frontiers | Frontiers in Virtual Reality

TYPE Technology and Code
PUBLISHED 08 November 2024
DOI 10.3389/frvir.2024.1391987

OPEN ACCESS

EDITED BY
Alessandro Pozzebon,
University of Padua, Italy

REVIEWED BY
Sven Ubik,
Czech Education and Scientific Net work, Czechia
Jozef Papán,
University of Žilina, Slovakia

*CORRESPONDENCE
Thomas Albert Rushton,
thomas.rushton@inria.fr

RECEIVED 26 February 2024
ACCEPTED 07 October 2024
PUBLISHED 08 November 2024

CITATION
Rushton TA, Michon R, Serafin S, Risset T and Letz S (2024) Networked microcontrollers for accessible, distributed spatial audio.
Front. Virtual Real. 5:1391987.
doi: 10.3389/frvir.2024.1391987

Networked microcontrollers for accessible, distributed spatial audio

Thomas Albert Rushton[1]*, Romain Michon[1], Stefania Serafin[2], Tanguy Risset[1] and Stéphane Letz[3]

[1]Inria, INSA Lyon, CITI, EA3720, Villeurbanne, France, [2]Department of Architecture, Design and Media Technology, Aalborg University, Copenhagen, Denmark, [3]GRAME-CNCM, INSA Lyon, Inria, CITI, EA3720, Villeurbanne, France

State-of-the-art systems for spatial and immersive audio are typically very costly, being reliant on specialist audio hardware capable of performing computationally intensive signal processing and delivering output to many tens, if not hundreds, of loudspeakers. Centralised systems of this sort suffer from limited accessibility due to their inflexibility and expense. Building on the research of the past few decades in the transmission of audio data over computer networks, and the emergence in recent years of increasingly capable, low-cost microcontroller-based development platforms with support for both networking and audio functionality, we present a prototype decentralised, modular alternative. Having previously explored the feasibility of running a microcontroller device as a networked audio client, here we describe the development of a client-server system with improved scalability via multicast data transmission. The system operates on ubiquitous, commonplace computing and networking equipment, with a view to it being a simple, versatile, and highly-accessible platform, capable of granting users the freedom to explore audio spatialisation approaches at vastly reduced expense. Though faced by significant technical challenges, particularly with regard to maintaining synchronicity between distributed audio processors, the system produces perceptually plausible results. Findings are commensurate with a capability, with further development and research, to disrupt and democratise the fields of spatial and immersive audio.

KEYWORDS

spatial audio, networked audio, distributed systems, wave field synthesis, microcontroller, accessibility

1 Introduction

Recent developments in virtual and augmented reality technologies and object-based audio have led to an acceleration in interest in the synthesis of virtual sound fields via approaches such as Wave Field Synthesis (WFS) and Higher Order Ambisonics (HOA) (Berkhout et al., 1993; Ahrens et al., 2008; Daniel et al., 2003; Frank et al., 2015). These techniques call for the deployment of large numbers of loudspeakers, and *in situ* installations of dedicated hardware and software. The costs associated with such installations have seen them largely restricted to the preserve of concert venues, cinemas, and institutions with the means to purchase and operate large-scale systems of this sort.

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

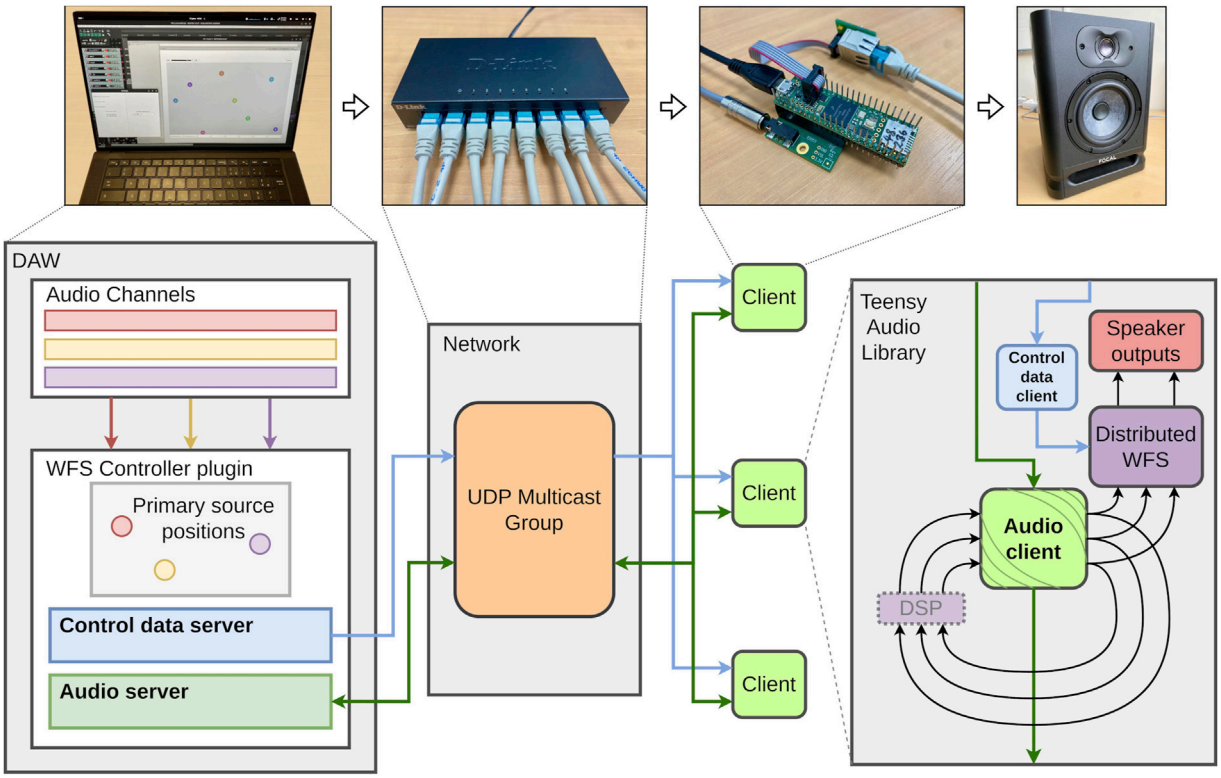

FIGURE 1
Overview of the proposed distributed, networked audio system. A general purpose computer runs DAW software, which in turn runs an instance of an audio plugin that encapsulates servers for audio and control data. The computer is physically connected, via a network switch, to microcontroller-based clients. The clients receive the audio and control data, which are delivered to a distributed audio spatialisation algorithm, and return an audio stream to the server (perhaps via some manner of signal processor). Clients deliver the output signals produced by their instance of the distributed algorithm to loudspeakers.

devices are relatively easy to program with open-source APIs and libraries, and may provide support for communication over ubiquitous computer networking equipment and protocols, an important capability in light of the rise of high-speed ethernet as a standard for multichannel audio transmission (Bakker et al., 2014). A network of such devices could be used to *distribute* the problem of audio spatialisation, permitting a modular, scalable approach that could lower the barrier to entry to what is otherwise a comparatively exclusive branch of audio research. This could in turn pave a more accessible path to research in a variety of domains in which sound field synthesis is a desirable component, including auditory scene personalisation (Geier et al., 2010), auralization and archaeoacoustics (Berger et al., 2023), telepresent videoconferencing (de Bruijn, 2004), and immersive networked music performance (Turchet and Tomasetti, 2023). Further, for implementations such as virtual acoustics, where computationally expensive impulse response convolutions play a part, a distributed system could afford an overall increase in available DSP resources.

In this article we describe the development of a distributed system for spatial and immersive audio. In Section 2 we discuss the technological and scholarly background to this project, including developments in networked audio and embedded hardware platforms, spatial audio techniques, and distributed audio systems. Building upon this background, and prior work on a microcontroller-based networked audio client (Rushton et al., 2023), Section 3 details the development of the proposed system, an overview of which is depicted in Figure 1. With a view to optimising accessibility and interoperability with existing audio tools, the system's server component is encapsulated in an audio plugin suitable for use in a Digital Audio Workstation (DAW). The server delivers audio and control data to a network of microcontroller-based clients via a local area network, and clients, programmed with a parallelised spatial audio algorithm, use the audio and control data to perform their part of the distributed signal process. Minimising inter-client asynchronicity in a distributed audio context is the most significant technical problem, and the network client implementation incorporates strategies for addressing this challenge.

The previous system was not formally evaluated, but, anecdotally, provided support for the holophonic effect of wave field synthesis. A perceptual evaluation of the new system was conducted, and this is described, along with a technical evaluation, in Section 4. Finally, in Section 5, we give an overview of our findings to date and describe our ambitions for future work.

2 Background

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—or "datagrams" — each consisting of a header and a payload. Voice transmission over ARPANET was being conducted as early as 1974 (Schulzrinne, 1992) and the first standard for voice communication over packet-switched networks—the Network Voice Protocol (NVP) — was released in 1977 (Cohen, 1977).

The NVP standard, with control messages for 'calling' and 'ringing', was clearly intended for digital telephony, and communication was the primary focus of networked audio research well into the 1990s. Efforts on supporting real-time voice communication over wide area networks (WAN) centred on *quality of service* (QoS), particularly with regard to the perennial issues of latency, packet loss, and jitter—inconsistencies in the rate of packet transmission (Hardman et al., 1995; 1998). 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 is of greater concern. The late 1990s, with the increasing availability of high-speed internet connections, saw the beginning of research into transmitting uncompressed audio data over the internet (Chafe et al., 2000; Xu et al., 2000). Work of this sort was spearheaded by the *SoundWIRE* project, developed by researchers at McGill University and Stanford University, and took the form of a wide variety of experiments with high quality audio transmission over both WAN and local area networks (LAN). These experiments included LAN-based real-time musical performances (Chafe et al., 2000), concert streaming over WAN (Xu et al., 2000; Chafe et al., 2000), and sonification of QoS via a distributed digital waveguide dubbed the *Network Harp* (Chafe et al., 2000; 2002).

2.1.1 Protocols and systems

VoIP research in the 1990s focused on audio codecs and data compression (Turletti, 1994; Hardman et al., 1998), 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 basic transport layer protocols of the Internet Protocol suite: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Chafe et al. characterised their compression-free system as taking a "simplified approach" to networked audio (Chafe et al., 2000), 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 TCP-based concert streaming. TCP's *connection-oriented*, one-to-one design enables packet flow control mechanisms that guarantee packet ordering and protect against packet loss (Schiavoni et al., 2013; AL-Dhief et al., 2018); at the expense of increased latency, these mechanisms safeguard quality of service, and thus audio fidelity; ideal for a remote concert scenario. UDP by comparison provides no such safeguards, but equally none of the associated computational or temporal overhead. Further, due to its *connectionless* model, *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 (Meyer et al., 2010). Via UDP, SoundWIRE was able to run as a distributed digital waveguide over a WAN spanning around 4,500 km (Chafe et al., 2000).

From the SoundWIRE project emerged *JackTrip* (Cáceres and Chafe, 2010a; Cáceres and Chafe, 2010b), a hybrid system that couples a TCP handshake with audio transmission over UDP, thus sidestepping the overhead of TCP packet flow control. Rather than relying on TCP's built-in mechanisms for stream integrity, JackTrip supplements UDP with a selection of optional buffering strategies that aim to optimise its operation in various network conditions. 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 as the protocol of choice for platforms enabling remote musical collaboration, serving as the basis for NetJACK (Carôt et al., 2009), part of the JACK Audio Connection Kit (a cross-platform audio host), the audiovisual performance streaming platform LOLA (Drioli et al., 2013), Jamulus (Fischer, 2015), Soundjack (Renaud et al., 2007), and other jamming-focused platforms, plus more recent entrants, the closed-source, but ultimately UDP-based networking component of Elk Audio OS (Turchet and Fischione, 2021), for instance. UDP also plays a fundamental role in networked media streaming, being the typical transport-layer protocol behind the Real-time Transport Protocol (RTP), and it features in proprietary networked audio systems such as Dante (Digital Audio Network Through Ethernet) (Dante, 2022).

2.1.2 AoE in the audio industry

In parallel with the work being carried out in academia on SoundWIRE, JackTrip and NetJACK, audio industry bodies—the IEEE (Institute of Electrical and Electronics Engineers) and AES (Audio Engineering Society) standards groups, and companies like Audinate, the creators of Dante—were taking an interest in networked audio. Traditional large-scale audio systems such as those used in broadcast, concert venues and recording studios rely on the installation of unwieldy combinations of analogue hardware and cabling, with many potential points of failure. Seeking literally to lighten the load posed by "hundreds of kilograms" (Bakker et al., 2014) of cabling in analogue audio installations, in the 2000s audio companies were looking to high speed ethernet as a means to simplify the provision of high-quality, multichannel audio in industry settings.

Key to these efforts was the release, in 2002, of the IEEE 1588 standard for the Precision Time Protocol (PTP), a means by which networked computer systems can achieve clock synchronicity (Edison et al., 2002). PTP (another protocol that typically uses UDP for transport) superseded the lower-resolution Network Time Protocol (NTP), and, under ideal conditions, can achieve synchronisation accuracy of sub-microsecond order (Tongzhou and Lunhui, 2022). Synchronisation is achieved via the exchange of timestamped packets, coupled with precise estimates for send and receive times. Precision is best when timestamps can be calculated at the *physical layer*—layer 1 of the Open Systems Interconnection (OSI) model, of which the aforementioned transport layer (layer 3) is a component—i.e., by dedicated timers at the level of the physical network interface. Legacy and low-cost networking equipment do not typically

possess support for hardware timestamping, however (Correll and Barendt, 2005), and devices that do offer such support are markedly more expensive.[1] PTP can be deployed as a software-only implementation (Correll and Barendt, 2005), albeit with impaired accuracy and a protracted clock-convergence period.

Dante, with its promise of low-latency, highly-multichannel audio over wired LAN, and device synchronisation via hardware PTP, has become the *de facto* industry standard in networked audio (Bakker et al., 2014). Bakker et al. refer to Dante as an "open" system, which is true, perhaps, in the sense that companies can incorporate the Dante system into their products under licence from Audinate; from the perspective of the academic community, however, Dante is very much a closed-source initiative and not a suitable platform for research.

In 2011, IEEE released the Audio Video Bridging (AVB, IEEE 802.1) standard (IEEE, 2011), and AES67 followed in 2013 (Hildebrand, 2014). These open technical standards describe *suites* of protocols for tasks such as media transmission, device discovery and synchronisation, and interoperability with other systems. Both use PTP for device synchronisation; AES67 uses RTP for media transmission, whereas AVB uses the data-link layer (layer 2) Audio Video Transport Protocol. Open implementations of AVB and AES67 exist, but, being complex standards featuring many components, such implementations may not be complete, support for embedded platforms is limited,[2] and a reliance on PTP raises the barrier to entry. Ultimately, if an accessible solution is sought, attention must be turned back to the transport layer, and to UDP directly.

2.1.3 Challenges posed by networked audio

Time, especially when dealing with the fine margins posed by real-time audio processing, represents the principal source of difficulty in a networked 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, and may be caused by a number of factors: packet prioritisation rules in the firmware of an ethernet switch, the timing of hardware interrupts for a computer's audio or networking subsystems, and software design decisions relating to network transmission or reception to name but three. In a naive implementation, jitter 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 at the recipient 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, whose operating frequency is subject to manufacturing tolerances, and whose stability is affected by factors such as ambient temperature, and computational load on the system it governs (Marouani and Dagenais, 2008). Relative drift, or *skew*, is the difference in clock rates between two or more systems. Whereas jitter is a transient phenomenon, clock drift is continuous, and 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 professional audio settings, devices may be synchronised via an authoritative clock source such as word clock, or, in a networked setting, via PTP. 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 and resampling (Adriaensen, 2005; Adriaensen, 2012).

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 an accessible, distributed audio application, the ideal computing platform should be small and inexpensive, plus easily and rapidly programmable; of course, it should also provide audio and networking hardware, and, ideally, well-documented APIs for programming and interacting with this hardware.

Recent years have seen the emergence of a number of small, low-cost platforms for embedded systems development, perhaps best known amongst these being the *Arduino* family of microcontroller development boards,[3] whose open-source Software Development Kit (SDK), software libraries, and Integrated Development Environment (IDE) have greatly improved the accessibility of development on embedded systems (Michon et al., 2020). Though support for audio is limited via Arduino devices, a number of audio-specific systems, programmable with the Arduino SDK and IDE, and operable with many Arduino-compatible add-ons (sensors, displays, etc.), have been produced; these include various *ESP32* and *STM32* models, and the *Daisy* and *Teensy* microcontroller ranges. These platforms benefit from the wealth of tools, documentation and support associated with Arduino and the surrounding D.I.Y. and maker communities. Also worthy of consideration are the *Raspberry Pi* and *Bela* platforms. Though these are *Embedded Linux Systems* rather than microcontrollers, they are small-footprint devices, suitable for embedded applications. Bela in particular has been designed with a focus on audio development and interaction via sensors; it can be programmed via a web-based IDE, and the user need not interact with the underlying Linux operating system. Raspberry Pi is less accessible as platform for embedded audio development, and tends to be operated as more of a general-purpose small computer, though support for treating the platform like a microcontroller—taking a *bare metal* approach—is offered via the *Circle* development environment.[4]

1 Consumer-grade, eight-port ethernet switches can cost as little as €20; The cheapest equivalent devices with PTP support cost, at the time of writing, on the order of €150–200, e.g., https://www.fs.com/de-en/products/148180.html — All URLs verified 12/01/2024.

2 See, for example, https://github.com/tschiemer/aes67 and https://github.com/adiknoth/Open-AVB

3 https://arduino.cc/

4 https://github.com/rsta2/circle

TABLE 1 Comparison of selected embedded audio development platforms. Prices as of January 2024.

Platform	Processor	Memory	Price
Teensy 4.1[7]	ARM Cortex-M7 600 MHz	1 MB SDRAM	€32
Daisy Seed[8]	ARM Cortex-M7 480 MHz	64 MB SDRAM	€28
ESP32-LyraTD[9]	Dual core Xtensa LX6 240 MHz	8 MB PSRAM	€19
STM32H747I[10]	ARM Cortex-M7 480 MHz + M4 240 MHz	1 MB RAM	€94
Bela[11]	ARM Cortex-A8 1 GHz[12]	512 MB SDRAM	€190
Raspberry Pi 4[13]	ARM Cortex-A72 1.8 GHz	1 GB–8 GB SDRAM	€30–100

The above systems are typically programmed in C++, with support for audio development provided by libraries such as Daisy's *DaisySP* and Teensy's *Teensy Audio Library*, which each provide audio APIs and a selection of pre-made algorithms for audio synthesis and DSP. Bela, as an alternative to its C++ audio API, can be programmed with the graphical programming language PureData, and Teensy, as a complement to its Audio Library, offers a web-based *Audio System Design Tool*, via which the user may describe an audio system diagrammatically and export the result to C++.

This profusion of tools and platform-specific APIs can render embedded audio development somewhat difficult to approach. A concerted effort has been made, however, by the community behind the *Faust* programming language,[5] to provide support for embedded platforms. Faust is a functional paradigm, audio domain-specific language, that was created to serve as a "viable and efficient alternative to C/C++" (Orlarey et al., 2009) for the development of audio applications on a variety of platforms. In Faust, a user can write high level sound synthesis or DSP code and export the result to C++ that meets the requirements of the audio API on a given target platform. This is achieved via a series of platform-specific "architecture files" and Faust's `faust2[...]` tools,[6] which include `faust2bela`, `faust2teensy`, etc. (Michon et al., 2019; 2020). Developers are thus able to focus on writing audio code, rather than being concerned with the peculiarities of the device or system upon which they wish to deploy their program; further, Faust's support for a variety of embedded platforms facilitates testing and rapid prototyping.

A comparison of selected devices can be found in Table 1. Bela is significantly more powerful than the microcontroller systems, but it is commensurately costly. The Raspberry Pi is also very capable, and a model with 1 GB RAM may cost as little as €30; its operating system stands as an impediment, however, to implementations that seek to prioritise audio functionality above all. Support for bare metal development on Raspberry Pi is not comprehensive, and there is no Faust tool to produce code that is compatible with Circle. Daisy Seed is well-appointed with memory (which is important for DSP algorithms featuring long delay-lines, for example,), but does not provide ethernet support. Teensy 4.1, and the selected ESP32 and STM32 devices support networking via ethernet add-ons, but the ESP32's CPU is underpowered, and the STM32 is unfavourably-priced. Though lacking in memory, Teensy's processor, low price, and networking support make it an attractive candidate platform for a distributed, networked audio implementation. Further, thanks to the presence of a vibrant developer community, utilities such as *TyTools*[14] exist, and can be used to program multiple Teensy devices in a single command—useful for a system distributed amongst many such devices.

One respect in which Teensy is found wanting is audio fidelity. By default, its audio add-on (or *shield*) produces CD quality output (16-bit, 44.1 kHz), falling short of modern requirements for high-quality audio, such as offered by Daisy Seed (24-bit, 96 kHz). While Teensy's sampling rate can be increased, sample resolution is fixed at the level of the device's audio codec. In spite of this shortcoming, and in light of its other, more advantageous qualities, Teensy was selected as the platform upon which to conduct development.

2.3 Audio spatialisation

Audio spatialisation is, plainly put, the practice of distributing sound in space. The spatialisation of *primary sound sources*, e.g., sound captured by microphones, stored as digital audio files, or synthesised in real-time, can be achieved simply by delivering those primary sources to *secondary sound sources*, i.e., loudspeakers or headphones. Exploiting auditory cues, 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 *recreate*) sonic environments for creative and immersive purposes, such as for

5 https://faust.grame.fr/

6 https://faustdoc.grame.fr/manual/tools/

7 https://pjrc.com/store/teensy41.html

8 https://electro-smith.com/daisy/daisy

9 https://espressif.com/en/products/devkits/esp-audio-devkits

10 https://st.com/en/evaluation-tools/stm32h747i-disco.html

11 https://shop.bela.io/products/bela-starter-kit

12 https://beagleboard.org/black

13 https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

14 https://koromix.dev/tytools

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* (Ahrens, 2012; Nicol, 2017), 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 (Ahrens, 2012). For a loudspeaker at position $\mathbf{x} = [\,x \;\; y \;\; z\,]^T$, the time-domain driving signal $\hat{d}(\mathbf{x}, t)$ can be expressed as a convolution of the input signal $\hat{s}_{\text{in}}(t)$ and the driving function $d(\mathbf{x}, t)$:

$$\hat{d}(\mathbf{x}, t) = \hat{s}_{\text{in}}(t) * d(\mathbf{x}, t), \tag{1}$$

where t denotes time.

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.

2.3.1 Periphony and binaural reproduction

The former, *periphonic*, types encompass stereophony and surround-sound systems, consisting of secondary sources in a planar arrangement equidistant from the listening position. These techniques exploit the interaural level difference (ILD) cue, i.e., the difference in perceived amplitude relative to the listener's ears (Pulkki, 1997; Verheijen, 1998; Ziemer, 2020), to encourage the listener to localise sound to a position on the circumference of an arc or circle around the listening position. 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) (Pulkki, 1997), which uses trios of speakers to position phantom sources on the surface of a sphere with the listening position at its origin.

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 (Pulkki, 1997; Verheijen, 1998). Thus the driving function for a time-based panning system is a delay of the form:

$$d(\mathbf{x}, t) = \delta(t - \tau), \tag{2}$$

where τ is the duration of the delay.

The effects of ILD and ITD cues transfer to headphone-based listening, in which case, rather than periphonic, they form a sort of *in-head* localisation (Ahrens, 2012). For a significantly more naturalistic auditory outcome, ILD and ITD cues, when combined with filters describing the dispersive and absorptive effects of the head, torso and outer ears, constitute a Head-Related Transfer Function (HRTF), the key component in what is termed *binaural reproduction*. Binaural recordings are taken either with a dummy head or ear-mounted microphones, and thus the signal for each ear is coloured by the head used during recording, or HRTF measurements can be taken and used to describe filters to be applied to arbitrary signals at playback. Binaural sound is suited to headphone-based listening but may be achieved with loudspeakers if suitable cross-talk cancellation is applied (Kaiser, 2011).

Periphonic approaches are subject to the phenomenon of an ideal listening position, or *sweet-spot* (Nicol, 2017), that is a listening position away from which the spatialisation effect is significantly degraded. Binaural reproduction, if not coupled with head motion tracking, is similarly afflicted by an ideal position and orientation (Verheijen, 1998); further, for faithful reproduction, HRTFs should be individualised (De Poli and Rocchesso, 1998). As such, these techniques are not suited to collective and immersive auditory experiences whereby multiple participants may move freely about their environment.

2.3.2 Physically-inspired techniques

Physical approaches fall into two main types: wave field synthesis (WFS) (Berkhout et al., 1993) and ambisonics (and higher-order ambisonics—HOA) (Frank et al., 2015). 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.

In the case of ambisonics, the sound field is decomposed into "spherical harmonics", spatial functions described by linear sums of directional components of increasing order (Nicol, 2017). Like periphonic approaches, ambisonics suffers from a sweet-spot effect which worsens with attempts to reproduce sounds of higher frequency, but can be mitigated by reproducing higher-order modes and increasing the density of the distribution of secondary sources.

WFS 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 (Mueller, 1971; Berkhout et al., 1993; Belloch et al., 2021) (see Figure 2). WFS is variously termed a form of *acoustic holography* or *holophony* (Berkhout, 1988; Ahrens, 2012). 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 auditory cues related to perceived distance, 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 consequences for spatial resolution; to mitigate the issue of spatial aliasing, whereby sounds of higher frequency cannot be recreated unambiguously (Winter et al., 2018), 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.

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 a sum of input-signal driving-function convolutions (see Equation 1):

$$\hat{d}(\mathbf{x}, t) = \sum_{k=0}^{m-1} \hat{s}_{\text{in},k} * d_k(\mathbf{x}, t), \tag{3}$$

where the driving function d_k is (Ahrens, 2012):

$$d_k(\mathbf{x}, t) = \frac{y_k}{r_k} f(t) * \delta\left(t - \frac{r_k}{c}\right). \tag{4}$$

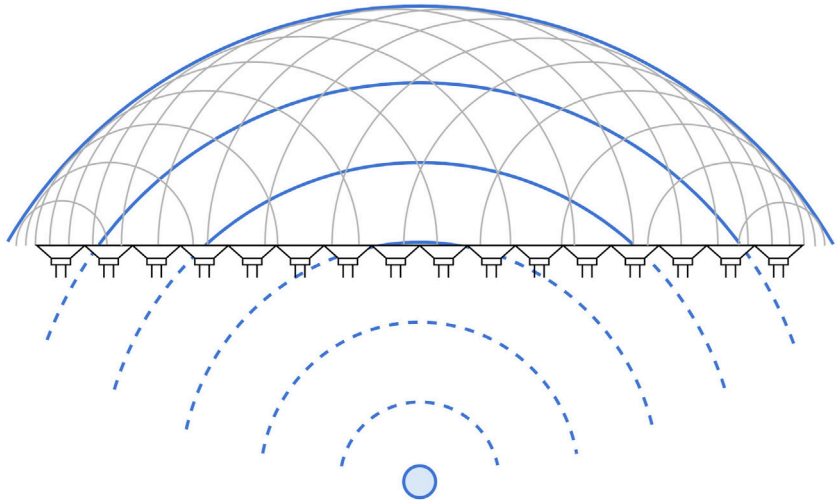

FIGURE 2
Holophony. Huygens' principle states that the propagation of a wavefront can be recreated by a collection of secondary point sources. The bottom of the figure represents a virtual sound field, and the top a real sound field, separated by a row of secondary point sources (loudspeakers). The small circle represents a virtual sound source and the dashed arcs are virtual wavefronts associated with that sound source; the small solid arcs are wavefronts produced by the array of secondary point sources; the large solid arcs represent the propagation of a reconstructed wavefront in the real sound field.

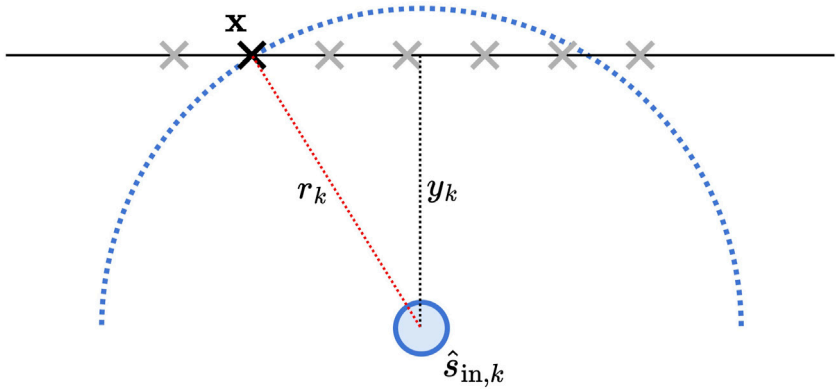

FIGURE 3
The driving signal for the WFS secondary source at position **x**, for virtual primary source $\hat{s}_{\mathrm{in},k}$, is dependent on the distance r_k of the primary source from the secondary source. This corresponds with a propagation delay via the simulated medium of propagation, coupled with a filter describing losses to that medium.

This is, in effect, a physically-informed extension of Equation 2. The *WFS prefilter* $f(t)$ is a function that simulates the absorption of energy into the simulated medium of acoustic propagation. The delta function δ has the effect of delaying the prefilter, and thus $\hat{s}_{\mathrm{in},k}$, 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 3.

2.3.3 State of the art spatial audio installations

As described, for optimal spatial resolution, systems implementing ambisonics and WFS require many output channels; in effect, the more channels, and the greater the loudspeaker-density, the better.

The Multisensory Experience Lab at Aalborg University (AAU), Copenhagen, hosts a 64-channel, square-array WFS system covering an area of 4 m × 4 m (Grani et al., 2016). It is driven by a Mac Pro desktop computer, connected, via a USB MADI (Multichannel Audio Digital Interface) interface, to two 32-channel MADI to analogue converters. Though this system's WFS engine is provided by open-source *WFSCollider* software,[15] being a centralised system, the computer that co-ordinates its operation is powerful — 12-core CPU, 64 GB RAM—and was costly at the time of purchase; likely on the order of several thousand Euros. With further regard to cost, the current equivalent MADI to analogue

15 https://github.com/GameOfLife/WFSCollider

converter models are priced at roughly €5,000 apiece. Excluding the cost of loudspeakers (and the gantry upon which they are mounted) one may reasonably place an estimate of €250 per output channel on this system.

The world's largest dedicated WFS system, at TU Berlin, features over 800 output channels served by a distributed cluster of fifteen computers acting as audio nodes and two additional control computers (Baalman et al., 2007).[16] The fifteen audio nodes of the TU Berlin system are each equipped with a MADI audio interface, connected to a MADI to ADAT bridge; these can, at the time of writing, be purchased for around €1,400 and €4,500 each, respectively, for a total of €88 500. Imagining €2000 per computer, a conservative estimate of €150/channel may be reached, however the expense associated with this system is difficult to assess as it is tied to the concert hall space in which it resides. Indeed, one thing besides expense that unites the systems at AAU and TU Berlin, plus the 339-speaker hybrid WFS/ambisonics system at IRCAM (Paris)[17], the 512-channel WFS system at the Rensselaer Polytechnic Institute (NY, United States)[18], and the behemoth installation at the *Sphere* (Las Vegas, United States)[19] is their *in-situ* nature; these are site-specific systems with, at best, limited flexibility.

What one is afforded by WFS installations of this scale and expense, however, is high quality audio reproduction with tantamount to perfect output synchronicity; the integrity of the holophonic effect of WFS is, unavoidable matters of spatial aliasing aside, guaranteed.

2.4 Distributed audio systems

As alluded to earlier in this section, our aim is to distribute the problem of audio spatialisation, and it is worthwhile to revisit why this is the case. In the broadest terms, a distributed system is *"a collection of independent entities that cooperate to solve a problem that cannot be individually solved"* (Kshemkalyani and Singhal, 2011). An ideal distributed system is characterised by: *modularity*, being comprised of separate, interchangeable entities; *scalability*, being extensible without incurring a performance penalty to the system as a whole, and; *improved performance/cost ratio*, since it can be constructed to meet the proportion that circumstances require, with the minimum degree of redundancy. Additionally, for algorithms that can be effectively *parallelised*, a distributed system may provide more aggregate computational power than its centralised counterpart.

Where a distributed system may suffer, by contrast, is in terms of reliability. Nodes in a distributed computational system must be served with power and access to the data they require in order to operate, which entails a proliferation of potential points of failure. The other side to the coin of modularity is a concern regarding the programmability of such a system; ensuring that all entities possess up-to-date instructions for operation may not be trivial. Further, some algorithms may be better suited to parallelisation than others; efficient use of increased computational resources is not guaranteed.

Distributed audio processing is by no means a matter without precedent. (Indeed, the WFS installation at TU Berlin described in Section 2.3.3 is of course a distributed system, albeit not an especially accessible one.) A selection of prior work in distributed DSP and audio spatialisation, plus systems incorporating microcontrollers and single-board computers is detailed below.

2.4.1 State of the art distributed audio systems

Applications of SoundWIRE to what its creators termed *Internet Acoustics* (Chafe et al., 2002) clearly stand as examples of distributed audio processing. These include a network reverberator (Chafe, 2018), or *"transcontinental echo chamber"* (Chafe et al., 2000), 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 real time in graphical or textual form due to the ephemeral nature of the phenomena of 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 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, topics that came to the fore in computer music research during the COVID-19 pandemic (Bosi et al., 2021; Sacchetto et al., 2021). That being said, more recent work on Internet Acoustics in an embedded context has yielded a port of JackTrip to the Raspberry Pi (Chafe and Oshiro, 2019).

Examples of distributed music production systems include the work of Lago and Kon (Lago and Kon, 2003), whose UDP-based system featured clients that acted as delegates for audio processing, and Gabrielli et al. (2012), who demonstrated a wireless relay of audio and control-data processors. Latency was an important metric for the latter system, and the authors measured latency via transmission round-trip times using a low frequency sawtooth wave as a timer (see Section 4.1 for an application of this technique).

Distributed approaches to audio spatialisation include Lopez-Lezcano's "network sound card" (Lopez-Lezcano, 2012), and embedded implementations as described by Devonport and Foss, (2019) and Belloch et al. (2021). The latter two address aims closely aligned with the work described here, but are based on costly computing platforms. Devonport and Foss used AVB, and thus PTP, for synchronisation; Belloch et al. employed a GPU-based hardware platform, reporting client synchronisation to the millisecond range—likely not sufficient for timing-critical audio spatialisation effects.

Also of interest is the OTTOsonics project (Mitterhuber et al., 2022); its emphasis on a fully-costed, flexible, 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. A full 24-channel OTTOsonics system, including speakers and audio interface, is costed at around €2,600 (€108.33/channel), however, which certainly places it favourably when compared with state-of-the-art spatialisation systems.

16 See also https://tu.berlin/en/ak/research/projects/wellenfeldsynthese-fuer-einen-grossen-hoersaal and WFS speaker module produced by Four Audio for installation at TU https://four-audio.com/en/products/wfs/

17 https://www.ircam.fr/article/connaissez-vous-lespace-de-projection

18 https://empac.rpi.edu/about/building/venues

19 https://holoplot.com/insights/case-studies/msg-sphere-case-study

3 Methods

As illustrated in Figure 1 (page 2), the proposed system is 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 DSP algorithm). In the subsections that follow, these elements are described in detail; finally, in Section 3.4, an overview of the system and its operation is provided.

3.1 The networked audio server

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 a sender and a receiver task for each client that connects (Cáceres and Chafe, 2010a); notionally this entails, should enough clients connect, exhaustion of all available network bandwidth; as such, a unicast system does not meet the requirement of scalability as described in Section 2.4.

A multicast NetJACK server was considered, but creating a client implementation on what is essentially a bare-metal platform in the shape of the Teensy, was not practical. Further, due to a break in compatibility with Mac OS X systems, JACK-based approaches are not truly cross-platform.[20] Prioritising simplicity, in the form of an audio server with minimal dependencies and a very specific task to achieve, we embarked upon the design of a bespoke multicast networked audio server.

3.1.1 Designing a networked audio protocol

Dependent on the intended application, and if assumptions can be made about matters such as sampling rate and bit resolution, a *no-protocol* approach, such as described by Lopez-Lezcano (Lopez-Lezcano, 2012), may be a viable one. To improve the flexibility of the system and render it somewhat future-proof, however, a simple packet header was devised. Its structure is given in Listing 1.

The resulting six-byte header comprises a two-byte (unsigned 16-bit integer) packet sequence number, to be incremented by the sender, plus four further bytes describing the structure of the audio data in the packet. Commonly-encountered sampling rates, and buffer sizes greater than 255, cannot be represented by unsigned eight-bit integers, so these are supported by enumerations inspired by those used by JackTrip.[21]

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

Listing 1. Packet header structure.

`BufferSize` describes the number of audio frames per packet[22] as the *n*th power of 2; for example, the enumeration `BufferSizeT` features a member `BufferSizeT::BUF16`; 16 being the fourth power of 2, this member is assigned the number 4. The `BitResolution` field could be used to transmit one of 8, 16, 24, or 32 as-is; there is a utility, however, when decoding a packet, in knowing the number of *bytes* per audio sample, so this is the number that is represented, e.g., `BitResolutionT::BIT16` takes the value 2, the number of bytes in a 16-bit integer.

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

One piece of information that is not stated in the packet header is the manner in which audio samples in the packet should be interleaved. The assumption taken—indeed, the same assumption used by JackTrip—is that audio data is channel-interleaved, i.e., audio data consists of a contiguous block of samples for one channel, followed by a block for the next channel, and so on.

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[23] and is encapsulated as a class called `NetAudioServer`. Initial development was conducted on a basic console application, and later work targeted a DAW plugin comprising a consolidated audio server and wave field synthesis controller.

The `NetAudioServer` instance expects to receive blocks of multichannel audio from an audio application's main processing loop. It sets up network *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"* (Kerrisk, 2023) to which a type—on Linux

20 A successor to the defunct CoreAudio/JACK bridge has been proposed but remains unrealised: https://github.com/jackaudio/jack-router/blob/main/macOS/docs/JackRouter-AudioServerPlugin.md. This issue of course also affects the viability of the JackTrip-based approach.

21 https://github.com/jacktrip/jacktrip/blob/v1.6.8/src/AudioInterface.h\#L56

22 Often used interchangeably with the word *sample*, a *frame* represents the samples for all channels for a given sample instant; thus the number of frames in a network packet or audio buffer is the number of samples divided by the number of channels.

23 JUCE 7.0.5 https://github.com/juce-framework/JUCE

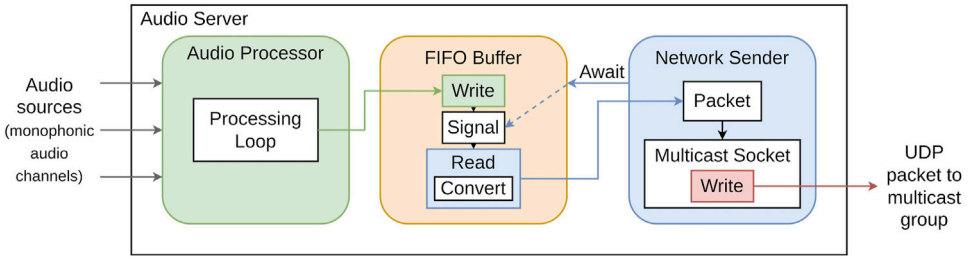

FIGURE 4
Overview of operation of the networked audio server. The network sender awaits notification of readiness to read samples from a first-in-first-out buffer of audio samples. The audio processor receives audio channels from a multichannel source (e.g., a DAW); at each iteration of its processing 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 desired bit resolution and byte order and bundled into a UDP packet which is then written to the network.

systems, `SOCK_STREAM` for TCP, `SOCK_DGRAM` for UDP—can be assigned. To avoid potentially blocking the audio application's main 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 the sender thread that a block is ready for transmission. The sender thread, which as 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. Byte order, or *endianness* (Cohen, 1981), is also specified as part of this conversion. Though network byte order is typically big-endian, or Most Significant Byte (MSB) first, it was found that little-endian transmission meant that samples could be decoded trivially at the client side.[24] Upon receiving the requested samples, the sender thread writes these to its socket, which has been configured to connect to a UDP multicast group. This process is illustrated in Figure 4.

```
        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...)...
```

Listing 2. Network capture: ethernet frame containing a UDP audio packet.

Listing 2 shows an example network capture of an outgoing audio packet. Bytes `0x0000` to `0x0029` comprise the headers for the data link (ethernet), network (IPv4), and transport (UDP) OSI layers 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 (Meyer et al., 2010). The six subsequent bytes are the header inserted into the packet by `NetAudioServer`. In Listing 2 these are:

- `0x1cdf`: a sequence number (little-endian) of 7391_{10};[25]
- `0x04`: buffer size 4 corresponding with `BufferSizeT::BUF16`;
- `0x02`: sampling rate 2 corresponding with `SamplingRateT::SR44`;
- `0x02`: bit resolution 2 corresponding with `BitResolutionT::BIT16`;
- `0x02`: 2 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 frames, 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 — $32\,767_{10}$ — at which point it wraps around to zero and repeats. This test signal serves two important purposes. First, its impulse-like behaviour once every 32,768 samples (roughly .74 s at a sampling rate of 44.1 kHz) is useful for taking basic synchronicity measurements, e.g., involving connecting two clients' audio outputs to an oscilloscope. Second, this numerically-predictable signal serves as a means to inspect the integrity of the audio server algorithm, and to verify that the

24 Endianness is a thorny issue—just consider Danny Cohen's *...Plea for Peace* (Cohen, 1981). To appeal momentarily to authority, JackTrip too transmits audio data (and port numbers, etc.) little-endian, "network byte order" notwithstanding.

25 Subscript 10 is employed here to indicate a decimal number.

expected sample interleaving and endianness is employed. Inspecting the first sixteen samples of the first audio channel 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.

The purpose of the server's receiver thread is to poll its socket for traffic reaching the multicast group from connected clients. Clients are programmed to return a stream of packets of audio data to the multicast group, and the receive thread uses the existence of a such a stream, with a given origin IP address, to indicate the presence of a client at that address. If a client fails to return a packet for more than 1 s it is considered disconnected. Clients could announce their presence with any periodic UDP transmission, but the possibility of returning audio data facilitates the measurement of client synchronicity via transmission round trip times (see Section 4.1).

3.1.3 Transmission considerations

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., without being split into multiple sub-packets. Two bytes are allocated to the 'Total Length' field of the IPv4 header, which suggests an MTU of $2^{16} - 1 = 65{,}535$ bytes; in practice, however, the data link layer imposes a basic limit of 1,500 bytes on the payload of an ethernet frame (Schiavoni et al., 2013; IEEE, 2018).

With the headers for the data link (Ethernet), network (IPv4), and transport (UDP) layers accounted for, plus the audio header described above, in principle 1,452 bytes remain in each packet for audio data. Assuming 16-bit resolution, and the transmission of one UDP packet per audio buffer, data for up to 90 audio channels can be transmitted at a buffer size of 16 frames without fragmentation, or up to 45 channels at 32 frames.

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.[26]

The task of the 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.

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. Networking operations are conducted from the method `NetJUCEClient::loop`. Avoiding conflicts with audio functionality, this method is called from Teensy's top level `loop()` function. A valid Teensy program must define a function by this name, and it is called repeatedly from the body of a non-terminating `while` loop throughout operation.

The two sets of operations are linked by way of an intermediate buffer, similar to the FIFO employed by the server. The client attempts to receive packets from, and, if it has generated a packet's worth of audio data, send a packet to, the multicast group on each call to `loop()`, with audio samples from incoming packets written to the intermediate buffer (see Listing 3). The client also performs a periodic check for the presence of the server, and, as described in Section 3.2.1, makes adjustments to its audio clock. When multiple clients are present, there are consequently multiple streams of audio packets reaching the multicast group. To avoid ambiguity and unnecessary packet reads at the client side, server and clients transmit audio data to the group on differing port numbers.

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

    checkConnectivity();

    send();

    adjustClock();
}
```

Listing 3. Loop method of the networked audio client implementation.

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

    handleAudioInput();
}
```

Listing 4. Update method of the networked audio client implementation.

26 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.

On each audio interrupt, the client reads from the intermediate buffer to produce samples for audio output. It also takes samples reaching its audio inputs and adds those to a packet to be sent to the multicast group at the earliest subsequent call to `NetJUCEClient::loop` (Listing 4). The client's inputs can receive samples from any Teensy Audio Library object to which it has been connected programmatically; for round-trip time measurements the client's audio outputs were routed back to its inputs. An illustrative timeline of client-server interaction is depicted in Figure 5.

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

3.2.1.1 Jitter compensation

Similar to the approach taken in prior work (Rushton et al., 2023), clients monitored their intermediate buffer for the difference between its write and read positions, using a delay-locked loop to keep this difference within an interval of one audio buffer's worth of frames. This was achieved by way of setting thresholds for the read-write difference, and adjusting the read-position increment if the difference 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 it 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.

3.2.1.2 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 (PLL) 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.

3.3 The audio spatialisation algorithm

WFS was chosen for implementation due to the comparative ease with which the WFS algorithm can be parallelised. Equations 3 and 4 (page 10) illustrate that the driving signal for a given secondary point source is dependent only on the signals and relative positions of the virtual primary sources, and is independent of the driving signals for the other secondary sources.

With some modifications, e.g., the possibility to specify speaker spacing parametrically, the WFS algorithm from (Rushton et al., 2023) was reused. This algorithm, facilitating the simulation of virtual primary sound sources, was written in Faust and compiled to a C++ class compatible with the Teensy Audio Library via Faust's `faust2teensy` utility. Hardware modules were connected to a general purpose computer via a USB hub and the `tycmd` utility from the TyTools suite (see Section 2.2) was used to ensure that all modules were programmed with the same instructions.

As illustrated in Figure 3, producing the driving signal for a WFS secondary source at position $\mathbf{x}$ entails applying a delay to an audio signal $\hat{s}_{\text{in},k}$, representing the kth virtual primary source. This delay is based on the distance r_k between $\mathbf{x}$ and the desired virtual position of $\hat{s}_{\text{in},k}$. In its distributed form, the WFS algorithm, informed of the position in the array of the two loudspeakers for which it is responsible, computes only the delays for each primary source with respect to those two loudspeakers, i.e., for the kth virtual source, the nth hardware module computes $r_{k,\mathbf{x}_{2n}}$ and $r_{k,\mathbf{x}_{2n+1}}$. To reduce 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.

For the WFS prefilter, r_k was mapped to an inverse square law for frequency-independent amplitude loss to the virtual medium of propagation, and to the cutoff frequency of a two-pole lowpass filter defined by Faust's `fi.lowpass` function.[27] Adopting a modified version of Equation 4, the driving function becomes:

$$d_k(\mathbf{x},t) = f(t,r_k) * \delta\left(t - \frac{r_k - y_k}{c}\right). \tag{5}$$

3.3.1 Modularity and maximum delay

The reduction in the maximum delay length represented by the subtraction of the longitudinal distance component in Equation 5 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 numerous 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, to the width of the speaker array. It was observed that, for eight virtual sources and eight hardware modules, the maximum speaker spacing permissible lay at around .4 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

27 https://faustlibraries.grame.fr/libs/filters/\#filowpass

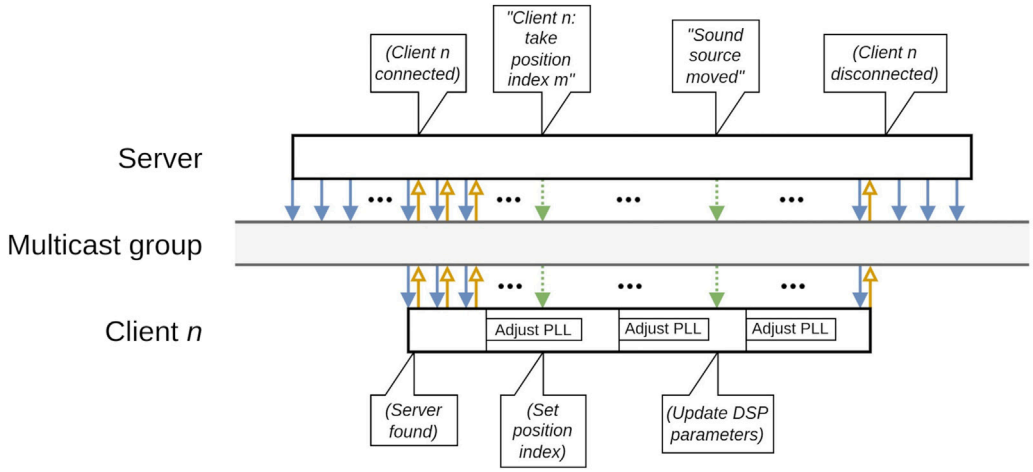

FIGURE 5
Example timeline of interaction between the server and a client, via the UDP multicast group. Solid arrows indicate audio data being sent from the server to the multicast group; arrows with open heads indicate audio data being sent from the client back to the multicast group; arrows with dotted tails represent control data. Note that the server transmits to the multicast group irrespective of the presence of any client.

presumption is that this places significant limits on the modularity of the system. Teensy's memory capacity can be extended by attaching up to two inexpensive PSRAM chips for a further 16 MB of memory. These chips must be soldered onto the Teensy board, however, and the suitability of such additional memory for rapid access, such as is required in an audio DSP algorithm, remains to be investigated.

3.3.2 Controlling the WFS algorithm

Parameter values are delivered to the Faust algorithm in the form of Open Sound Control (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. Source positions are described as coordinates in a two-dimensional plane, with x and y components, each normalised to the range [0,1], transmitted separately. The width of the speaker array is inferred from the speaker spacing (in metres), multiplied by the number of speaker-intervals in the array, i.e., one fewer than the number of speakers. For the proposed implementation, the number of speakers is known to the server and clients at compile time; with further development this could be made specifiable at runtime. Similarly, at the time of writing, the longitudinal depth of the virtual sound field is hard-coded into the clients and will be generalised in a future iteration of the system.

Listing 5 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 big-endian 32-bit floating point number, `0x3d1b5fa2`, approximately 0.038_{10}.

```
        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 5. Network capture: ethernet frame containing a UDP control data packet.

3.4 System overview

3.4.1 Hardware setup

The networked audio server runs on a general purpose computer. Throughout development, testing and evaluation, 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, the hope being that external hardware would provide more consistent audio interrupt timing, thus minimising jitter originating at the server.

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 networked audio clients, this switch was daisy-chained to an additional switch (D-Link DES-1008D). Teensy 4.1 hardware modules, assembled as per Figure 6, 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 loudspeakers.

3.4.2 Software system

Server-side, the software system consists of a VST plugin running in Reaper digital audio workstation software.[28] 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

28 https://reaper.fm/

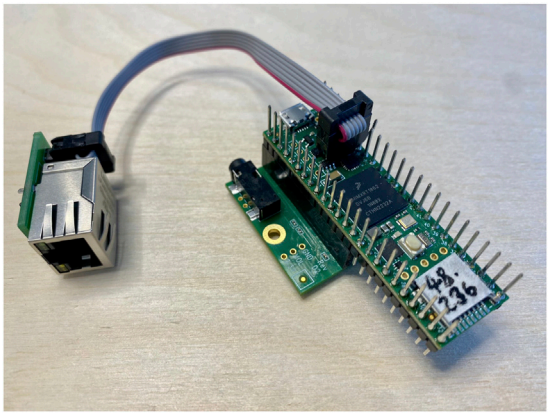

FIGURE 6
A hardware module consisting of Teensy 4.1 microcontroller (labelled with the last 2 bytes of its serial number-derived IP address), connected via headers to an audio shield and via ribbon cable to an ethernet shield.

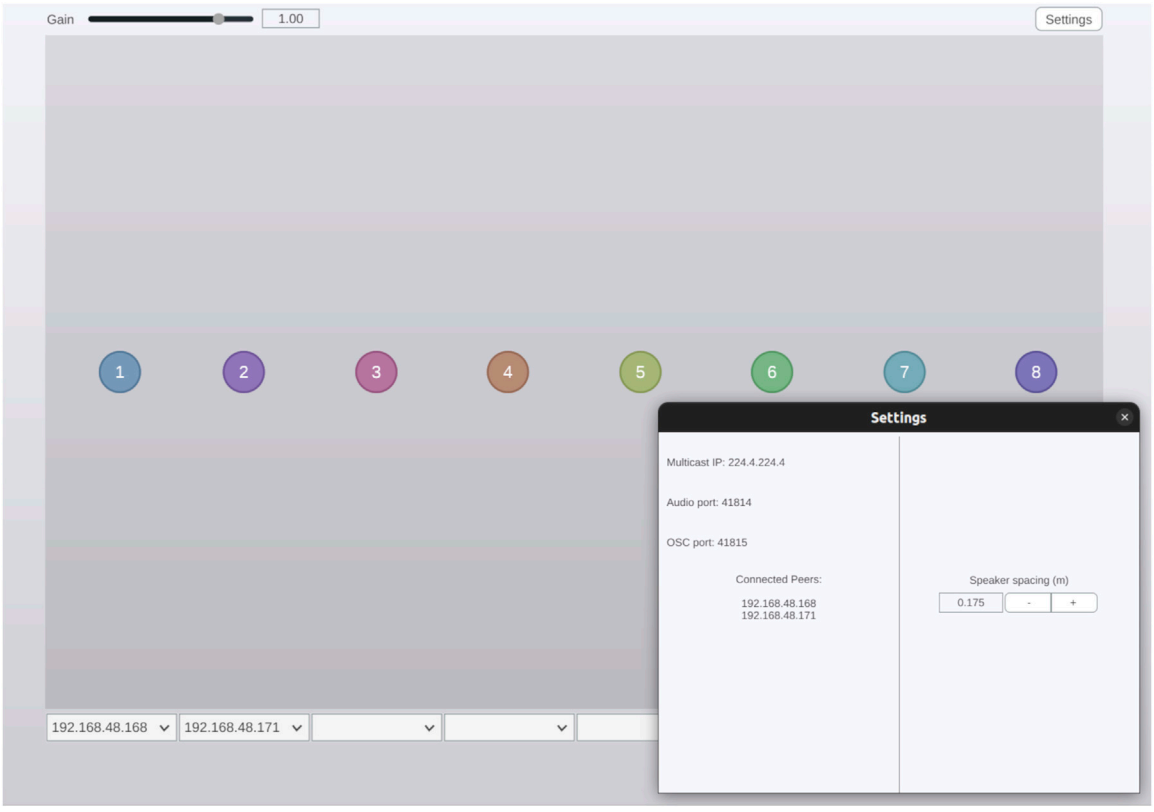

FIGURE 7
User interface for the WFS controller DAW plugin, with modal settings window visible. The interface consists of an X/Y control surface, with eight nodes representing the coordinates, normalised to $x, y \in [0, 1]$, of sound sources in a virtual sound field. Dropdown menus at the bottom of the interface correspond with hardware module positions in the loudspeaker array; there are eight such menus in total, each associated hardware module producing output for two loudspeakers. The settings window facilitates specifying the speaker spacing, and shows a list of connected network peers.

parameter automation via the DAW, or manually via a graphical user interface (see Figure 7). The audio and control data servers send streams of UDP packets to a UDP multicast group.

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.

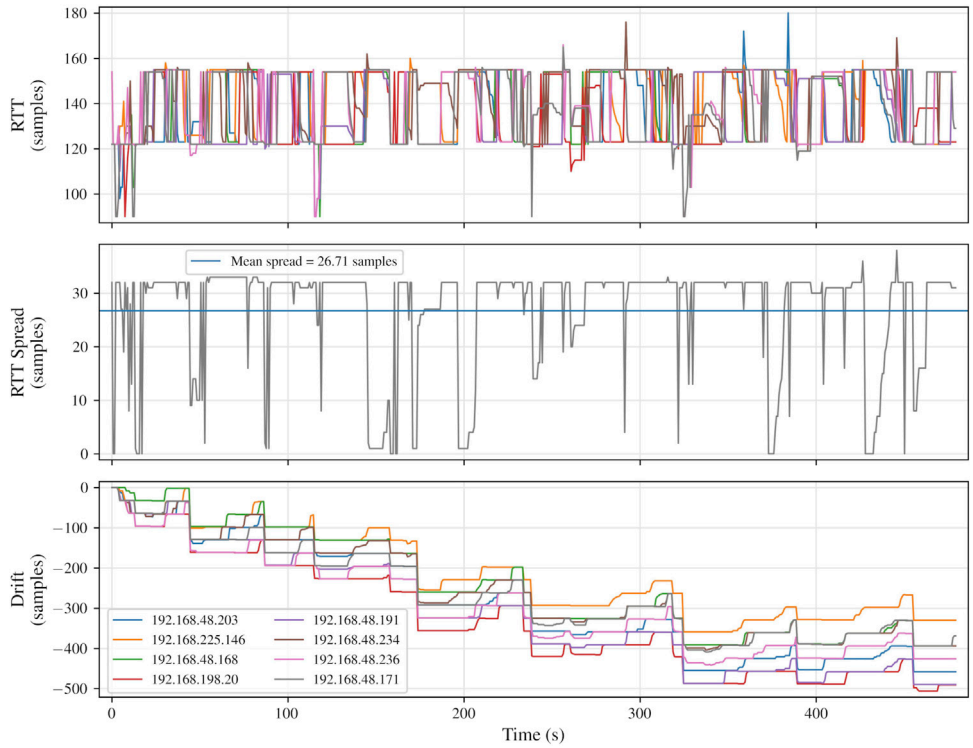

FIGURE 8
Illustration of the use of a test signal, a unipolar sawtooth wave, to measure round trip time. Subtracting the return signal from the outgoing signal gives the time (in samples) between transmission and reception.

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.

4 Results

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 Section 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 a 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 is of limited worth, however, as a technical exercise in isolation; the subjective assessment of listeners may help identify the most critical aspects of the technical implementation and guide future development.

4.1 Technical evaluation

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 (Rushton et al., 2023; Gabrielli et al., 2012).

4.1.1 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 the signal simply returned it immediately to the group to be read 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:

$$\text{RTT} = \begin{cases} x_{out} + \max_{int16} - x_{ret}, & x_{out} < x_{ret}, \\ x_{out} - x_{ret}, & \text{otherwise}, \end{cases} \tag{6}$$

where $\max_{int16}$ is the maximum value representable by a signed 16-bit integer, `0x7fff` ($32\,767_{10}$).

The resulting value is the number of samples elapsed between transmission and reception (see Figure 8). 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.

4.1.2 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 as per Equation 6)

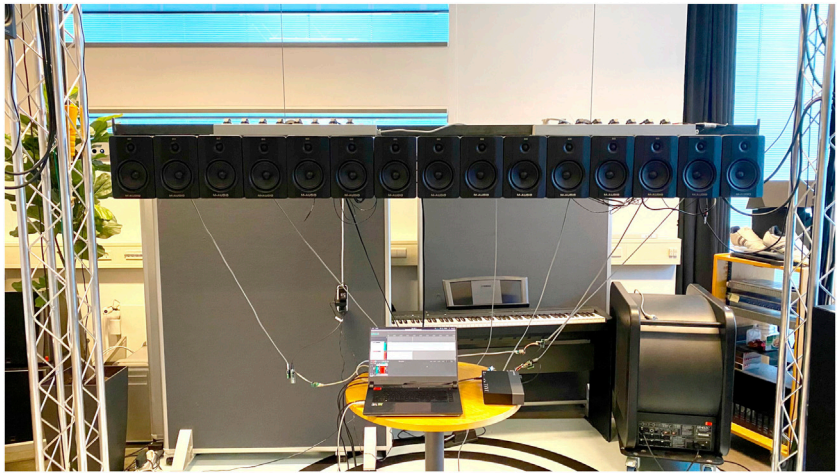

FIGURE 9
Round-trip time, RTT spread, and clock drift measurements for eight networked audio clients, for a networked audio session of 8 minutes' duration. Audio buffer size, 16 frames, sampling rate 44.1 kHz. The legend in the bottommost plot applies also to the upper plot. Round-trip time, in samples, measured at the server, and found as the difference between an outgoing sawtooth wave and its returning counterparts from each of eight connected clients. Round-trip time spread found as the range (RTT_{max} – RTT_{min}), in samples, at each point in time, of round-trip times reported for all eight clients. Mean spread is the arithmetic mean of RTT spread values taken across the entire test. Drift, in samples, found as the difference between an outgoing sawtooth wave and a sawtooth wave generated on each client, that difference returned to the multicast group for consumption by the server.

returned to the multicast group for consumption by the server. The incoming signal and the one being generated on a given client should, under ideal conditions, be out of phase by some constant value; if this value changes then relative drift has occurred between server and client. The client-side clock-adjustment strategy was designed to minimise the reliance on the adaptive resampling approach that it complements; low drift would be indicative of the effectiveness of that strategy.

Initial RTT and relative drift measurements for eight clients are shown in Figure 9. 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 with approximately 282 μs. RTT is clustered around a respectable 190 samples (~4.3 μs).

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 9 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. 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 white Gaussian noise signal 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 10A; ideally (subject to the frequency 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 frames 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 3.1.3). Recalling, however, that previous work (Rushton et al., 2023) had employed a 32-frame audio buffer, equivalent measurements were taken for the larger buffer size, the results of which are depicted in Figures 10B, 11.

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, equating to ~626 μs, is comparable with results from prior work. Importantly, however, and as demonstrated in Figure 10B, the rate of relative inter-client temporal movement was much improved by the switch to a 32-frame buffer. Although exhibiting similar visual striations to the spectrogram for the test at 16 frames, fluctuations occur less frequently, and seemingly more gradually. Indeed, subjectively-speaking, the disruption caused by the phasing effect that afflicted the 16-frame 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 size of 32 frames that was exposed to perceptual evaluation.

Clock drift measurements in Figures 9, 11 exhibit comparable trends. Increasing negative drift over time is indicative of the clients running faster than the server. Visually, there is evidence that client

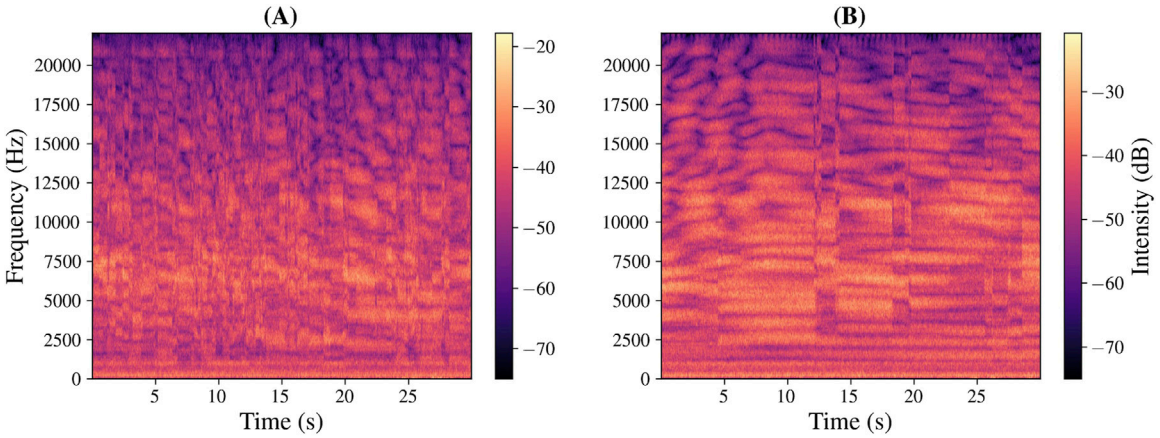

FIGURE 10
Magnitude spectrograms of ambient, monophonic recordings of a reproduction of white Gaussian noise by a group of eight networked audio clients driving an array of fifteen loudspeakers spaced at intervals of .175 m. Capacitor microphone placed ~ 2 m from the speaker array. Audio buffer size **(A)** 16 frames; **(B)** 32 frames.

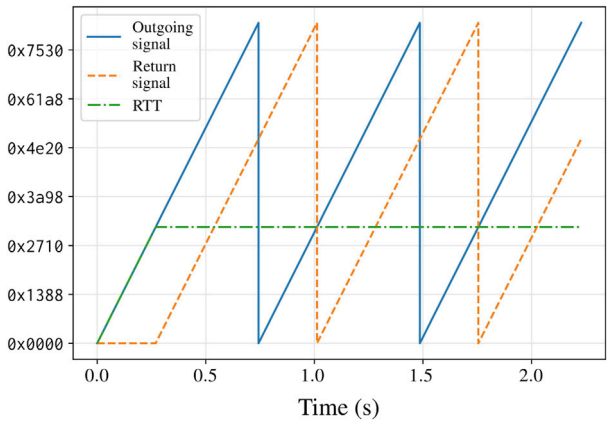

FIGURE 11
Round-trip time, RTT spread, and clock drift measurements for eight networked audio clients, for a networked audio session of 8 minutes' duration. Audio buffer size, 32 frames.

clocks adjust to approximate parity with the server for periods of time, perhaps falling slightly slower (e.g., the drift plot in Figure 9, between 90 and 160 s), but intermittently demonstrate leaps, the largest of which are in the negative direction.

4.1.3 Discussion

The temporal clustering and polarisation seen in Figures 9, 11 is indicative of two points for improvement with regard to technical implementation: the read-write difference threshold strategy may be insufficiently forgiving, forcing the read position into rapid changes in response to periods of jitter; and without an authoritative clock to indicate to the clients when each block of audio data should be output, even if clock rates were perfectly aligned, there is nothing to guarantee agreement of the timing of audio interrupts at the client side.

The steps seen in the drift plots in Figures 9, 11 appear (particularly in Figure 11) to occur in multiples of the audio buffer size. This suggests either packet loss or, more likely,[29] moments of pronounced jitter, causing clients to rapidly reduce (or, less commonly, increase) their read-position increment to maintain the read-write delta. One may expect such a phenomenon to be followed by an immediate rebound, but this appears to be a more gradual process. The overall trend, for thiscombination of server and clients at least, is for clients to run faster than the server; the clock adjustment strategy employed relies on inferring time from the rate of packet transmission, which may not offer sufficient temporal resolution for accurate drift compensation.

29 During many hours of testing, no instance of packet loss was reported by any of the clients.

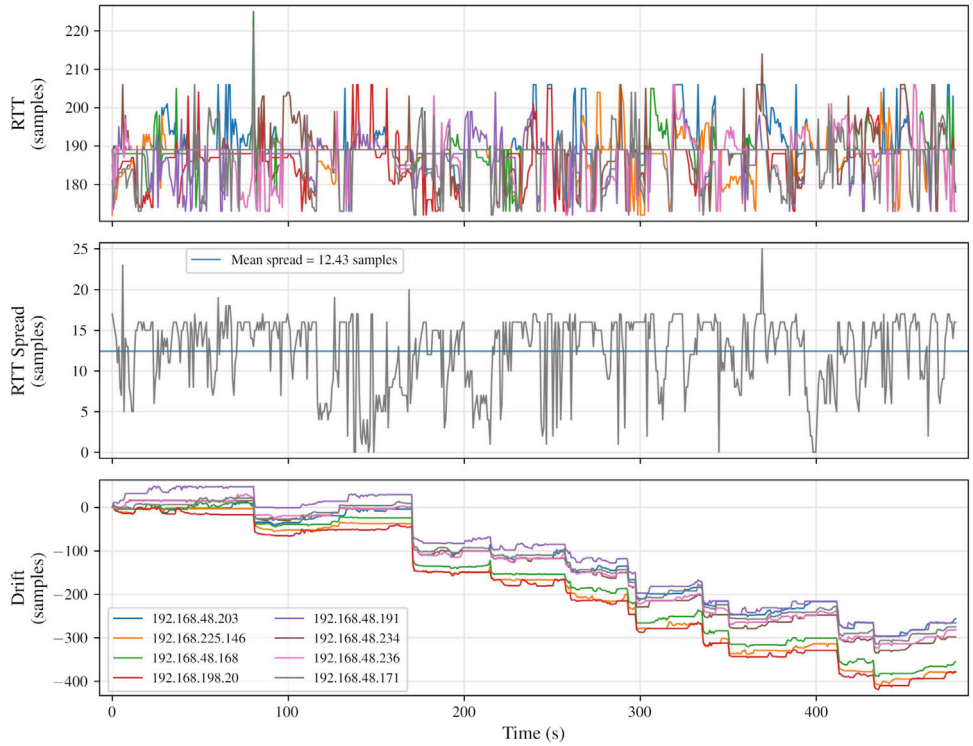

FIGURE 12
System configuration for technical and perceptual evaluation. Eight hardware modules connected to fifteen loudspeakers—seven of the modules produced output for two loudspeakers each; the final module used only its first output channel.

4.2 Perceptual evaluation

The WFS system was subjected to an informal perceptual evaluation, a localisation experiment of a similar form to that presented by Verheijen (Verheijen, 1998, ch. 6), albeit with the inclusion of simulated distance as well as lateral position. Participants were presented with a virtual sound source at various locations and asked to indicate, on a diagram of the virtual sound field, the point from which they estimated the sound had emanated. The informality of the experiment arose in 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 12) 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.

It was felt that listeners would be most comfortable localising a naturalistic sound, so, rather than use bursts of white noise as in (Verheijen, 1998), and 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 .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).

The system was presented to eight participants; a mixture of masters and PhD students aged between 22 and 38, with knowledge of audio and interactive computer systems. 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 centre of the driver of the leftmost speaker to that of the missing 16th loudspeaker; drivers lay at intervals of .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 played, and repeated at the participant's request. Each participation was five to 10 minutes in duration. Details of the source positions for each test, and responses for the eight participants, are displayed in Figure 13.

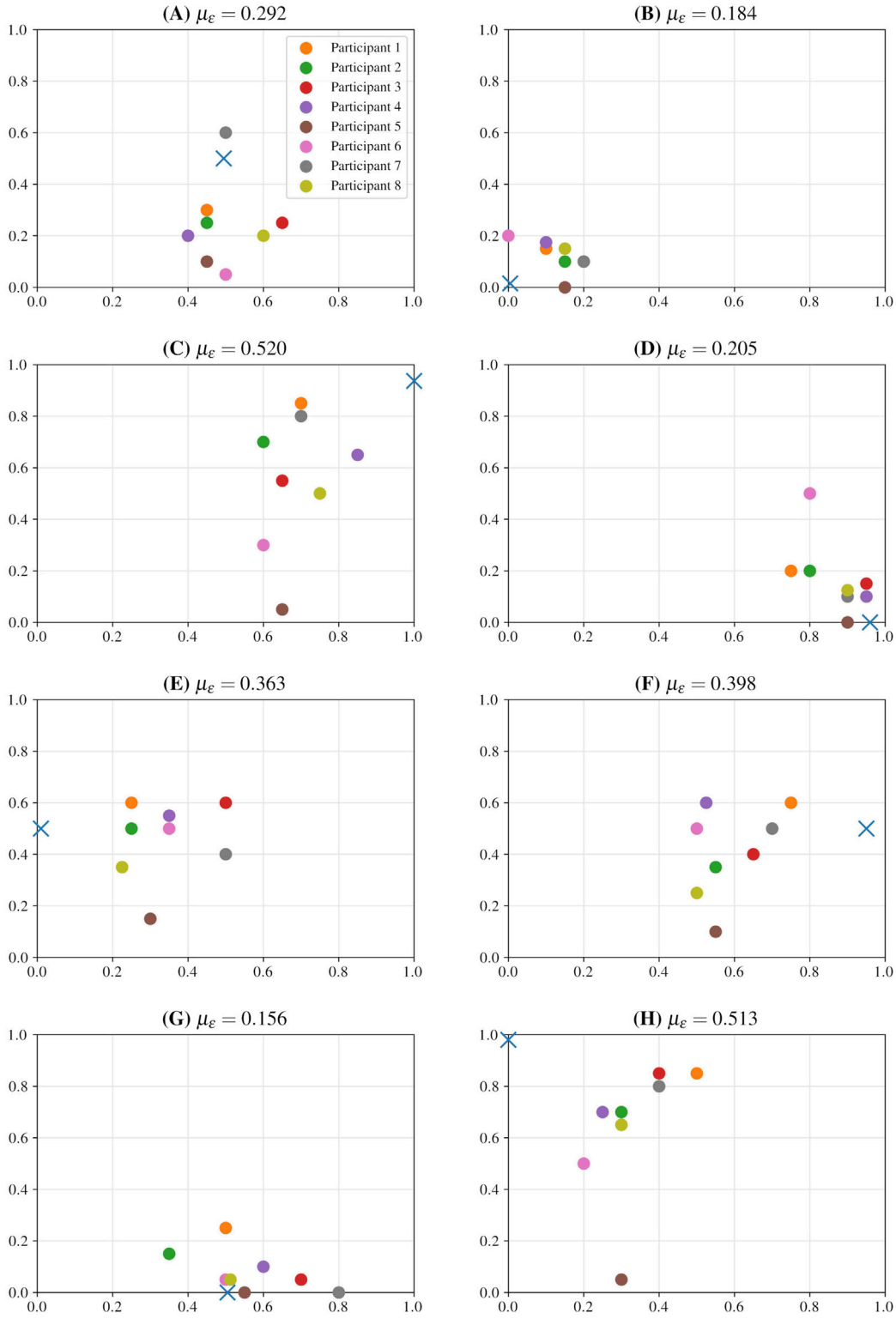

FIGURE 13
Results of a localisation experiment based on WFS virtual primary sources produced by the proposed system. Lateral (horizontal axis) and longitudinal (vertical axis) components are normalised to [0,1]. Each plot represents the virtual sound field; the horizontal axis (i.e., longitudinal component equalling 0) corresponds with the location of the speaker array. Participants stood at a distance of approximately 2 m from the array. Each plot shows the intended position of the virtual sound source as specified by parameters to the WFS plugin interface (cross) and estimated sound source positions as reported by participants (dots). Each plot is labelled with the mean Euclidean error μ_ε between intended position and reported positions. Legend in plot **(A)** also applies to plots **(B)** to **(H)**.

As can be seen, although demonstrating significant outliers (e.g., the position reported by the fifth participant for test **(D)**), certain trends do appear to emerge from the results. Firstly, responses 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 perhaps 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 a tendency in results **(C)** and **(H)** 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.

4.2.1 Discussion

The phasing effect noted by one participant is a consequence of the approach taken to combating jitter and keeping the clients close, temporally, together, and as close to the server as possible. The current approach is likely too aggressive to be viable for high-quality audio output across a broad array of source signals. A transient, unpitched sound source like a snare drum, though perhaps 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.

The above being said, the perceptual test indicates that the system produces virtual sound sources that listeners are, at least to some extent, able to localise. Further, it achieves this at a significantly lower cost-per-channel than any of the systems discussed in sections Section 2.3.3, and, speakers and cables excepted, compares favourably with the OTTOSonics system referred to in Section 2.4, particularly as channel-count increases, i.e., in terms of cost, it has the potential to scale better. The most costly component of the system is the computer, but this could be exchanged for any interested user's personal machine, so long as it is able to run a DAW and has an ethernet interface. The Teensy modules, including audio shield, cost around €45; eight-port ethernet switches can be purchased for as little as €20–30. Assuming a computer costing €1,500, at 16 channels we can estimate around €120/channel, dropping to €50/channel for 64 channels.

5 Conclusion

In this article we have described the development of a novel, distributed system for audio spatialisation. The proposed networked audio system, featuring low-cost, microcontroller-based clients, represents a milestone on the road towards an accessible alternative to state of the art spatial and immersive audio installations. Evaluation of the system exposed the extent of the technical challenges that confront it in its current form, but revealed that it may offer performance sufficient to support timing-critical sound field synthesis techniques.

Client asynchronicity may affect the integrity of the spatial audio algorithm and give rise to audible artefacts, and the strategies presented here for mitigation of asynchronicity call for further refinement. Ultimately, an authoritative source of time should be sought, either in the form of a shared physical clock or a PTP implementation; in light of the disruptive potential of the system, care should be taken, however, to find a solution that is cost-effective and easy to replicate.

Plans for future research include an exploration of other potential hardware platforms for the network client. A successor to Teensy 4.1 may provide more memory or support higher-quality audio, for example. Only briefly considered here, the Raspberry Pi family of embedded Linux systems is priced comparably with the Teensy and supports audio breakout boards capable of 24-bit audio, with the added benefit of significantly greater memory. Further, the Raspberry Pi Compute Module 4 is capable of physical layer timestamping and thus may facilitate the creation of a system synchronised via hardware PTP.

A basic, linear, wave field synthesis algorithm has been demonstrated, implementing virtual primary sound sources; the client implementation and control software should be generalised to support nonlinear speaker arrays, plane and focused sources, and better models for energy absorption. A self-calibrating system, whereby clients *discover* their position in an installation rather than needing to be informed of it, would represent an additional boost to accessibility. Higher order ambisonics, subject to an assessment of its suitability to parallelisation, remains as a worthy target for implementation in future work. Further, convolution-heavy DSP algorithms, such as those supporting virtual acoustics and auralization, may be well-served by the extent of the computational resources afforded by our distributed system.

Data availability statement

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

Ethics statement

Ethical approval was not required for the studies involving humans because participants gave their age, but no other identifying information, and their participation was not filmed or otherwise recorded. The studies were conducted in accordance with the local legislation and institutional requirements. The participants provided their written informed consent to participate in this study.

Author contributions

TR: Conceptualization, Investigation, Methodology, Software, Visualization, Writing–original draft, Writing–review and editing. RM: Conceptualization, Resources, Supervision, Writing–review and editing. SS: Resources, Supervision, Writing–review and editing. TR: Resources, Supervision, Writing–review and editing. SL: Supervision, Writing–review and editing.

Funding

The author(s) declare that financial support was received forthe research, authorship, and/or publication of this article. This project was funded by the FAST ANR project (ANR-20-CE38-0001) and the "moyens incitatifs" program of the Lyon Inria center.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The author(s) declared that they were an editorial board member of Frontiers, at the time of submission. This had no impact on the peer review process and the final decision.

Publisher's note

References

Adriaensen, F. (2005). "Using a DLL to filter time," in Linux audio conference *(Karlsruhe, Germany)*.

Adriaensen, F. (2012). "Controlling adaptive resampling," in 10th International Linux Audio Conference Stanford, CA, USA: CCRMA (Stanford University), 145–151.

Ahrens, J. (2012). Analytic Methods of sound field synthesis. *T-labs series in telecommunication services*. Springer Science and Business Media.

Ahrens, J., Rabenstein, R., and Spors, S. (2008). *The theory of wave field synthesis revisited*. Amsterdam, Netherlands: Audio Engineering Society.

AL-Dhief, F. T., Sabri, N., Latiff, N. A., Malik, N., Abbas, M., Albader, A., et al. (2018). Performance comparison between TCP and UDP protocols in different simulation scenarios. *Int. J. Eng. and Technol.* 7, 172–176. doi:10.14419/ijet.v7i4.36.23739

Baalman, M. A. J., Hohn, T., Schampijer, S., and Koch, T. (2007). "Renewed architecture of the sWONDER software for Wave Field Synthesis on large scale systems," in Proceedings of the 5th int. Linux audio conference *(Berlin, Germany)*.

Bakker, R., Cooper, A., and Kitagawa, A. (2014). *An introduction to networked audio*. Rellingen, Germany: White Paper, Yamaha Commercial Audio Team.

Belloch, J. A., Badía, J. M., Larios, D. F., Personal, E., Ferrer, M., Fuster, L., et al. (2021). On the performance of a GPU-based SoC in a distributed spatial audio system. *J. Supercomput.* 77, 6920–6935. doi:10.1007/s11227-020-03577-4

Berger, J., Farzaneh, N., Murakami, E., and Valentin, L. (2023). "Exploring the past with virtual acoustics and virtual reality," in 2023 Immersive and 3D audio: from Architecture to automotive *(bologna, Italy: IEEE)*.

Berkhout, A. J. (1988). A holographic approach to acoustic control. *J. Audio Eng. Soc.* 36, 977–995.

Berkhout, A. J., de Vries, D., and Vogel, P. (1993). Acoustic control by wave field synthesis. *J. Acoust. Soc. Am.* 93, 2764–2778. doi:10.1121/1.405852

Bosi, M., Servetti, A., Chafe, C., and Rottondi, C. (2021). Experiencing remote classical music performance over long distance: a JackTrip concert between two continents during the pandemic. *J. Audio Eng. Soc.* 69, 934–945. doi:10.17743/jaes.2021.0056

Cáceres, J.-P., and Chafe, C. (2010a). JackTrip: under the hood of an engine for network audio. *J. New Music Res.* 39, 183–187. doi:10.1080/09298215.2010.481361

Cáceres, J.-P., and Chafe, C. (2010b). JackTrip/SoundWIRE meets server farm. *Comput. Music J.* 34, 29–34. doi:10.1162/comj_a_00001

Carôt, A., Hohn, T., and Werner, C. (2009). "Netjack – remote music collaboration with electronic sequencers on the Internet," in Proceedings of the 7th Linux audio conference *(Parma, Italy)*.

Chafe, C. (2018). I am streaming in a room. *Front. Digital Humanit.* 5. doi:10.3389/fdigh.2018.00027

Chafe, C., and Oshiro, S. (2019). "Jacktrip on Raspberry Pi," in *Proceedings of the Linux audio conference 2019* (Stanford, CA, USA: CCRMA: Stanford University).

Chafe, C., Wilson, S., Leistikow, R., Chisholm, D., and Scavone, G. (2000). "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) *(Verona, Italy)*.

Chafe, C., Wilson, S., and Walling, D. (2002). "Physical model synthesis with application to Internet acoustics," in 2002 IEEE international Conference on acoustics, speech, and signal processing *(Orlando, FL, USA: IEEE)*, IV–4056–IV–4059.

Cohen, D. (1977). Specifications for the network voice protocol (NVP). *Tech. Rep. RFC0741*.

Cohen, D. (1981). On holy wars and a plea for Peace. *Computer* 14, 48–54. doi:10.1109/c-m.1981.220208

Correll, K., and Barendt, N. (2005). "Design considerations for software only implementations of the IEEE 1588 precision time protocol," in Proceedings of the IEEE 1588 conference *(Winterthur, Switzerland)*.

Daniel, J., Moreau, S., and Nicol, R. (2003). "Further investigations of high-order ambisonics and wavefield synthesis for holophonic sound imaging," in *114th convention of the*, 114. Amsterdam, Netherlands: Audio Engineering Society.

Dante (2022). What is Dante? *Audinate | Dante Pro Av. Netw.* Available at: https://www.audinate.com/meet-dante/what-is-dante (Accessed November 01, 2024).

de Bruijn, W. (2004). *Application of wave field synthesis in videoconferencing*. Delft, Netherlands: Technische Universiteit Delft. Ph.D. thesis.

De Poli, G., and Rocchesso, D. (1998). Physically based sound modelling. *Organised Sound.* 3, 61–76. doi:10.1017/s1355771898009182

Devonport, S., and Foss, R. (2019). "The distribution of ambisonic and point source rendering to ethernet AVB speakers," in Proceedings of ICSA 2019 *(ilmenau, Germany)*.

Drioli, C., Allocchio, C., and Buso, N. (2013). "Networked performances and natural interaction via LOLA: low latency high quality A/V streaming system," in *Conference proceedings of the second international conference on information technologies for performing arts, media access and entertainment, ECLAP* (Porto, Portugal: Springer), 240–250.

Edison, J. C., Fischer, M., and White, J. (2002). "IEEE-1588 standard for a precision clock synchronization protocol for networked measurement and control systems," in *Proceedings of the 34th annual precise time and time interval systems and applications meeting* (Reston, VA, USA).

Fischer, V. (2015). Case study: performing band rehearsals on the internet with Jamulus

Frank, M., Zotter, F., and Sontacchi, A. (2015). "Producing 3D audio in ambisonics," in *Audio engineering society 57th international conference* (Hollywood, CA, USA).

Gabrielli, L., Squartini, S., Principi, E., and Piazza, F. (2012). "Networked Beagleboards for wireless music applications," in Proceedings of the 5th European DSP Education and research conference *(amsterdam, The Netherlands)*, 291–295.

Geier, M., Ahrens, J., and Spors, S. (2010). Object-based audio reproduction and the audio scene description format. *Organised Sound.* 15, 219–227. doi:10.1017/s1355771810000324

Grani, F., Di Carlo, D., Madrid Portillo, J., Girardi, M., Paisa, R., Banas, J. S., et al. (2016). "Gestural control of wavefield synthesis," in Sound and music computing conference proceedings *(hamburg, Germany)*.

Hardman, V., Sasse, M. A., Handley, M., and Watson, A. (1995). "Reliable audio for use over the internet," in Proceedings of INET'95 (Honolulu, Hawaii: The Internet Society), 171–178.95

Hardman, V., Sasse, M. A., and Kouvelas, I. (1998). Successful multiparty audio communication over the Internet. *Commun. ACM* 41, 74–80. doi:10.1145/274946.274959

Hildebrand, A. (2014). "AES67-2013: AES standard for audio applications of networks - high-performance streaming audio-over-IP interoperability," in Proceedings of the NAB broadcast engineering conference *(Las Vegas, NV, USA)*.

IEEE (2011). "IEEE Std 802.1BA-2011, IEEE standard for local and metropolitan area networks—audio Video bridging (AVB) systems," in *Tech. rep.* IEEE.

IEEE (2018). "IEEE standard for ethernet (IEEE Std 802.3™-2018 revision of IEEE Std 802.3-2015)," in *Tech. rep.* IEEE.

Kaiser, F. (2011). "Transaural Audio - the reproduction of binaural signals over loudspeakers," in *Universität für Musik und darstellende Kunst*. Graz, Austria: Graz/Institut für Elekronische Musik und Akustik/IRCAM. Ph.D. thesis.

Kerrisk, M. (2023). socket(2) - Linux manual page. Available at: https://man7.org/linux/man-pages/man2/socket.2.html (Accessed November 01, 2024).

Kshemkalyani, A. D., and Singhal, M. (2011). *Distributed computing: principles, algorithms, and systems*. Cambridge University Press.

Lago, N. P., and Kon, F. (2003). "A middleware system for distributed real-time multimedia processing," in *Proceedings of the IX Brazilian symposium on multimedia systems and the WEB*.

Lopez-Lezcano, F. (2012). "From Jack to UDP packets to sound and back," in 10th International Linux Audio Conference (Stanford, CA, USA: CCRMA, Stanford University).

Marouani, H., and Dagenais, M. R. (2008). Internal clock drift estimation in computer clusters. *J. Comput. Netw. Commun.* 2008, e583162. doi:10.1155/2008/583162

Meyer, D., Cotton, M., and Vegoda, L. (2010). IANA Guidelines for IPv4 multicast address assignments. *Request for comments RFC 5771*. Internet Engineering Task Force.

Michon, R., Orlarey, Y., Letz, S., and Fober, D. (2019). "Real time audio digital signal processing with faust and the teensy," in Proceedings of the Sound and music computing conference (SMC-19) *(malaga, Spain)*.

Michon, R., Orlarey, Y., Letz, S., Fober, D., and Roosenburg, D. (2020). "Embedded real-time audio signal processing with faust," in Proceedings of the international faust conference (IFC-20) *(paris, France)*.

Mitterhuber, M., Sharafi, R., and Tomás, E. (2022). Ottosonics. *Tangible Music Lab.* Available at: https://tamlab.kunstuni-linz.at/projects/ottosonics/(Accessed November 01, 2024).

Mueller, R. K. (1971). Acoustic holography. *Proc. IEEE* 59, 1319–1335. doi:10.1109/proc.1971.8407

Nicol, R. (2017). "Sound field," in *Immersive sound* (NY, USA: Routledge), 276–310.

Orlarey, Y., Fober, D., and Letz, S. (2009). FAUST: an efficient functional approach to DSP programming. *New Comput. paradigms Comput. music*, 65–96.

Pulkki, V. (1997). Virtual sound source positioning using vector base amplitude panning. *J. Audio Eng. Soc.* 45, 456–466.

Renaud, A., Carôt, A., and Rebelo, P. (2007). "Networked music performance: state of the art," in 30th AES international Conference on intelligent audio environments *(saariselkä, Finland)*.

Rushton, T. A., Michon, R., and Letz, S. (2023). "A microcontroller-based network client towards distributed spatial audio," in Proceedings of the Sound and music computing conference (SMC-23) *(stockholm, Sweden)*.

Sacchetto, M., Servetti, A., and Chafe, C. (2021). "JackTrip-WebRTC: networked music experiments with PCM stereo audio in a Web browser," in *Proceedings of the International web audio Conference* (Barcelona, Spain: UPF). WAC '21.

Schiavoni, F. L., Queiroz, M., and Wanderley, M. M. (2013). "Alternatives in network transport protocols for audio streaming applications," in Proceedings of the international computer music conference *(perth, Australia)*.

Schulzrinne, H. (1992). "Voice communication across the Internet: a network voice terminal,". Amherst, MA, USA: University of Massachusetts at Amherst, Department of Computer and Information Science.

Tongzhou, W., and Lunhui, D. (2022). "Research and implementation of high precision clock synchronization of network audio system based on FPGA and 10-gigabit ethernet," in *Proceedings of the 5th international conference on information systems and computer aided education (ICISCAE)* (China: IEEE: Dalian), 154–161.

Turchet, L., and Fischione, C. (2021). Elk audio OS: an open source operating system for the internet of musical things. *ACM Trans. Internet Things* 2 (12), 1–18. doi:10.1145/3446393

Turchet, L., and Tomasetti, M. (2023). "Immersive networked music performance systems: identifying latency factors," in 2023 Immersive and 3D audio: from Architecture to automotive *(bologna, Italy: IEEE)*.

Turletti, T. (1994). The INRIA videoconferencing system (IVS). *ConeXions* 8.

Verheijen, E. N. G. (1998). Sound Reproduction by wave field synthesis. *Ph.D. Thesis*. Delft, Netherlands: Technical University Delft.

Winter, F., Ahrens, J., and Spors, S. (2018). "A geometric model for spatial aliasing in wave field synthesis," in *Proceedings of the German annual conference on acoustics (DAGA)* (Munich, Germany).

Xu, A., Woszczyk, W., Settel, Z., Pennycook, B., Rowe, R., Galanter, P., et al. (2000). Real-time streaming of multichannel audio data over internet. *J. Audio Eng. Soc.* 48, 627–641.

Ziemer, T. (2020). "Wave field synthesis," in *Psychoacoustic music sound field synthesis* Current Research in Systematic Musicology. Publishing Springer International, 203–243.