A Microcontroller-Based Network Client Towards Distributed Spatial Audio

Thomas Albert Rushton, Romain Michon, Stéphane Letz

A MICROCONTROLLER-BASED NETWORK CLIENT TOWARDS DISTRIBUTED SPATIAL AUDIO

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

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

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

ABSTRACT

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

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

1. INTRODUCTION

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

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

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

2. BACKGROUND

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

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

Networked audio presents a unique challenge amongst modes of real-time network communication due to the

short timescales and fine margins at play [9]. Though packet loss can be monitored relatively easily, jitter, no less deleterious to the quality of an audio signal under transmission, may occur suddenly and sporadically, and can be difficult to monitor and quantify. Perhaps as a by-product of these difficulties, pioneering work on AoE featured creative applications of networked audio; Chafe et al. [13, 14] essentially treated the internet as a resonant medium, using the network as a digital waveguide with their SoundWIRE-based *Network Harp*. This approach served as a way of *sonifying* jitter, thus providing an intuitive, auditory quality of service (QoS) measure.

2.1 Protocols for Networked Audio

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

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

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

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

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

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

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

2.2 Distributed, Networked Audio Systems

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

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

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

Centralised approaches may, however, offer advantages in terms of reliability — distributed systems may have more potential points of failure — and synchronicity

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

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

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

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

— distributed audio systems are subject to timing discrepancies, as each node in such a system has its own clock. The latter point is particularly pertinent in the case of timing-critical applications such as spatial audio; Belloch et al. relied on an external Network Time Protocol (NTP) clock to synchronise the nodes in their WFS implementation.

2.3 Wave Field Synthesis

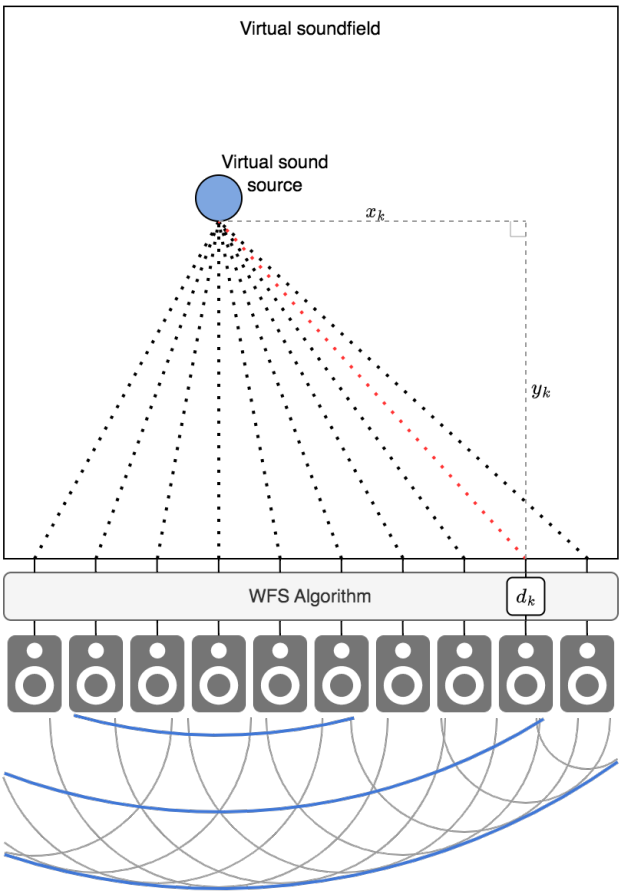


Figure 1: The basic principle of WFS. A sound source is situated within a virtual soundfield; dotted lines indicate simulated distances from an array of secondary point sources. x, y position and delay d indicated with respect to the kth secondary point source. With appropriate timing, a wavefront can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

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

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

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

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

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

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

3. DEVELOPMENT

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

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

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

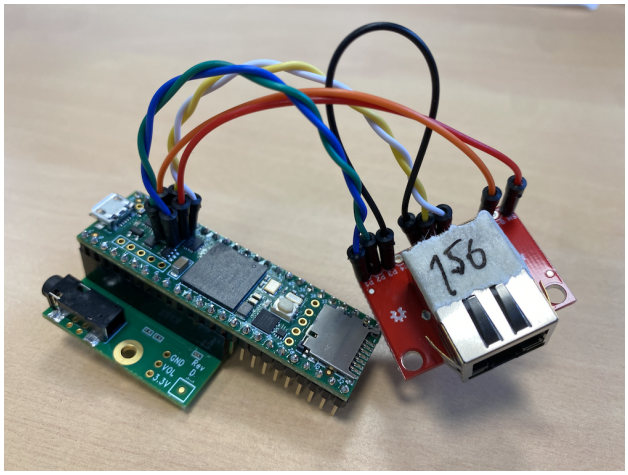


Figure 2: A hardware module, featuring Teensy 4.1 microcontroller, audio shield, and Ethernet add-on, labelled with the last octet of its IP address.

3.1 Hardware Setup

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

3.2 Audio Protocol

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

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

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

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

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

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

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

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

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

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

3.3 Challenges

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

3.3.1 Clock Drift

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

Due to Teensy's threadless architecture, it was deemed most sensible to check for an incoming UDP packet once per audio hardware interrupt. In a perfectly synchronous world this would be fine; in reality, however, the rate of

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

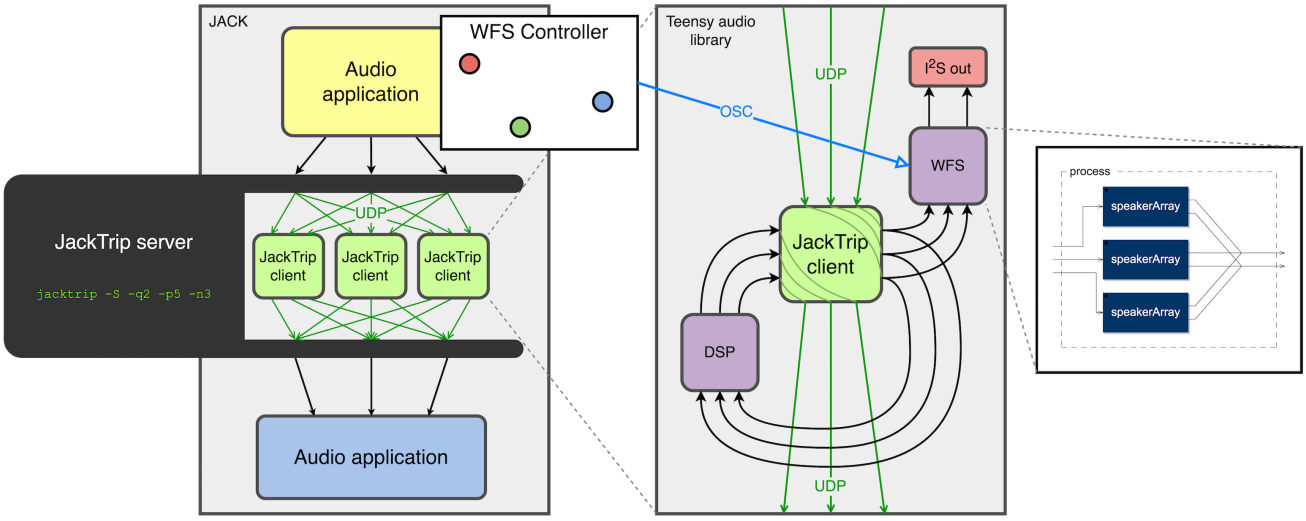


Figure 3: Overview of a networked audio system incorporating the microcontroller-based JackTrip client and distributed WFS system described in this work. Connections between components are illustrative of one potential configuration.

packet arrival does not perfectly correspond with the occurrence of Teensy's audio interrupt, so the number of available packets may be zero, or greater than one. The solution to this issue was to establish a circular buffer at the client side, and read *greedily* from the network, consuming all available packets at each interrupt. This approach guarantees an increase in latency, though for a low audio buffer size this increase will be modest. Indeed, for a technique such as WFS, synchronisation is of course of far more pressing concern.

This strategy was not sufficient to handle situations where the Teensy might fall far behind (or run far ahead of) the server, in which case the circular buffer read and write positions would eventually overlap. To mitigate this, a form of adaptive resampling was implemented: the ratio of circular buffer writes to reads was taken every 1000 writes, and from that a non-integer read-position increment derived, designed to keep the now fractional read position at a manageable interval behind the integer write index. Cubic interpolation was used with respect to the read position to compute an appropriate sample value to use for output. This differs from JackTrip's own strategies for handling circular buffer underrun and overflow conditions [17, §2.2], and aims to ensure audio output free from the audible artefacts of large read-position adjustments or circular buffer resets.

3.3.2 Jitter

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

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

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

3.4 The WFS Algorithm

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

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

component may be found:

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

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

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

which has the advantage of limiting the maximum delay line length to the time taken for sound to traverse the speaker array, and thus minimises the memory footprint of the algorithm. Equation (4) produces a non-integer value for $d_{i,k}$ so the algorithm employs linearly-interpolated fractional delay lines via Faust's `de.fdelay` function. [10]

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

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

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

4. RESULTS

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

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

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

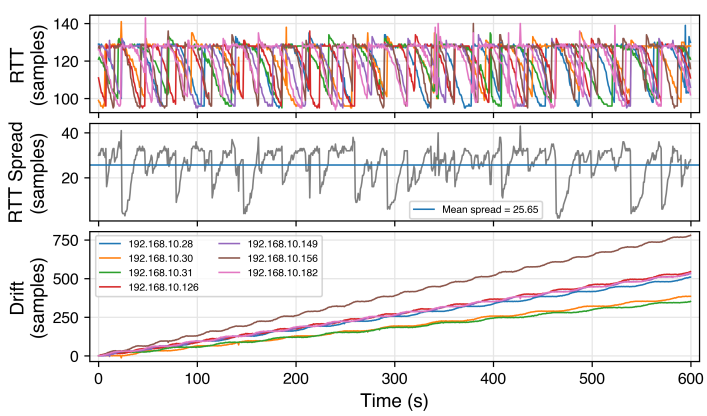
These results suggest that the system as demonstrated may have performed well enough to create a convincing spatial effect at least some of the time, but that there is scope for improvement. Speakers were spaced at intervals of 0.23 m, a distance travelled by sound in air in ~671 µs or ~30 samples at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

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

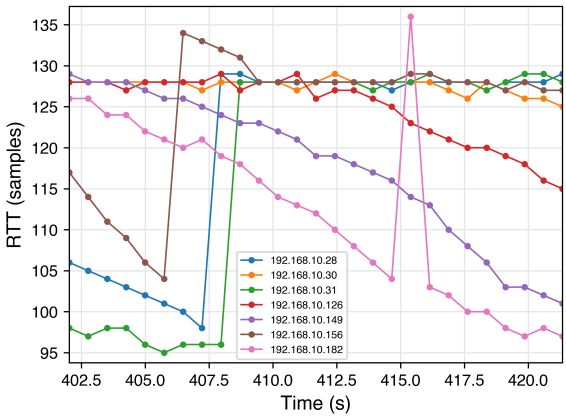
5. CONCLUSION

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

[10] `https://faustlibraries.grame.fr/libs/delays/#defdelay`
[11] JUCE 7.0.2 `https://github.com/juce-framework/JUCE`
[12] Only one eight-port ethernet switch was available for this test, one port of which was reserved for connection to the JackTrip server, so it was possible to test only seven of the eight modules.



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



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

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

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

The WFS model implemented is fairly rudimentary and calls for extension to other types of virtual sound sources and speaker array configurations. Ambisonics would be an ideal follow-up implementation to attempt, but an investigation of the feasibility of distributing an ambisonics algorithm would be required. Nevertheless, there may be other spatial audio techniques, and DSP algorithms in general, that would find a good home in a distributed setting.

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

Acknowledgments

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

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

6. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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