation of *planar sources* — analogous to primary point sources at infinite distance — and *focused sources*, virtual sources located in the non-virtual sound field [29]. Extensions to three-dimensional and circular or irregular arrays of loudspeakers are also possible [2]. Further, the discrete nature of the positioning of secondary sources gives rise to the phenomenon of spatial aliasing [30]. Suffice it to say, WFS is a topic of significant depth and nuance, the greater part of which will not be treated here.

Of greatest pertinence to this work is the fact that the *WFS Algorithm* pictured in Figure 1 does not need to be one single piece of signal processing. Each speaker in the array could be driven by its own dedicated algorithm, so long as the underlying implementation is aware of its position in the array.

3. DEVELOPMENT

The Teensy 4.1 [5] microcontroller was selected as the hardware platform for the networked audio client. Teensy 4.1 is powerful (600 MHz CPU, 1024 kB RAM), inexpensive ($31.50 at the time of writing) device, with a dedicated audio shield and accompanying audio development library, and built-in Ethernet functionality (albeit available via a further hardware add-on); it is also, as the name suggests, rather small (61 mm in length). Development for Teensy is conducted in the C++ programming language. Teensy is also one of a number of embedded platforms supported by the Faust audio programming language [6] via the `faust2teensy` utility, which compiles Faust code to C++ compatible with the Teensy Audio Library [31]. [7]

Unlike the single-board platforms used in [26–28] Teensy is a true microcontroller system and does not run an operating system. [8] This means development on Teensy can be conducted very rapidly (in part due to a very short start-up time), but precludes the installation of complex software such as JACK, or even JackTrip as was the case for [26]. Teensy has no native threading model; operations take place on hardware interrupts, with a default sampling rate of 44.1 kHz and buffer size of 128 samples, though the latter can be altered via a compiler flag.

3.1 Hardware Setup

Eight Teensy 4.1s were assembled, with audio shield and Ethernet add-on (see Figure 2). Teensy receives 5 V power over USB, and each module was connected to a USB hub. The audio shield provides two-channel output and these outputs were connected to a pair of powered monitor speakers. A Cat 6 Ethernet cable was used to connect each Teensy's Ethernet add-on to a Gigabit Ethernet switch. The switch was in turn connected to a laptop, running a Linux operating system (Ubuntu 20.04.5).

[5] `https://www.pjrc.com/teensy/`
[6] `https://faust.grame.fr/`
[7] `https://www.pjrc.com/teensy/td_libs_Audio.html`
[8] An OS, e.g. RTOS or Zephyr, *can* be run on Teensy, but, in the interests of maintaining flexibility, and compatibility with the Teensy Audio Library, this possibility was dismissed early in development.

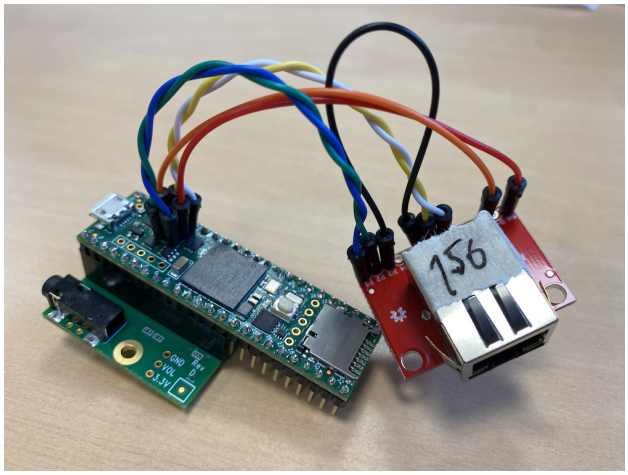


Figure 2: A hardware module, featuring Teensy 4.1 microcontroller, audio shield, and Ethernet add-on, labelled with the last octet of its IP address.

3.2 Audio Protocol

The JackTrip and Netjack protocols were considered for implementation. Netjack's multicast functionality makes it an attractive candidate, but due to its comparatively complex architecture, it was not possible within the scope of the project to establish a functioning Netjack client on the Teensy platform.

JackTrip, by comparison, proved relatively simple to interact with. With JACK running, JackTrip can be started with the following terminal command:

```
jacktrip -S -q2 -p5 -n<N>
```

with `-S` indicating that JackTrip should run in hub server mode, i.e. as a server to which multiple clients may join; `-q2` instructs JackTrip to use its lowest packet buffer size (two) in the interests of minimising network transmission round trip time (RTT); `-p5` specifies that no autopatching via the JACK API should occur when a client connects to the server; `-n<N>` specifies the use of `N` input and output audio channels.

A connection to a JackTrip server is initiated via a TCP handshake, whereby server and client exchange UDP port numbers. Following a successful handshake, the server creates a send and a receive thread and transmission begins over UDP as described in [17]; this is subject only to the condition that peers operate at the same sampling rate and buffer size.

JackTrip uses the IP address of the client and, via a series of calls to the JACK API, sets up a device in the JACK graph representing that client, complete with the appropriate number of input and output ports.

The number of audio channels is limited by a combination of the buffer size and maximum packet size available under the network configuration. Though the maximum theoretical UDP datagram size is 65535 bytes, this is limited by the Maximum Transmission Unit of the network link layer and the Ethernet payload size (1500 bytes) [15]. Additionally, Teensy sets a default maximum socket size of 1024 bytes. JackTrip sends a header with each UDP packet (containing information such as the sampling rate and bit resolution) totalling 16 bytes; this leaves 1008 bytes for audio data. The maximum number of channels that may be transmitted, then, is

$$C_{\max} = \left\lfloor \frac{1008}{N_B N_s} \right\rfloor \tag{2}$$

where N_B is the number of bytes per audio sample and N_s is the number of samples per buffer. A smaller audio buffer size permits a greater number of channels, and vice versa.

With a couple of provisos, it was straightforward to program the modules to achieve the client-side requirements described in the preceding section. Each device that joins an Ethernet network must have a unique MAC (Media Access Control) address; Teensy has no such address, but one can be derived from its unique serial number. Additionally, since no network router is present, and thus no DHCP (Dynamic Host Configuration Protocol) server to automatically assign IP addresses, each Teensy was assigned an IP address derived also from its serial number. Modules were programmed to poll the network immediately after booting for a JackTrip server. The TCP handshake typically takes place within a few seconds, following which the exchange of UDP packets begins. The JackTrip client implementation was composed as a class compatible with the Teensy Audio Library named `JackTripClient`.[9]

3.3 Challenges

Ideal transmission would be latency-free and perfectly synchronous; temporal inconsistencies between the server and clients emerged as the principal source of difficulty with regard to supporting a low-latency, high-synchronicity system.

3.3.1 Clock Drift

The timing of a computer system is typically governed by a crystal oscillator. Naturally no two crystals are the same, so no two computers run at the same speed. Clock speed and stability are affected, additionally, by factors such as ambient temperature and computational load [32].

Due to Teensy's threadless architecture, it was deemed most sensible to check for an incoming UDP packet once per audio hardware interrupt. In a perfectly synchronous world this would be fine; in reality, however, the rate of packet arrival does not perfectly correspond with the occurrence of Teensy's audio interrupt, so the number of available packets may be zero, or greater than one. The solution to this issue was to establish a circular buffer at the client side, and read *greedily* from the network, consuming all available packets at each interrupt. This approach guarantees an increase in latency, though for a low audio buffer size this increase will be modest. Indeed, for a technique such as WFS, synchronisation is of course of far more pressing concern.

This strategy was not sufficient to handle situations where the Teensy might fall far behind (or run far ahead of) the server, in which case the circular buffer read and write

[9] Code and usage notes can be found in a repository at `https://github.com/hatchjaw/jacktrip-teensy`

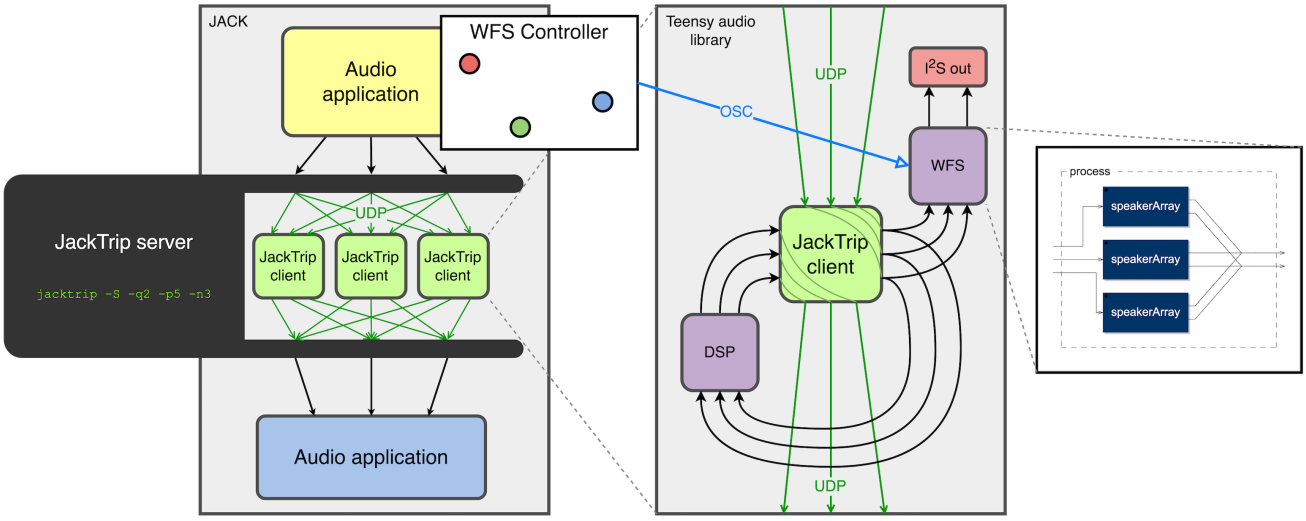


Figure 3: Overview of a networked audio system incorporating the microcontroller-based JackTrip client and distributed WFS system described in this work. Connections between components are illustrative of one potential configuration.

positions would eventually overlap. To mitigate this, a form of adaptive resampling was implemented: the ratio of circular buffer writes to reads was taken every 1000 writes, and from that a non-integer read-position increment derived, designed to keep the now fractional read position at a manageable interval behind the integer write index. Cubic interpolation was used with respect to the read position to compute an appropriate sample value to use for output. This differs from JackTrip's own strategies for handling circular buffer underrun and overflow conditions [17, §2.2], and aims to ensure audio output free from the audible artefacts of large read-position adjustments or circular buffer resets.

3.3.2 Jitter

The scenario described in section 3.3.1, whereby an attempted packet-read may yield a number of packets less than or greater than one, represents a form of *jitter*. Jitter manifestations of that form, related to clock drift, are relatively easy to guard against, but short-term network instability and changes in process prioritisation by a computer running many other tasks besides a JackTrip server, were found to result in chaotic changes in the delta between the read position and write index to the client's circular buffer, potentially erratic enough to overcome the cushion provided by the circular buffer and fractional read-position increment.

To mitigate, low (L) and high (H) thresholds were introduced for the read-write delta (Δ). Between the thresholds, the read-position increment is derived from the long-term read-write ratio. If the delta exceeds the high threshold the read-position increment is increased to Δ/H to return the read position to between H and L; when the delta falls below the low threshold the read-position increment is reduced to Δ/L.

Protecting against jitter entails increasing latency, and the method used here requires striking a compromise between latency, inter-client synchronicity, and the perceptual impact of rapid fluctuations in the read-position increment. Lowering the low read-write delta threshold, with the aim of minimising server-client latency, increases the risk of the read position approaching the write index and coming to a halt as the read-position increment approaches zero. Set too narrow an interval between low and high thresholds, with the aim of optimising inter-client synchronicity, and it is inevitable, during periods of high jitter, that the read position increment will fluctuate wildly enough to introduce audible artefacts to the output signal.

3.4 The WFS Algorithm

A distributed WFS program for Teensy was created by combining `JackTripClient` with a modular WFS algorithm developed in Faust. Parameters to this algorithm are c, the simulated speed of sound (m/s) in the virtual sound field, and s_H, the inter-speaker spacing (m).

For the ith virtual sound source, with position (x_i, y_i) in the virtual sound field, the Faust algorithm must calculate the appropriate delay line length $d_{i,k}$ for each secondary point source k for which it is responsible. For all secondary point sources the longitudinal component $y_{i,k} = y_i$, but $x_{i,k}$ varies with respect to k. The algorithm receives an index m representing the position in the speaker array of the module on which it is running; with this value the lateral component may be found:

$$x_{i,k} = x_i - s_H(k + 2m). \tag{3}$$

In practice, the longitudinal distance between the virtual sources and the speaker array is not temporally relevant, so this can be subtracted from equation (1) to give the the relative delay for each secondary point source:

$$d_{i,k} = \frac{F_s}{c}\left(\sqrt{x_{i,k}^2 + y_{i,k}^2} - y_{i,k}\right), \tag{4}$$

which has the advantage of limiting the maximum delay line length to the time taken for sound to traverse the speaker array, and thus minimises the memory footprint of the algorithm. Equation (4) produces a non-integer value for $d_{i,k}$

so the algorithm employs linearly-interpolated fractional delay lines via Faust's `de.fdelay` function. [10]

This effectively models the lateral position of a given sound source. As sound propagates, some of its energy is dissipated to the medium of propagation, with energy in high frequencies lost more rapidly than low frequencies. So, to give a basic impression of longitudinal distance, an inverse mapping was created between $y_{i,k}$ and the amplitude of the ith sound source, and the cutoff frequency of a second order Butterworth lowpass filter applied to that sound source.

To support the WFS system, a desktop application was developed using the JUCE C++ audio application framework. [11] The application consists of a multichannel audio source, with each channel representing a virtual sound source, and user interface elements that trigger control messages as Open Sound Control (OSC) data distributed to the modules over multicast UDP. Additionally, the application uses the JACK C API to connect its own audio output ports to the inputs of all `JackTripClient` instances found in the JACK graph.

The user interface features a series of dropdown menus corresponding with speaker-pair positions, containing the IP addresses of the connected modules. Assigning a module to a speaker pair sends the index m to the appropriate module. Also featured is an XY-coordinate controller to which movable nodes may be added. Upon adding a node, the user is instructed to select an audio file to associate with that node, and the file is registered to the multichannel audio source. The position, (x_i, y_i), of the node is reported to all clients and updated when the node is moved.

4. RESULTS

The eight-module WFS system was demonstrated, with clients using an audio buffer size of 32 samples and low and high read-write delta thresholds of 32 and 64 samples respectively. As per equation (2), the buffer size permitted the transmission of a maximum of 15 channels. Anecdotally, the system supported the holophonic effect. To assess the quality of inter-module synchronisation, a separate test configuration of the modules was created and recordings taken. [12] A test signal was transmitted to the clients, taking the form of a unipolar 16 bit sawtooth wave with unit amplitude increment per sample, thus having a period of 2^{15} samples. In addition, a unipolar sawtooth wave was generated on the clients themselves. Signal routing on the clients was configured such that each client would return two channels to the server: 1) the incoming sawtooth wave, following adaptive resampling; 2) the difference in amplitude between the incoming sawtooth wave and the sawtooth wave generated on the client.

The first returning channel was subtracted from the outgoing sawtooth wave to provide a measure of the full network round trip transmission time for each client (see [27] for a similar approach). The middle plot in Figure 4a shows that, over the course of a 10 min session, clients remained synchronised, on average, to around 26 samples (~590 µs), albeit exhibiting temporal drift relative to each other, as shown in the upper plot in Figure 4a and detail in Figure 4b.

Regarding clock drift, as the lower plot in Figure 4a shows, the trend is for clients to fall behind the server in approximately linear fashion (under stable ambient and computational conditions) but at differing rates. The oscillatory modulation of the drift patterns arises from an interaction between the clock drift and the 'greedy' packet-read strategy (described in section 3.3.1); as hardware interrupts on the client fall in and out of phase with the arrival of UDP packets, the likelihood changes of reading zero or multiple packets. It is intuitive, then, that clients exhibiting greater drift also display higher-frequency oscillatory modulation.

These results suggest that the system as demonstrated may have performed well enough to create a convincing spatial effect at least some of the time, but that there is scope for improvement. Speakers were spaced at intervals of 0.23 m, a distance travelled by sound in air in ~671 µs or ~30 samples at 44.1 kHz. There would almost certainly have been some spatial ambiguity at the boundary of adjacent speaker pairs.

Computationally, however, the system does not place undue strain on the Teensy platform. Memory usage in testing was consistently low (512 B to 1536 B), and, handling the full 15 available virtual sources, the algorithm consumes ~25% of available CPU time. `JackTripClient` alone uses only ~4% CPU.

5. CONCLUSION

This project demonstrated a proof-of-concept distributed audio spatialisation system comprised of a collection of networked audio modules based on an affordable microcontroller platform. The system proposed here has been shown to effectively distribute the computation of a WFS system across a collection of networked modules; the underlying networked audio client provides respectable module synchronicity, and may support timing-critical applications, albeit with scope for improvement in this regard.

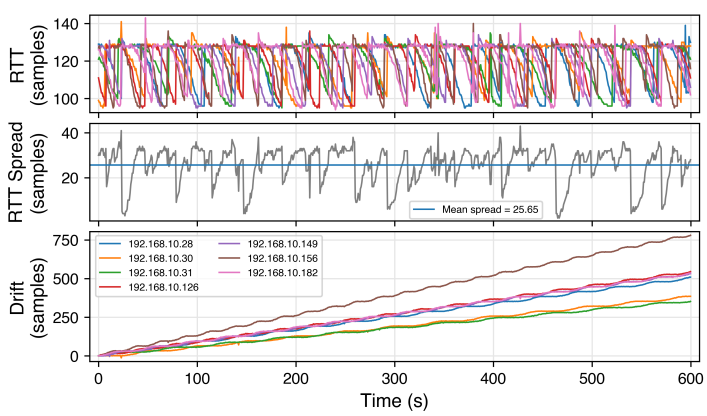
The implementation of a JackTrip client in the Teensy Audio environment is an ideal starting-point for further networked audio projects, and in this sense interoperability with existing networked audio systems has been shown to be well within the reach of the Teensy platform. That being said, for high-synchronicity applications JackTrip proved perhaps not to be the ideal choice of protocol, owing to its multithreaded model and lack of multicast functionality. Although more complex to establish as a bare-bones embedded client, the creation of a Teensy-based Netjack client promises to open the way for multicast transmission, and, it is hoped, greater guarantees of synchronicity. An AVB client would also be a very useful addition to the Teensy ecosystem.

The WFS model implemented is fairly rudimentary and calls for extension to other types of virtual sound sources and speaker array configurations. Ambisonics would be an ideal follow-up implementation to attempt, but an in-

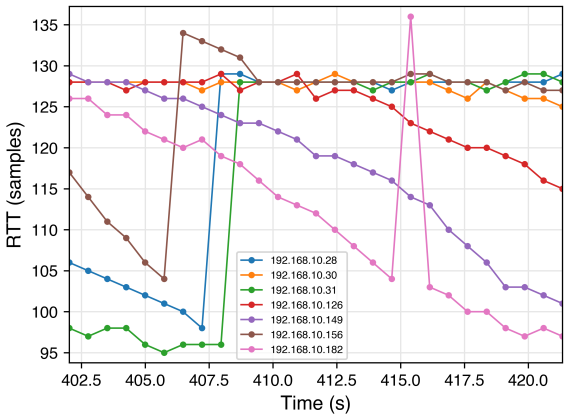
[10] `https://faustlibraries.grame.fr/libs/delays/#defdelay`

[11] JUCE 7.0.2 `https://github.com/juce-framework/JUCE`

[12] Only one eight-port ethernet switch was available for this test, one port of which was reserved for connection to the JackTrip server, so it was possible to test only seven of the eight modules.



(a) RTT and clock drift across a 10 min networked audio session. *RTT Spread* is the difference between the maximum and minimum per-client RTT at each point in time.



(b) Excerpt of RTT measurement from Figure 4a (upper plot). Data points represent samples taken once per period of the test signal, i.e. at intervals of (~0.8 s).

Figure 4: RTT and clock drift measurements for seven `JackTripClient` instances, each identified by its IP address. JACK configured to use a hardware audio driver (Focusrite Scarlett 18i20) with a sampling rate of 44.1 kHz and buffer size of 32 samples. For a description of the measurement methodology, see section 4.

vestigation of the feasibility of distributing an ambisonics algorithm would be required. Nevertheless, there may be other spatial audio techniques, and DSP algorithms in general, that would find a good home in a distributed setting.

An expansion on the synchronicity measurements taken in section 4 and a thorough evaluation of the adaptive resampling technique employed would be prudent next steps. The effects of different buffer sizes and read-write delta thresholds on client synchronicity should be assessed, both in terms of mean spread and spread variability. Resampling may introduce phase modulation and will certainly add noise to a signal; quantifying these effects would help to legitimise the approach taken here, or point toward better approaches. Indeed, it may be profitable to pursue other clock discrepancy mitigations, such as modifying Teensy's audio clock, either by altering parameters to the phase locked loop from which that clock is derived or adopting an external PTP or GPS clock source. The addition of a master clock would raise the barrier to entry, but it may prove to be the only way to achieve sample-accurate synchronisation.

Acknowledgments

This project was conducted as part of an internship with the *Emeraude* team at Inria, [13] in Villeurbanne, France. Forming part of the *PLASMA* project, [14] the work was carried out at the Center of Innovation in Telecommunication and Integration of Service (CITI) at INSA-Lyon [15] and at GRAME Studios. [16] An internship salary was kindly provided by Inria.

Thanks: to Guillaume Anthouard and colleagues, who conducted the initial work on running Teensy as a JackTrip client; to Pierre Cochard and the Emeraude team; to Chris Chafe, Fernando Lopez-Lezcano, and Yann Orlarey; and to members of the Teensy and JUCE forum communities.

[13] *Institut national de recherche en sciences et technologies du numérique*, `https://www.inria.fr/`

[14] *Pushing the Limits of Audio Spatialization with eMerging Architectures*, a collaboration between Emeraude and the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University, `https://team.inria.fr/emeraude/plasma/`

[15] *Institut National des Sciences Appliquées*, `https://www.insa-lyon.fr/`

[16] GRAME CNCM (*Centre national de création musicale*), `https://www.grame.fr/`

6. REFERENCES

[1] A. J. Berkhout, D. de Vries, and P. Vogel, "Acoustic control by wave field synthesis," *The Journal of the Acoustical Society of America*, vol. 93, no. 5, pp. 2764–2778, May 1993.

[2] J. Ahrens, R. Rabenstein, and S. Spors, "The Theory of Wave Field Synthesis Revisited," May 2008.

[3] J. Daniel, S. Moreau, and R. Nicol, "Further Investigations of High-Order Ambisonics and Wavefield Synthesis for Holophonic Sound Imaging," 2003.

[4] M. Geier, J. Ahrens, and S. Spors, "Object-based Audio Reproduction and the Audio Scene Description Format," *Organised Sound*, vol. 15, no. 03, pp. 219–227, Dec. 2010.

[5] L. Ward and B. Shirley, "Personalization in Object-based Audio for Accessibility: A Review of Advancements for Hearing Impaired Listeners," *Journal of the Audio Engineering Society*, vol. 67, no. 7/8, pp. 584–597, Aug. 2019.

[6] V. Hardman, M. A. Sasse, M. Handley, and A. Watson, "Reliable Audio for Use Over the Internet," in *Proceedings of INET '95*, 1995.

[7] V. Hardman, M. A. Sasse, and I. Kouvelas, "Successful multiparty audio communication over the Internet," *Communications of the ACM*, vol. 41, no. 5, pp. 74–80, May 1998.

[8] T. Turletti, "The INRIA videoconferencing system (IVS)," *ConeXions*, vol. 8, Jan. 1995.

[9] C. Chafe, S. Wilson, R. Leistikow, D. Chisholm, and G. Scavone, "A Simplified Approach to High Quality Music and Sound Over IP," 2000.

[10] A. Xu, W. Woszczyk, Z. Settel, B. Pennycook, R. Rowe, P. Galanter, and J. Bary, "Real-Time Streaming of Multichannel Audio Data over Internet," *Journal of the Audio Engineering Society*, vol. 48, no. 7/8, pp. 627–641, Jul. 2000.

[11] P. Ferguson, C. Chafe, and S. Gapp, "Trans-Europe Express Audio: testing 1000 mile low-latency uncompressed audio between Edinburgh and Berlin using GPS-derived word clock, first with jacktrip then with Dante." May 2020.

[12] M. Bosi, A. Servetti, C. Chafe, and C. Rottondi, "Experiencing Remote Classical Music Performance Over Long Distance: A JackTrip Concert Between Two Continents During the Pandemic," *Journal of the Audio Engineering Society*, vol. 69, no. 12, pp. 934–945, Dec. 2021.

[13] C. Chafe, S. Wilson, and D. Walling, "Physical model synthesis with application to Internet acoustics," in *2002 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 4, May 2002, pp. IV–4056–IV–4059, iSSN: 1520-6149.

[14] C. Chafe and R. Leistikow, "Levels of temporal resolution in sonification of network performance," 2001.

[15] F. L. Schiavoni, M. Queiroz, and M. M. Wanderley, "Alternatives in network transport protocols for audio streaming applications," in *ICMC*, 2013.

[16] A. Carôt, T. Hohn, and C. Werner, "Netjack – Remote music collaboration with electronic sequencers on the Internet," Jan. 2009.

[17] J.-P. Cáceres and C. Chafe, "JackTrip: Under the Hood of an Engine for Network Audio," *Journal of New Music Research*, vol. 39, no. 3, pp. 183–187, Sep. 2010.

[18] J.-P. Cáceras and C. Chafe, "JackTrip/SoundWIRE Meets Server Farm," *Computer Music Journal*, vol. 34, no. 3, pp. 29–34, 2010.

[19] F. Adriaensen, "Controlling adaptive resampling," in *Linux Audio Conference*, 2012.

[20] "What is Dante? | Audinate | Dante Pro AV Networking." [Online]. Available: https://www.audinate.com/meet-dante/what-is-dante

[21] V. Fischer, "Case Study: Performing Band Rehearsals on the Internet With Jamulus," *URL: https://jamulus. io/PerformingBandRehearsalsontheInternetWithJamulus. pdf*, 2015.

[22] L. Turchet and C. Fischione, "Elk Audio OS: An Open Source Operating System for the Internet of Musical Things," *ACM Transactions on Internet of Things*, vol. 2, no. 2, pp. 12:1–12:18, Mar. 2021.

[23] A. Renaud, A. Carôt, and P. Rebelo, "Networked Music Performance : State of the Art," Jan. 2012.

[24] F. Lopez-Lezcano, "From Jack to UDP packets to sound and back," in *Proceedings of the Linux Audio Conference*, vol. 2012, 2012.

[25] C. Chafe, "I am Streaming in a Room," *Frontiers in Digital Humanities*, vol. 5, 2018.

[26] C. Chafe and S. Oshiro, "Jacktrip on Raspberry Pi," in *LAC-Linux Audio Conference*, 2019.

[27] L. Gabrielli, S. Squartini, E. Principi, and F. Piazza, "Networked Beagleboards for wireless music applications," in *2012 5th European DSP Education and Research Conference (EDERC)*, Sep. 2012, pp. 291–295.

[28] J. A. Belloch, J. M. Badía, D. F. Larios, E. Personal, M. Ferrer, L. Fuster, M. Lupoiu, A. Gonzalez, C. León, A. M. Vidal, and E. S. Quintana-Ortí, "On the performance of a GPU-based SoC in a distributed spatial audio system," *The Journal of Supercomputing*, vol. 77, no. 7, pp. 6920–6935, Jul. 2021.

[29] T. Sporer, "Wave field Synthesis - Generation and Reproduction of Natural Sound Environments," 2004.

[30] F. Winter, J. Ahrens, and S. Spors, "A Geometric Model for Spatial Aliasing in Wave Field Synthesis," 2018.

[31] R. Michon, Y. Orlarey, S. Letz, and D. Fober, "Real Time Audio Digital Signal Processing With Faust and the Teensy," in *Sound and Music Computing Conference (SMC-19)*, Malaga, Spain, May 2019.

[32] H. Marouani and M. R. Dagenais, "Internal Clock Drift Estimation in Computer Clusters," *Journal of Computer Networks and Communications*, vol. 2008, p. e583162, May 2008.